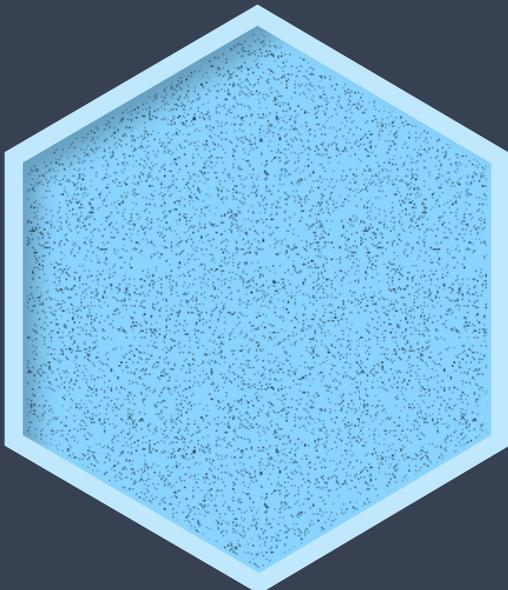


GOING FROM BEGINNER TO ADVANCED IN THE TIDYVERSE

10 chapter and 23 tutorials to boost your
Tidyverse skills



CHRISTIAN BURKHART

Table of contents

Preface	7
I Improve reading files	9
1 How to improve reading files with the <code>read_*</code> functions	10
1.1 Converting column names to lowercase	11
1.2 Replacing and removing character strings in your column names	12
1.3 Using a specific naming convention for column names	12
2 How to read many files into R	15
2.1 How to create a character vector of file paths?	16
2.2 How to read the files into R from a character vector of paths	17
2.3 But what if the column names of the files are not consistent?	19
2.4 What if the files are not in the same folder?	22
2.5 What if I don't need some of these files?	25
II Improve working with columns	29
3 How to select variables with the <code>tidyselect</code> functions	30
3.1 Selecting every column	31
3.2 Selecting the last column	32
3.3 Selecting columns that start and end with a specific string	33
3.4 Selecting columns that contain certain strings	34
3.5 Selecting columns with regular expressions	35
3.6 Select columns with number ranges	36
3.7 Selecting columns of a specific type	37
3.8 Combining selections	38
4 How to rename many column names at once	40
4.1 The <code>rename_with</code> function	41
4.2 How to use <code>rename_with</code> to replace characters	42
4.3 How to rename variables for specific variables	44

III Improve creating and modifying variables	47
5 How to count with count, add_count, and add_tally	48
5.1 How to count with continuous variables	49
5.2 How to calculate the sum of a variable based on groups without using group_by.	51
5.3 How to add counts as a variable to your data frame	52
5.4 How to add a new variable to your data frame that contains the sum of a specific variable	53
6 How to use extract to create multiple columns from one column	54
6.1 How to extract a simple character column	55
6.2 How to extract more complicated character column	58
7 How to anonymize columns	63
7.1 What is the difference between pseudonymization and anonymization?	64
7.2 How to anonymize a factor column with fct_anon	64
7.3 How to replace names with random names	65
7.4 How to mask values (e.g. credit card numbers)	67
7.5 How to turn ages into groups	69
7.6 How to remove house numbers from street names	72
7.7 How to encrypt and decript columns	73
8 How to lump factor levels	76
8.1 fct_lump_min: How to lump levels that occur no more than min times	78
8.2 fct_lump_n: Lumps n of the most or least frequent levels	80
8.3 fct_lump_prop: Lumps levels that appear no more than n * prop times	86
8.4 fct_lump_lowfreq: Lumps the least frequent levels	88
9 How to order factor levels	91
9.1 How to order factor levels manually	91
9.2 How to order the levels based on how frequently each level occurs	95
9.3 How to order the levels based on the values of a numeric variable	99
9.4 How to order levels based on the values of two numeric variables	110
10 How to apply a function across many columns	115
10.1 across and summarise	116
10.2 across and mutate	122
IV Improve working with rows	130
11 How to filter rows based on multiple columns	131
11.1 How to filter rows based on a condition across mulitple columns	133
11.2 How to filter rows that contain missing values	136

11.3 How to create new columns based on conditions across multiple columns	137
12 How to improve slicing rows	139
12.1 Overview of the <code>slice</code> function	139
12.2 How to slice off the top and bottom of a data frame	142
12.3 How to slice rows with the highest and lowest values in a given column	144
12.4 How to combine the slice functions with <code>group_by</code>	146
12.5 How to create bootstraps with <code>slice_sample</code>	148
13 How to do rowwise calculations	155
13.1 Introducing <code>rowwise</code>	157
13.2 How to use <code>tidyselect</code> functions with <code>rowwise</code>	159
13.3 Don't forget to <code>ungroup()</code>	161
13.4 Calculating proportions with <code>rowwise</code>	163
13.5 If you care about performance, choose alternativ approaches	164
13.6 A short deep-dive into <code>pmax</code> and <code>pmin</code>	166
13.7 A deep-dive into <code>rowSums</code> and <code>rowMeans</code>	166
V Improve grouping data	169
14 How to run many models with the new dplyr grouping functions	170
14.1 Building a single linear model	172
14.2 The split > apply > combine technique	174
14.3 Split > apply > combine for running many models	176
14.4 The Wild West of split > apply > combine for running many models	179
14.5 <code>group_by</code> > <code>group_map</code> > <code>map_dfr</code>	180
14.6 <code>group_by</code> > <code>group_modify</code> > <code>ungroup</code>	183
14.7 <code>split</code> > <code>map2_dfr</code>	187
14.8 <code>group_split</code> > <code>map_dfr</code>	189
14.9 <code>group_nest</code> > <code>mutate/map</code> > <code>unnest</code>	190
14.10Conclusion	194
VI Improve working with incomplete data	195
15 How to expand data frames and create complete combinations of values	196
15.1 <code>complete</code>	197
15.2 <code>expand</code>	199
15.3 <code>expand/complete</code> with <code>group_by</code>	202
15.4 <code>expand</code> with <code>nesting</code>	209
15.5 <code>crossing</code>	210

VII Improve converting data frames between long and wide formats	213
16 How to make a data frame longer	214
16.1 Column headers are values of one variable, not variable names	216
16.2 Multiple variables are stored in columns	223
16.3 Multiple variables are stored in one column	227
16.4 Variables are stored in both rows and columns	232
17 How to make a data frame wider	235
17.1 How to use <code>pivot_wider</code> (the simplest example)	236
17.2 How to use <code>pivot_wider</code> to calculate ratios/percentages	239
17.3 How to use <code>pivot_wider</code> to create tables of summary statistics	242
17.4 How to make data frames wider for use in other software tools	245
17.5 How to deal with multiple variable names stored in a column	249
17.6 How to use <code>pivot_wider</code> to one-hot encode a factor	252
17.7 How to use <code>pivot_wider</code> without an id column	255
VIII Improve your tidyverse fundamentals	258
18 How to make use of curly curly inside functions	259
18.1 What is curly curly and the <code>rlang</code> package?	259
18.2 What is tidy evaluation?	260
18.3 What is data masking?	262
18.4 What is curly curly <code>\{\}\{}</code> ?	262
18.5 How to pass multiple arguments to a function with the dot-dot-dot argument .	265
IX Improve your purrr skills	269
19 How to use the map function family effectively	270
19.1 Preparation	270
19.2 A list, vector and data frame primer	270
19.3 For loops in R	281
19.4 Introduction of the map family of functions	282
19.5 Using map with a list as input	286
19.6 Error handling with <code>safely</code> and <code>possibly</code>	288
19.7 <code>map_vec</code>	291
19.8 Using map with nested data frames	293
19.9 Using map with a list of data frames	298
20 How to use the map2 and pmap function family effectively	301
20.1 An overview of <code>map2</code> and <code>pmap</code>	301
20.2 Introduction to <code>map2</code>	301

20.3 Introduction to <code>pmap</code>	307
20.4 The dot-dot-dot argument in <code>pmap</code>	311
20.5 <code>pmap</code> or <code>map2?</code>	314
20.6 Use Case 1: Creating plots from a nested tibbles	315
20.7 Use Case 2: Fitting models	318
20.8 Use Case 3: p-hacking	320
21 How to use the <code>walk</code> function family effectively	324
21.1 What are side-effects?	324
21.2 Educational examples of <code>walk</code>	326
21.3 Saving plots to disk with <code>pwalk</code>	328
21.4 Creating folders based on information within a data frame	333
21.5 Organize images into folders based on metadata information	334
21.6 Intermediate tests	340
X Improve your code's performance	342
22 How to utilize your computer's parallel processing capabilities using <code>future</code> and <code>furrr</code>	343
22.1 An analogy on parallel processing	345
22.2 The <code>future</code> package	345
22.3 The <code>furrr</code> package	351
22.4 Simulating the speed of <code>furrr</code>	352
23 How to speed up your data analysis with <code>dplyr</code>	358

Preface

This ebook, available at <https://christianb.gumroad.com/l/tidyverse-booster>, is the outcome of a 10-month project that began in July 2022. Over the course of the project, I wrote 23 tutorials on various topics related to the Tidyverse packages until March 2022, resulting in a 360-page reference book for Tidyverse programmers.

My aim with this project was to compile all the tips and tricks that advanced Tidyverse users should know. I have spent over 300 hours to gather this information in one place so that my readers won't have to go through numerous tutorials to acquire the same knowledge.

The Tidyverse community provided me with valuable insights into specific use cases and tips, which significantly influenced the book's creation. I have acknowledged and credited these individuals throughout the book and owe them my sincerest gratitude.

As this is the book's first edition, I cannot guarantee that it is entirely free of typos and errors. Consider it a first draft that I plan to refine and improve over time. If you come across any mistakes while reading, please don't hesitate to contact me.

All the following code was generated with R version 4.2.1 (2022-06-23) on March 18, 2023 with this session:

```
R version 4.2.1 (2022-06-23)
Platform: aarch64-apple-darwin20 (64-bit)
Running under: macOS Monterey 12.1

Matrix products: default
BLAS:      /Library/Frameworks/R.framework/Versions/4.2-arm64/Resources/lib/libRblas.0.dylib
LAPACK:   /Library/Frameworks/R.framework/Versions/4.2-arm64/Resources/lib/libRlapack.dylib

locale:
[1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8

attached base packages:
[1] stats      graphics   grDevices  utils      datasets   methods    base

other attached packages:
[1]forcats_0.5.2  stringr_1.5.0  dplyr_1.1.0    purrr_1.0.1
```

```
[5] readr_2.1.2      tidyrr_1.3.0      tibble_3.1.8      ggplot2_3.3.6
[9] tidyverse_1.3.2

loaded via a namespace (and not attached):
 [1] tidyselect_1.2.0    xfun_0.32          haven_2.5.1
 [4] gargle_1.2.0       colorspace_2.0-3   vctrs_0.5.2
 [7] generics_0.1.3     htmltools_0.5.3   yaml_2.3.5
[10] utf8_1.2.2        rlang_1.0.6       pillar_1.8.1
[13] withr_2.5.0       glue_1.6.2        DBI_1.1.3
[16] dbplyr_2.2.1      readxl_1.4.1    modelr_0.1.9
[19] lifecycle_1.0.3    munsell_0.5.0    gtable_0.3.0
[22] cellranger_1.1.0  rvest_1.0.3      memoise_2.0.1
[25] evaluate_0.16     knitr_1.40      tzdb_0.3.0
[28] fastmap_1.1.0    fansi_1.0.3     broom_1.0.1
[31] backports_1.4.1   scales_1.2.1    googlesheets4_1.0.1
[34] cachem_1.0.6     jsonlite_1.8.0   fs_1.5.2
[37] hms_1.1.2         conflicted_1.1.0 digest_0.6.29
[40] stringi_1.7.8    grid_4.2.1       cli_3.5.0
[43] tools_4.2.1       magrittr_2.0.3   crayon_1.5.1
[46] pkgconfig_2.0.3   ellipsis_0.3.2  xml2_1.3.3
[49] reprex_2.0.2      googledrive_2.0.0 lubridate_1.8.0
[52] assertthat_0.2.1  rmarkdown_2.16   httr_1.4.4
[55] rstudioapi_0.14   R6_2.5.1        compiler_4.2.1
```

Part I

Improve reading files

1 How to improve reading files with the `read_*` functions

What will this tutorial cover?

In this tutorial you will learn how to clean column names, replace strings from column names, and select columns directly in the `read_*` functions.

Who do I have to thank?

I came across this trick from a [blog post by Jim Hester on tidyverse.org](#). If you want to dive deeper, read the fantastic blog post.

It is rare to read a CSV file without any data cleaning. Suppose I want to convert the column names of this CSV file to lowercase and select only columns that start with the letter “m”:

```
MANUFACTURER,MODEL,DISPL,YEAR,CYL,TRANS,DRV,CTY,HWY,FL,CLASS
audi,a4,1.8,1999,4,auto(15),f,18,29,p,compact
audi,a4,1.8,1999,4,manual(m5),f,21,29,p,compact
audi,a4,2,2008,4,manual(m6),f,20,31,p,compact
audi,a4,2,2008,4,auto(av),f,21,30,p,compact
audi,a4,2.8,1999,6,auto(15),f,16,26,p,compact
audi,a4,2.8,1999,6,manual(m5),f,18,26,p,compact
```

Most of us would probably read the CSV file first and then do the data cleaning. For example, using the `clean_names` function from the `janitor` package (fyi: I use the `show_col_types` argument here to hide the output. You don’t need to use this argument):

```
library(tidyverse)
library(janitor)

mpg_new <- read_csv("data/mpg_uppercase.csv",
                     show_col_types = FALSE) %>%
  clean_names() %>%
  select(c(manufacturer, model)) %>%
```

```
glimpse()
```

```
Rows: 234
Columns: 2
$ manufacturer <chr> "audi", "audi", "audi", "audi", "audi", "audi", "~
$ model           <chr> "a4", "a4", "a4", "a4", "a4", "a4", "a4 quattro", "~
```

This approach is perfectly fine. It turns out, however, that the `read_*` functions have some data cleaning arguments built in. These arguments don't allow you to do something new, but they do allow you to encapsulate the reading of the data with the data cleaning. Let's see how.

1.1 Converting column names to lowercase

In my previous example, I have used the `clean_names` function from the `janitor` package to convert the column names to lowercase. The same can be achieved inside `read_csv` with the function `make_clean_names` for the `name_repair` argument:

```
read_csv("data/mpg_uppercase.csv",
         show_col_types = FALSE,
         name_repair = make_clean_names) %>%
glimpse()
```

```
Rows: 234
Columns: 11
$ manufacturer <chr> "audi", "audi", "audi", "audi", "audi", "audi", "~
$ model           <chr> "a4", "a4", "a4", "a4", "a4", "a4", "a4 quattro", "~
$ displ            <dbl> 1.8, 1.8, 2.0, 2.0, 2.8, 2.8, 3.1, 1.8, 1.8, 2.0, 2.0, 2.~
$ year             <dbl> 1999, 1999, 2008, 2008, 1999, 1999, 2008, 1999, 1999, 200~
$ cyl               <dbl> 4, 4, 4, 4, 6, 6, 4, 4, 4, 6, 6, 6, 6, 6, 8, 8, ~
$ trans              <chr> "auto(15)", "manual(m5)", "manual(m6)", "auto(av)", "auto~
$ drv                <chr> "f", "f", "f", "f", "f", "f", "4", "4", "4", "4", "4", "4~
$ cty                <dbl> 18, 21, 20, 21, 16, 18, 18, 18, 16, 20, 19, 15, 17, 17, 1~
$ hwy                <dbl> 29, 29, 31, 30, 26, 26, 27, 26, 25, 28, 27, 25, 25, 25, 2~
$ fl                 <chr> "p", "p~
$ class              <chr> "compact", "compact", "compact", "compact", "compact", "c~
```

As you can see, I use the `make_clean_names` function here and not `clean_names`. This is because `clean_names` does not work with vectors, but `make_clean_names` does.

1.2 Replacing and removing character strings in your column names

With `make_clean_names` you can also replace certain characters from the column names. Suppose we want to replace the character “%” with the actual word “_percent”:

```
make_clean_names(c("A", "B%", "C"),
                 replace = c("%" = "_percent"))
```

```
[1] "a"           "b_percent" "c"
```

If you are familiar with regular expressions, you can make more complex replacements. For example, you could remove the underscore for all column names that start with the letter “A”:

```
make_clean_names(c("A_1", "B_1", "C_1"),
                 replace = c("^A_" = "a"))
```

```
[1] "a1"    "b_1"   "c_1"
```

1.3 Using a specific naming convention for column names

You may have noticed that in the last example `make_clean_names` converted the column names to lowercase. That’s because the function uses the snake naming convention by default. Snake converts all names to lowercase and separates words with an underscore:

```
make_clean_names(c("myHouse", "MyGarden"),
                 case = "snake")
```

```
[1] "my_house"  "my_garden"
```

If you do not want to change the naming convention of your column names at all, use “none” for the case:

```
make_clean_names(c("myHouse", "MyGarden"),
                 case = "none")
```

```
[1] "myHouse"  "MyGarden"
```

Here is a list of all naming conventions you can use:

Table 1.1: Naming conventions

Naming Convention	example1	example2
snake	myHouse -> my_house	MyGarden -> my_garden
small_camel	myHouse -> myHouse	MyGarden -> myGarden
big_camel	myHouse -> MyHouse	MyGarden -> MyGarden
screaming_snake	myHouse -> MY_HOUSE	MyGarden -> MY_GARDEN
parsed	myHouse -> my_House	MyGarden -> My_Garden
mixed	myHouse -> my_House	MyGarden -> My_Garden
lower_upper	myHouse -> myHOUSE	MyGarden -> myGARDEN
upper_lower	myHouse -> MYhouse	MyGarden -> MYgarden
swap	myHouse -> MYhOUSE	MyGarden -> mYgARDEN
all_caps	myHouse -> MY_HOUSE	MyGarden -> MY_GARDEN
lower_camel	myHouse -> myHouse	MyGarden -> myGarden
upper_camel	myHouse -> MyHouse	MyGarden -> MyGarden
internal_parsing	myHouse -> my_House	MyGarden -> My_Garden
none	myHouse -> myHouse	MyGarden -> MyGarden
flip	myHouse -> MYhOUSE	MyGarden -> mYgARDEN
sentence	myHouse -> My house	MyGarden -> My garden
random	myHouse -> MyhoUSe	MyGarden -> MYGaRDEN
title	myHouse -> My House	MyGarden -> My Garden

For example, in our dataset we could change the column names to `upper_camel`:

```
read_csv("data/mpg_uppercase.csv",
        show_col_types = FALSE,
        name_repair = ~ make_clean_names(., case = "upper_camel")) %>%
glimpse()
```

```
Rows: 234
Columns: 11
$ Manufacturer <chr> "audi", "audi", "audi", "audi", "audi", "audi", "~"
$ Model <chr> "a4", "a4", "a4", "a4", "a4", "a4", "a4 quattro", "~"
$ Displ <dbl> 1.8, 1.8, 2.0, 2.0, 2.8, 2.8, 3.1, 1.8, 1.8, 2.0, 2.0, 2.~
$ Year <dbl> 1999, 1999, 2008, 2008, 1999, 1999, 2008, 1999, 1999, 200~
$ Cyl <dbl> 4, 4, 4, 4, 6, 6, 4, 4, 4, 6, 6, 6, 6, 6, 8, 8, ~
$ Trans <chr> "auto(15)", "manual(m5)", "manual(m6)", "auto(av)", "auto~
$ Drv <chr> "f", "f", "f", "f", "f", "f", "f", "4", "4", "4", "4", "4", "4~
$ Cty <dbl> 18, 21, 20, 21, 16, 18, 18, 18, 16, 20, 19, 15, 17, 17, 1~
```

```
$ Hwy      <dbl> 29, 29, 31, 30, 26, 26, 27, 26, 25, 28, 27, 25, 25, 25, 2~  
$ F1       <chr> "p", "p~  
$ Class    <chr> "compact", "compact", "compact", "compact", "compact", "c~
```

The dot . in `make_clean_names` denotes the vector of column names.

1.3.1 Selecting specific columns

Apart from cleaning your column names, you can also select columns directly from `read_csv` using the `col_select` argument:

```
read_csv("data/mpg_uppercase.csv",  
        show_col_types = FALSE,  
        name_repair = make_clean_names,  
        col_select = c(manufacturer, model)) %>%  
glimpse()
```

```
Rows: 234  
Columns: 2  
$ manufacturer <chr> "audi", "audi", "audi", "audi", "audi", "audi", "a~  
$ model         <chr> "a4", "a4", "a4", "a4", "a4", "a4", "a4 quattro", "a~
```

In this example, I have explicitly selected the columns. You can also use the tidyselect functions, which we will cover in another tutorial in this course.

i Summary

Here's what you can take away from this tutorial:

- You can clean column names, replace strings in columns, and select columns directly in your `read_*` functions
- With the function `make_clean_names` you can convert your column names to certain naming conventions (e.g., small camel)
- You can use regular expressions with the `replace` argument in `make_clean_names` to remove or replace specific characters from your column names
- You can use tidyselect functions to select a subset of columns with the `col_select` argument

2 How to read many files into R

What will this tutorial cover?

In this tutorial you will learn how to read many files into R. You will learn how to create a character vector of file paths and then read them into R. You will also be able to work with some edge cases that you might stumble upon.

Who do I have to thank?

I have to thank [Beatriz Milz](#) who asked me to write a tutorial on this topic. I also want to thank Jim Hester for his [excellent blog post](#) about `readr`. He showed me that I don't need `map_dfr` to read many files into R, but can achieve the same results with the `read_*` functions.

You don't always read just one file into R. It is not uncommon for your data to be scattered in hundreds or thousands of files. Of course, you don't want to read these files into R manually. So you need an automatic method for reading in files.

To illustrate how this works in tidyverse, lets create 25 CSV files. How we create the CSV files is not important here. Essentially, I sample 20 rows from the mpg dataset 25 times and save these data frames to disk. Also, at the beginning of the code, I create a new directory named `many_files` where the CSV files will be stored.

```
library(tidyverse)
library(fs) # install.packages("fs")
# Create dir
dir_create(c("many_files"))

mpg_samples <- map(1:25, ~ slice_sample(mpg, n = 20))

iwalk(mpg_samples, ~ write_csv(., paste0("many_files/", .y, ".csv")))
```

Later in this course we will discuss the `iwalk` function in more detail. If you look in the `many_files` directory, you should now see 25 CSV files.

2.1 How to create a character vector of file paths?

Before we can read the files into R, we need to create a character vector of the file paths. You have several options to create such a vector. You can use the R base function `list.files`, which returns character vectors of the names of files in a directory or you use the function `dir_ls` from the `fs` package. Let's start with `list.files`:

```
(csv_files_list_files <- list.files(path = "many_files",
                                     pattern = "csv", full.names = TRUE))
```

```
[1] "many_files/1.csv"  "many_files/10.csv" "many_files/11.csv"
[4] "many_files/12.csv" "many_files/13.csv" "many_files/14.csv"
[7] "many_files/15.csv" "many_files/16.csv" "many_files/17.csv"
[10] "many_files/18.csv" "many_files/19.csv" "many_files/2.csv"
[13] "many_files/20.csv" "many_files/21.csv" "many_files/22.csv"
[16] "many_files/23.csv" "many_files/24.csv" "many_files/25.csv"
[19] "many_files/3.csv"  "many_files/4.csv"  "many_files/5.csv"
[22] "many_files/6.csv"  "many_files/7.csv"  "many_files/8.csv"
[25] "many_files/9.csv"
```

The function has a number of arguments. With `path` you specify where to find your files. Since the path is relative, make sure you are either working in an R-Studio project or have defined your working directory. `pattern` receives a regular expression. In this case, we said that the file should contain the string “`csv`”. Finally, the `full.names` argument indicates that we want to store the full paths of the files, not just the file names. If you do not set this argument to `TRUE`, you will have trouble reading in your files later.

The other option is to use the `dir_ls` function from the `fs` package. `fs` provides a cross-platform interface for accessing files on your hard disk. It supports all file operations (deleting, creating files, moving files, etc.).

```
# library(fs) # install.packages("fs")
(csv_files_dir_ls <- dir_ls(path = "many_files/",
                            glob = "*.csv", type = "file"))
```

```
many_files/1.csv  many_files/10.csv many_files/11.csv many_files/12.csv
many_files/13.csv  many_files/14.csv many_files/15.csv many_files/16.csv
many_files/17.csv  many_files/18.csv many_files/19.csv many_files/2.csv
many_files/20.csv  many_files/21.csv many_files/22.csv many_files/23.csv
many_files/24.csv  many_files/25.csv many_files/3.csv  many_files/4.csv
many_files/5.csv   many_files/6.csv  many_files/7.csv  many_files/8.csv
many_files/9.csv
```

The results are the same as above. Again, you specify the path where your files are stored. The glob argument is used to specify the file type of your files. With `type` you indicate that you are looking for a file and not a folder or something else.

2.2 How to read the files into R from a character vector of paths

Now that we know the file paths, we can load the files into R. The tidyverse way to do this is to use the `map_dfr` function from the `purrr` package. `map_dfr` loops through all the file paths and binds the data frames into a single data frame. The `.x` in the following code stands for the file name. To output the actual csv files and not the filenames, we need to put `.x` (the path) in a `read_*` function. In this example we are working with CSV files. The trick works the same for all rectangular file formats.

```
data_frames <- map_dfr(csv_files_dir_ls,
                      ~ read_csv(.x, show_col_types = FALSE))
glimpse(data_frames)
```

```
Rows: 500
Columns: 11
$ manufacturer <chr> "dodge", "ford", "volkswagen", "subaru", "subaru", "dodge~
$ model          <chr> "dakota pickup 4wd", "explorer 4wd", "passat", "forester ~
$ displ           <dbl> 3.9, 5.0, 2.8, 2.5, 2.5, 3.9, 2.5, 6.5, 5.4, 5.7, 2.5, 4.~
$ year            <dbl> 1999, 1999, 1999, 1999, 2008, 1999, 1999, 1999, 2008, 199~
$ cyl              <dbl> 6, 8, 6, 4, 4, 6, 4, 8, 8, 8, 4, 8, 6, 4, 6, 4, 8, 8, ~
$ trans             <chr> "auto(14)", "auto(14)", "auto(15)", "auto(14)", "manual(m~
$ drv                <chr> "4", "4", "f", "4", "4", "4", "4", "r", "r", "4", "4~
$ cty                <dbl> 13, 13, 16, 18, 20, 13, 18, 14, 12, 16, 19, 11, 18, 24, 1~
$ hwy                <dbl> 17, 17, 26, 24, 27, 17, 25, 17, 18, 26, 26, 15, 26, 32, 2~
$ fl                 <chr> "r", "r", "p", "r", "r", "r", "d", "r", "p", "r", "p~
$ class              <chr> "pickup", "suv", "midsize", "suv", "compact", "suv", "suv~
```

Another approach is to use the `read_csv` function directly by putting the character vector of the file names directly into `read_csv`. I found this trick in the [blog post by Jim Hester on tidyverse.org](#). A neat trick is to specify an `id` argument that adds a new column to the data frame indicating which files the data came from:

```
read_csv(csv_files_dir_ls, id = "filename",
         show_col_types = FALSE) %>%
glimpse
```

```

Rows: 500
Columns: 12
$ filename      <chr> "many_files/1.csv", "many_files/1.csv", "many_files/1.csv~
$ manufacturer <chr> "dodge", "ford", "volkswagen", "subaru", "subaru", "dodge~
$ model        <chr> "dakota pickup 4wd", "explorer 4wd", "passat", "forester ~
$ displ         <dbl> 3.9, 5.0, 2.8, 2.5, 2.5, 3.9, 2.5, 6.5, 5.4, 5.7, 2.5, 4.~
$ year          <dbl> 1999, 1999, 1999, 1999, 2008, 1999, 1999, 1999, 2008, 199~
$ cyl           <dbl> 6, 8, 6, 4, 4, 6, 4, 8, 8, 8, 4, 8, 6, 4, 6, 4, 4, 8, 8, ~
$ trans          <chr> "auto(14)", "auto(14)", "auto(15)", "auto(14)", "manual(m~
$ drv            <chr> "4", "4", "f", "4", "4", "4", "4", "r", "r", "4", "4~
$ cty            <dbl> 13, 13, 16, 18, 20, 13, 18, 14, 12, 16, 19, 11, 18, 24, 1~
$ hwy            <dbl> 17, 17, 26, 24, 27, 17, 25, 17, 18, 26, 26, 15, 26, 32, 2~
$ fl             <chr> "r", "r", "p", "r", "r", "r", "d", "r", "p", "r", "p~
$ class          <chr> "pickup", "suv", "midsize", "suv", "compact", "suv", "suv~

```

The first column now specifies the actual file name. We can do the same with our previous approach. We would just have to add a new column with mutate representing the file name:

```

map_dfr(csv_files_dir_ls,
  ~ read_csv(.x, , show_col_types = FALSE) %>%
    mutate(filename = .x)) %>%
glimpse()

```

```

Rows: 500
Columns: 12
$ manufacturer <chr> "dodge", "ford", "volkswagen", "subaru", "subaru", "dodge~
$ model        <chr> "dakota pickup 4wd", "explorer 4wd", "passat", "forester ~
$ displ         <dbl> 3.9, 5.0, 2.8, 2.5, 2.5, 3.9, 2.5, 6.5, 5.4, 5.7, 2.5, 4.~
$ year          <dbl> 1999, 1999, 1999, 1999, 2008, 1999, 1999, 1999, 2008, 199~
$ cyl           <dbl> 6, 8, 6, 4, 4, 6, 4, 8, 8, 8, 4, 8, 6, 4, 6, 4, 4, 8, 8, ~
$ trans          <chr> "auto(14)", "auto(14)", "auto(15)", "auto(14)", "manual(m~
$ drv            <chr> "4", "4", "f", "4", "4", "4", "4", "r", "r", "4", "4~
$ cty            <dbl> 13, 13, 16, 18, 20, 13, 18, 14, 12, 16, 19, 11, 18, 24, 1~
$ hwy            <dbl> 17, 17, 26, 24, 27, 17, 25, 17, 18, 26, 26, 15, 26, 32, 2~
$ fl             <chr> "r", "r", "p", "r", "r", "r", "d", "r", "p", "r", "p~
$ class          <chr> "pickup", "suv", "midsize", "suv", "compact", "suv", "suv~
$ filename       <fs::path> "many_files/1.csv", "many_files/1.csv", "many_files/~

```

2.3 But what if the column names of the files are not consistent?

Unfortunately, we live in a messy world. Not all column names are the same. Let's create a messy dataset with inconsistent column names (we have covered this technique in `make_clean_names` in our last tutorial):

```
mpg_samples <- map(1:10, ~ slice_sample(mpg, n = 20))

inconsistent_dframes <- map(mpg_samples,
                           ~ janitor::clean_names(dat = .x, case = "random"))
```

The column names of these 10 data frames consist of the same names, but are randomly written in upper or lower case.

```
map(inconsistent_dframes, ~ colnames(.x)) %>%
  head

[[1]]
[1] "MANUFActuRER" "MoDEL"          "dispL"        "Year"        "cyL"
[6] "trANS"          "DrV"           "cty"         "hWy"         "FL"
[11] "cLAss"

[[2]]
[1] "mAnufACTuRER" "ModEL"          "DiSpL"        "YeaR"        "CYL"
[6] "trANS"          "DRv"           "CTY"         "hwy"         "fL"
[11] "cLass"

[[3]]
[1] "ManuFAcTUreR" "mODEL"          "dIspl"        "YEar"        "cyl"
[6] "tRaNs"          "dRv"            "CTY"         "hwy"         "FL"
[11] "CLASs"

[[4]]
[1] "maNUfactUREr" "mODEL"          "diSpl"        "YeaR"        "CYl"
[6] "trAnS"          "dRV"            "cTy"         "hwy"         "fL"
[11] "CIASS"

[[5]]
[1] "ManuFAcTUUrER" "ModEl"          "diSPl"        "YeAR"        "CyL"
[6] "TRans"          "DRv"            "CTy"         "HWY"         "fL"
[11] "CLass"
```

```

[[6]]
[1] "MANuFaCtURER" "MODEL"          "DIspl"        "yEAR"        "cYL"
[6] "tRanS"          "dRv"           "ctY"         "HWy"         "f1"
[11] "Class"

```

To make this data set even more messy, let's select a random set of columns per data frame:

```

inconsistent_dframes <- map(inconsistent_dframes,
  ~ .x[sample(1:length(.x), sample(1:length(.x), 1))])

map(inconsistent_dframes, ~ colnames(.x)) %>%
  head

[[1]]
[1] "MAnUFActuRER" "hWy"          "trANS"        "FL"          "Year"
[6] "dispL"          "Drv"          "cLAss"        "fL"          "CTY"

[[2]]
[1] "ModEL"          "trANS"        "DRV"         "fL"          "CTY"
[6] "mAnufACTuRER"  "DiSpL"

[[3]]
[1] "CTY"            "cyl"          "tRaNs"        "ManuFAcTUreR" "dRv"
[6] "YEar"           "mODEL"

[[4]]
[1] "mODEL"          "trAnS"        "YeaR"         "hwy"         "CYl"
[6] "dRV"            "CLASS"        "diSpl"        "fL"          "maNUfactUREr"
[11] "cTy"

[[5]]
[1] "CTy"            "CyL"          "fL"           "diSP1"        "TRans"
[6] "DRv"            "YeAR"         "CLass"        "ManuFAcTUrER" "HWY"

[[6]]
[1] "tRanS"          "MANuFaCtURER"  "MODEL"        "f1"          "cYL"
[6] "DIspl"          "HWy"

```

Finally, we save the data to disk:

```

dir_create(c("unclean_files"))

iwalk(inconsistent_dframes,
~ write_csv(.x, paste0("unclean_files/", .y, ".csv")))

```

If we tried to load this data using our previous approach, it would work, but the inconsistent column names would result in a plethora of columns:

```

many_columns_data_frame <- dir_ls(path = "unclean_files/",
                                    glob = "*.csv", type = "file") %>%
  map_dfr(~ read_csv(.x, show_col_types = FALSE) %>%
    mutate(filename = .x))

colnames(many_columns_data_frame) %>% sort

[1] "cLAss"          "ClASS"          "CLass"          "cTy"            "CTy"
[6] "CTY"            "cyl"             "cYL"            "CyL"            "CYl"
[11] "CYL"            "dispL"           "diSpl"          "diSpL"          "diSP1"
[16] "dISpl"          "DiSpL"           "DIspl"          "drV"            "dRv"
[21] "dRV"            "DrV"             "DRv"            "filename"       "f1"
[26] "fL"              "F1"               "FL"              "hwy"            "hWY"
[31] "hWY"            "HwY"             "HWy"            "HWY"            "manUfactuREr"
[36] "maNUfactUREr"   "mAnufACtuRER"   "mANufaCTUrER"  "ManuFAcTUrEr"   "ManuFAcTUrER"
[41] "ManUFActTuRER"  "MAnUFActuRER"   "MANuFaCtURER"  "moDEL"          "mODEL"
[46] "mODEL"          "ModEL"           "MODEL"          "trAnS"          "trANS"
[51] "tRanS"          "tRaNs"           "TRans"          "TTrans"         "Year"
[56] "YeaR"            "YeAR"            "YEar"           ""                ""

```

Clearly, that's not what we want. Instead, we can use our last trick and clean up the data frames and convert the column names to a specific naming convention. We write them all in lowercase and bind them together:

```

many_columns_data_frame <- dir_ls(path = "unclean_files/",
                                    glob = "*.csv", type = "file") %>%
  map_dfr(~ read_csv(.x, name_repair = tolower, show_col_types = FALSE) %>%
    mutate(filename = .x))

many_columns_data_frame %>% glimpse()

```

Rows: 200

```

Columns: 12
$ manufacturer <chr> "toyota", "nissan", "toyota", "toyota", "hyundai", "subar-
$ hwy          <dbl> 20, 31, 20, 26, 26, 26, 20, 28, 20, 19, 17, 25, 12, 20, 1-
$ trans         <chr> "auto(15)", "auto(av)", "auto(14)", "manual(m5)", "manual-
$ fl           <chr> "r", "r", "r", "r", "r", "r", "p", "r", "r", "r", "p-
$ year         <dbl> 2008, 2008, 1999, 1999, 1999, 2008, 2008, 2008, 2008, 200-
$ displ        <dbl> 4.0, 2.5, 2.7, 3.0, 2.5, 2.5, 5.3, 2.0, 5.3, 4.6, 5.0, 2.-
$ drv          <chr> "4", "f", "4", "f", "4", "r", "f", "r", "4", "4", "4-
$ class         <chr> "pickup", "midsize", "pickup", "midsize", "midsize", "suv-
$ filename      <fs::path> "unclean_files/1.csv", "unclean_files/1.csv", "uncle-
$ model         <chr> NA, N-
$ cty           <dbl> NA, N-
$ cyl            <dbl> NA, N-

```

As you can see, the new file has the same number of columns as the original mpg data frame. Only some values are `NAs` because we selected a random set of columns per file.

2.4 What if the files are not in the same folder?

So far, we have assumed that all files are in the same folder. This is of course not always the case. Sometimes your files are deeply nested. In that case, we need to search through each folder recursively. Recursively means that we search through each folder until we can't find another folder to crawl. Before we see how this works, let's store our data in two folders (this trick works with as many folders as you want):

```

mpg_samples <- map(1:40, ~ slice_sample(mpg, n = 20))

# Create directories
dir_create(c("nested_folders",
            "nested_folders/first_nested_folder",
            "nested_folders/second_nested_folder"))

# First folder
iwalk(mpg_samples[1:20],
      ~ write_csv(.x,
                  paste0("nested_folders/first_nested_folder/", .y, "_first.csv")))

# Second folder
iwalk(mpg_samples[21:40],
      ~ write_csv(.x,

```

```
paste0("nested_folders/second_nested_folder/", .y, "_second.csv"))
```

If you now try to load all csv files from the `nested_folders` folder, you would get an empty vector:

```
(csv_files_nested <- dir_ls("nested_folders/", glob = "*.csv", type = "file"))

character(0)
```

This is because `dir_ls` does not look in the nested folders, but only in the parent folder. To make `dir_ls` search through the folders recursively, you need to set the `recurse` argument to `TRUE`:

```
(csv_files_nested <- dir_ls("nested_folders/", glob = "*.csv", type = "file",
                             recurse = TRUE))
```

```
nested_folders/first_nested_folder/10_first.csv
nested_folders/first_nested_folder/11_first.csv
nested_folders/first_nested_folder/12_first.csv
nested_folders/first_nested_folder/13_first.csv
nested_folders/first_nested_folder/14_first.csv
nested_folders/first_nested_folder/15_first.csv
nested_folders/first_nested_folder/16_first.csv
nested_folders/first_nested_folder/17_first.csv
nested_folders/first_nested_folder/18_first.csv
nested_folders/first_nested_folder/19_first.csv
nested_folders/first_nested_folder/1_first.csv
nested_folders/first_nested_folder/20_first.csv
nested_folders/first_nested_folder/2_first.csv
nested_folders/first_nested_folder/3_first.csv
nested_folders/first_nested_folder/4_first.csv
nested_folders/first_nested_folder/5_first.csv
nested_folders/first_nested_folder/6_first.csv
nested_folders/first_nested_folder/7_first.csv
nested_folders/first_nested_folder/8_first.csv
nested_folders/first_nested_folder/9_first.csv
nested_folders/second_nested_folder/10_second.csv
nested_folders/second_nested_folder/11_second.csv
nested_folders/second_nested_folder/12_second.csv
nested_folders/second_nested_folder/13_second.csv
```

```
nested_folders/second_nested_folder/14_second.csv
nested_folders/second_nested_folder/15_second.csv
nested_folders/second_nested_folder/16_second.csv
nested_folders/second_nested_folder/17_second.csv
nested_folders/second_nested_folder/18_second.csv
nested_folders/second_nested_folder/19_second.csv
nested_folders/second_nested_folder/1_second.csv
nested_folders/second_nested_folder/20_second.csv
nested_folders/second_nested_folder/2_second.csv
nested_folders/second_nested_folder/3_second.csv
nested_folders/second_nested_folder/4_second.csv
nested_folders/second_nested_folder/5_second.csv
nested_folders/second_nested_folder/6_second.csv
nested_folders/second_nested_folder/7_second.csv
nested_folders/second_nested_folder/8_second.csv
nested_folders/second_nested_folder/9_second.csv
```

Now you can access all files inside the nested_folders directory:

```
map_dfr(csv_files_nested, ~ read_csv(.x, show_col_types = FALSE) %>%
    mutate(filename = .x)) %>%
glimpse()
```

```
Rows: 800
Columns: 12
$ manufacturer <chr> "subaru", "subaru", "ford", "ford", "pontiac", "volkswage~
$ model         <chr> "impreza awd", "forester awd", "explorer 4wd", "f150 pick~
$ displ          <dbl> 2.5, 2.5, 4.0, 5.4, 3.8, 1.8, 4.0, 2.4, 6.5, 5.4, 2.2, 6.~
$ year           <dbl> 2008, 2008, 2008, 2008, 2008, 1999, 1999, 1999, 1999, 200~
$ cyl            <dbl> 4, 4, 6, 8, 6, 4, 6, 4, 8, 8, 4, 8, 6, 4, 6, 6, 4, 6, ~
$ trans          <chr> "auto(s4)", "manual(m5)", "auto(15)", "auto(14)", "auto(1~
$ drv             <chr> "4", "4", "4", "f", "f", "4", "f", "4", "r", "f", "r~
$ cty            <dbl> 20, 19, 13, 13, 18, 21, 15, 19, 14, 12, 21, 15, 17, 18, 1~
$ hwy            <dbl> 25, 25, 19, 17, 28, 29, 20, 27, 17, 18, 29, 25, 24, 27, 2~
$ fl              <chr> "p", "p", "r", "r", "p", "r", "r", "d", "r", "r", "p~
$ class          <chr> "compact", "suv", "suv", "pickup", "midsize", "midsize", ~
$ filename        <fs::path> "nested_folders/first_nested_folder/10_first.csv", "~
```

2.5 What if I don't need some of these files?

You don't always need all the files in your directory and need to remove some files from the list of file paths. A good way to do this is to use the `str_detect` function from the `stringr` package. Let's look at an example. In the following example, I created a character vector and kept the files that contain the string beach:

```
str_detect(c("house", "beach"), pattern = "beach")
```

```
[1] FALSE TRUE
```

The function returns logical values. To change the actual character vector, we need to add these logical values to the character vector itself:

```
c("my house", "my beach")[str_detect(c("house", "beach"), pattern = "beach")]
```

```
[1] "my beach"
```

But what if you want to remove these files? With the `negate` argument you can find only the files that do not match the pattern:

```
c("my house", "my beach")[str_detect(c("house", "beach"), pattern = "beach",  
negate = TRUE)]
```

```
[1] "my house"
```

The hard part is finding the right pattern for your files. Suppose you don't want to keep CSV files that contain the numbers 2, 3 or 4:

```
csv_files_nested[str_detect(csv_files_nested, pattern = "[2-4]",  
negate = TRUE)]
```

```
nested_folders/first_nested_folder/10_first.csv  
nested_folders/first_nested_folder/11_first.csv  
nested_folders/first_nested_folder/15_first.csv  
nested_folders/first_nested_folder/16_first.csv  
nested_folders/first_nested_folder/17_first.csv  
nested_folders/first_nested_folder/18_first.csv
```

```
nested_folders/first_nested_folder/19_first.csv  
nested_folders/first_nested_folder/1_first.csv  
nested_folders/first_nested_folder/5_first.csv  
nested_folders/first_nested_folder/6_first.csv  
nested_folders/first_nested_folder/7_first.csv  
nested_folders/first_nested_folder/8_first.csv  
nested_folders/first_nested_folder/9_first.csv  
nested_folders/second_nested_folder/10_second.csv  
nested_folders/second_nested_folder/11_second.csv  
nested_folders/second_nested_folder/15_second.csv  
nested_folders/second_nested_folder/16_second.csv  
nested_folders/second_nested_folder/17_second.csv  
nested_folders/second_nested_folder/18_second.csv  
nested_folders/second_nested_folder/19_second.csv  
nested_folders/second_nested_folder/1_second.csv  
nested_folders/second_nested_folder/5_second.csv  
nested_folders/second_nested_folder/6_second.csv  
nested_folders/second_nested_folder/7_second.csv  
nested_folders/second_nested_folder/8_second.csv  
nested_folders/second_nested_folder/9_second.csv
```

The regular expression [2-4] looks for the numbers 2 to 4. But what if you want to exclude files that end with a 2, 3, or 4? Then, of course, this regular expression won't work. In this case we need another pattern:

```
csv_files_nested[str_detect(csv_files_nested,  
                           pattern = "[2-4]_first|second\\.csv$",
                           negate = TRUE)]
```

```
nested_folders/first_nested_folder/10_first.csv  
nested_folders/first_nested_folder/11_first.csv  
nested_folders/first_nested_folder/15_first.csv  
nested_folders/first_nested_folder/16_first.csv  
nested_folders/first_nested_folder/17_first.csv  
nested_folders/first_nested_folder/18_first.csv  
nested_folders/first_nested_folder/19_first.csv  
nested_folders/first_nested_folder/1_first.csv  
nested_folders/first_nested_folder/20_first.csv  
nested_folders/first_nested_folder/5_first.csv  
nested_folders/first_nested_folder/6_first.csv  
nested_folders/first_nested_folder/7_first.csv  
nested_folders/first_nested_folder/8_first.csv
```

```
nested_folders/first_nested_folder/9_first.csv
```

This pattern is a bit more complicated. Again, we look for the numbers 2 to 4 followed by an underscore and the words first or second (indicated by a vertical bar |). Since a period represents any arbitrary character in regular expressions, we need to terminate it with two backslashes. Finally, our file should end with the characters csv, which is indicated by the dollar sign \$.

The rest is similar to what we did before:

```
csv_files_nested[str_detect(csv_files_nested,
                             pattern = "[2-4]_first|second\\.csv$",
                             negate = TRUE)] %>%
  map_dfr(~ read_csv(.x, show_col_types = FALSE) %>%
    mutate(filename = .x)) %>%
  glimpse()
```

```
Rows: 280
Columns: 12
$ manufacturer <chr> "subaru", "subaru", "ford", "ford", "pontiac", "volkswage-
$ model          <chr> "impreza awd", "forester awd", "explorer 4wd", "f150 pick-
$ displ           <dbl> 2.5, 2.5, 4.0, 5.4, 3.8, 1.8, 4.0, 2.4, 6.5, 5.4, 2.2, 6.-
$ year            <dbl> 2008, 2008, 2008, 2008, 1999, 1999, 1999, 1999, 200-
$ cyl              <dbl> 4, 4, 6, 8, 6, 4, 6, 8, 8, 4, 8, 6, 4, 6, 6, 4, 6, ~
$ trans             <chr> "auto(s4)", "manual(m5)", "auto(15)", "auto(14)", "auto(l-
$ drv                <chr> "4", "4", "4", "f", "f", "4", "f", "4", "r", "f", "r-
$ cty               <dbl> 20, 19, 13, 13, 18, 21, 15, 19, 14, 12, 21, 15, 17, 18, 1-
$ hwy               <dbl> 25, 25, 19, 17, 28, 29, 20, 27, 17, 18, 29, 25, 24, 27, 2-
$ fl                 <chr> "p", "p", "r", "r", "p", "r", "r", "d", "r", "r", "p-
$ class              <chr> "compact", "suv", "suv", "pickup", "midsized", "midsized", ~
$ filename           <fs::path> "nested_folders/first_nested_folder/10_first.csv", "~
```

i Summary

Here is what you can take from this tutorial.

- To read many files into R, you need to create a character vector of file paths. Once you have this vector you can read the files with `map_dfr` or the `read_*` functions.
- If you only need a subset of data frames, you can filter the character vector of the file paths with regular expressions
- If your files are not in the same folder search them recursively
- If the column names of your files are not consistent, use the `name_repair` argument

of your `read_*` functions

Part II

Improve working with columns

3 How to select variables with the tidyselect functions

What will this tutorial cover?

In this tutorial you will get an introduction to the tidyselect functions. These functions give you many options to select columns in R.

Who do I have to thank?

For this tutorial I referred to the [official dplyr documentation of tidyselect](#). A big thanks to [Lionel Henry](#), who was instrumental in implementing these functions.

Selecting columns sounds fair and easy. But could you solve the following problems spontaneously in R?

- Select all columns of a data frame that are numeric
- Select all numeric columns that do not contain the numbers 1 or 2
- Select all columns that start with the string “wk”
- Select all columns that contain a number

If not, this tutorial might be of interest to you. We will cover a range of functions from the tidyselect package. The tidyselect package was designed specifically for these use cases and makes solving these problems much easier.

The following tricks are actually not too complicated. But we need them for the upcoming tutorials. So in this tutorial, we will go through these functions in detail:

- `everything()`
- `last_col()`
- `starts_with()`
- `ends_with()`
- `contains()`
- `matches()`
- `num_range()`
- `where()`

A common feature of all functions is that they allow you to select columns. Later in this course you will use them not only inside `select`, but also inside `summarise`, or `mutate`. Remember that any column selection can be negated with an exclamation mark `!`. I'll sprinkle in a few examples of negated column selections, just remember that negation can be applied to any of these functions. The most complicated function will be `matches` because it works with regular expressions. If you are familiar with regular expressions, you should have no problems with the examples. If not, you can read more about regular expressions [on the official stringr website](#). I'll try to explain the examples as good as I can.

3.1 Selecting every column

The first function is `everything`. As the name suggests, it lets you select all columns of a data frame:

```
mpg %>%
  select(everything()) %>%
  glimpse()
```

```
Rows: 234
Columns: 11
$ manufacturer <chr> "audi", "audi", "audi", "audi", "audi", "audi", "~
$ model          <chr> "a4", "a4", "a4", "a4", "a4", "a4", "a4 quattro", "~
$ displ           <dbl> 1.8, 1.8, 2.0, 2.0, 2.8, 2.8, 3.1, 1.8, 1.8, 2.0, 2.0, 2.~
$ year            <int> 1999, 1999, 2008, 2008, 1999, 1999, 2008, 1999, 1999, 200~
$ cyl              <int> 4, 4, 4, 4, 6, 6, 6, 4, 4, 4, 6, 6, 6, 6, 6, 8, 8, ~
$ trans            <chr> "auto(15)", "manual(m5)", "manual(m6)", "auto(av)", "auto~
$ drv               <chr> "f", "f", "f", "f", "f", "f", "4", "4", "4", "4", "4", "4~
$ cty              <int> 18, 21, 20, 21, 16, 18, 18, 16, 20, 19, 15, 17, 17, 1~
$ hwy              <int> 29, 29, 31, 30, 26, 26, 27, 26, 25, 28, 27, 25, 25, 25, 2~
$ fl                <chr> "p", "p~
$ class             <chr> "compact", "compact", "compact", "compact", "compact", "c~
```

You will rightly ask yourself why you need such a function. One use case is relocating columns with `everything`. For example, we could move the column `cyl` and `manufacturer` to the beginning of the data frame:

```
mpg %>%
  select(manufacturer, cyl, everything()) %>%
  glimpse()
```

```

Rows: 234
Columns: 11
$ manufacturer <chr> "audi", "audi", "audi", "audi", "audi", "audi", "~"
$ cyl           <int> 4, 4, 4, 4, 6, 6, 6, 4, 4, 4, 4, 6, 6, 6, 6, 6, 8, 8, ~
$ model         <chr> "a4", "a4", "a4", "a4", "a4", "a4", "a4", "a4", "a4 quattro", "~"
$ displ          <dbl> 1.8, 1.8, 2.0, 2.0, 2.8, 2.8, 3.1, 1.8, 1.8, 2.0, 2.0, 2.~ 
$ year          <int> 1999, 1999, 2008, 2008, 1999, 1999, 2008, 1999, 1999, 200~ 
$ trans          <chr> "auto(15)", "manual(m5)", "manual(m6)", "auto(av)", "auto~
$ drv            <chr> "f", "f", "f", "f", "f", "f", "4", "4", "4", "4", "4", "4~
$ cty            <int> 18, 21, 20, 21, 16, 18, 18, 18, 16, 20, 19, 15, 17, 17, 1~
$ hwy            <int> 29, 29, 31, 30, 26, 26, 27, 26, 25, 28, 27, 25, 25, 25, 2~
$ fl              <chr> "p", "p~
$ class          <chr> "compact", "compact", "compact", "compact", "compact", "compact", "c~

```

Other than that, I haven't seen any other use cases for `everything`.

3.2 Selecting the last column

Then we have `last_col`. With this function you can select the last column in a data frame:

```

mpg %>%
  select(last_col()) %>%
  glimpse()

```

```

Rows: 234
Columns: 1
$ class <chr> "compact", "compact", "compact", "compact", "compact", "compact"~

```

I have already told you that you can also negate a selection of columns. So let's then try to select all columns except the last one:

```

mpg %>%
  select(!last_col()) %>%
  glimpse()

```

```

Rows: 234
Columns: 10
$ manufacturer <chr> "audi", "audi", "audi", "audi", "audi", "audi", "audi", "~"
$ model         <chr> "a4", "a4", "a4", "a4", "a4", "a4", "a4", "a4 quattro", "~"

```

```
$ displ      <dbl> 1.8, 1.8, 2.0, 2.0, 2.8, 2.8, 3.1, 1.8, 1.8, 2.0, 2.0, 2.~  
$ year       <int> 1999, 1999, 2008, 2008, 1999, 1999, 2008, 1999, 1999, 200~  
$ cyl        <int> 4, 4, 4, 4, 6, 6, 4, 4, 4, 6, 6, 6, 6, 6, 8, 8, ~  
$ trans      <chr> "auto(15)", "manual(m5)", "manual(m6)", "auto(av)", "auto~  
$ drv         <chr> "f", "f", "f", "f", "f", "f", "4", "4", "4", "4", "4", "4~  
$ cty        <int> 18, 21, 20, 21, 16, 18, 18, 16, 20, 19, 15, 17, 17, 1~  
$ hwy        <int> 29, 29, 31, 30, 26, 26, 27, 26, 25, 28, 27, 25, 25, 2~  
$ fl          <chr> "p", "p~
```

Also, you can use `last_col` to select the n-to-last column:

```
mpg %>%  
  select(last_col(1)) %>%  
  glimpse
```

```
Rows: 234  
Columns: 1  
$ fl <chr> "p", "p~
```

Note that the index starts with 0, so the number 1 indicates the second to last column.

3.3 Selecting columns that start and end with a specific string

Next we have the two functions `starts_with` and `ends_with`. You use these functions when you want to select columns that start or end with exactly a certain string. We could use `starts_with` to select all columns that start with the letter “m”:

```
mpg %>%  
  select(starts_with("m")) %>%  
  glimpse()
```

```
Rows: 234  
Columns: 2  
$ manufacturer <chr> "audi", "audi", "audi", "audi", "audi", "audi", "a~  
$ model           <chr> "a4", "a4", "a4", "a4", "a4", "a4", "a4 quattro", "a~
```

`starts_with` and `ends_with` works with any character, but also with a vector of characters. Suppose we want to select columns ending with the letter l or r:

```

mpg %>%
  select(ends_with(c("l", "r"))) %>%
  glimpse()

Rows: 234
Columns: 6
$ model      <chr> "a4", "a4", "a4", "a4", "a4", "a4", "a4 quattro", "~
$ displ       <dbl> 1.8, 1.8, 2.0, 2.0, 2.8, 2.8, 3.1, 1.8, 1.8, 2.0, 2.0, 2.~
$ cyl        <int> 4, 4, 4, 4, 6, 6, 6, 4, 4, 4, 4, 6, 6, 6, 6, 6, 8, 8, ~
$ fl         <chr> "p", "p~
$ manufacturer <chr> "audi", "audi", "audi", "audi", "audi", "audi", "audi", "~
$ year       <int> 1999, 1999, 2008, 2008, 1999, 1999, 2008, 1999, 1999, 200~

```

Of course, the character string can be longer than just one character:

```

billboard %>%
  select(starts_with("wk")) %>%
  colnames()

[1] "wk1"   "wk2"   "wk3"   "wk4"   "wk5"   "wk6"   "wk7"   "wk8"   "wk9"   "wk10"
[11] "wk11"  "wk12"  "wk13"  "wk14"  "wk15"  "wk16"  "wk17"  "wk18"  "wk19"  "wk20"
[21] "wk21"  "wk22"  "wk23"  "wk24"  "wk25"  "wk26"  "wk27"  "wk28"  "wk29"  "wk30"
[31] "wk31"  "wk32"  "wk33"  "wk34"  "wk35"  "wk36"  "wk37"  "wk38"  "wk39"  "wk40"
[41] "wk41"  "wk42"  "wk43"  "wk44"  "wk45"  "wk46"  "wk47"  "wk48"  "wk49"  "wk50"
[51] "wk51"  "wk52"  "wk53"  "wk54"  "wk55"  "wk56"  "wk57"  "wk58"  "wk59"  "wk60"
[61] "wk61"  "wk62"  "wk63"  "wk64"  "wk65"  "wk66"  "wk67"  "wk68"  "wk69"  "wk70"
[71] "wk71"  "wk72"  "wk73"  "wk74"  "wk75"  "wk76"

```

3.4 Selecting columns that contain certain strings

Next we have the `contains` function. `contains` searches for columns that contain a specific string. Note that it does not work with regular expressions, but searches for exactly the string you specify. By default, however, the function is not case-sensitive. It doesn't matter if your columns are in uppercase or lowercase. Let's select all columns that contain the letter "m":

```

mpg %>%
  select(contains("m")) %>%
  glimpse()

```

```

Rows: 234
Columns: 2
$ manufacturer <chr> "audi", "audi", "audi", "audi", "audi", "audi", "~"
$ model           <chr> "a4", "a4", "a4", "a4", "a4", "a4", "a4", "a4 quattro", "~"

```

I told you that `contains` is not case sensitive. If you are concerned about case sensitivity, set the `ignore.case` argument to `FALSE` (this also works with `starts_with`, `ends_with`, and `matches`):

```

mpg %>%
  rename(Manufacturer = manufacturer) %>%
  select(contains("m", ignore.case = FALSE)) %>%
  glimpse()

```

```

Rows: 234
Columns: 1
$ model <chr> "a4", "a4", "a4", "a4", "a4", "a4", "a4", "a4 quattro", "a4 quat~

```

3.5 Selecting columns with regular expressions

Unlike `contains`, `matches` works with regular expressions. As mentioned at the beginning, if you don't know too much about regular expressions, don't worry. I'll do my best to explain the code as good as I can.

Suppose you want to select all columns that contain a number. The `billboard` dataset is a good use case for this. The dataset contains the rankings of songs over a period of 76 weeks.

```

billboard %>%
  select(matches("\\d")) %>%
  colnames()

[1] "wk1"   "wk2"   "wk3"   "wk4"   "wk5"   "wk6"   "wk7"   "wk8"   "wk9"   "wk10"
[11] "wk11"  "wk12"  "wk13"  "wk14"  "wk15"  "wk16"  "wk17"  "wk18"  "wk19"  "wk20"
[21] "wk21"  "wk22"  "wk23"  "wk24"  "wk25"  "wk26"  "wk27"  "wk28"  "wk29"  "wk30"
[31] "wk31"  "wk32"  "wk33"  "wk34"  "wk35"  "wk36"  "wk37"  "wk38"  "wk39"  "wk40"
[41] "wk41"  "wk42"  "wk43"  "wk44"  "wk45"  "wk46"  "wk47"  "wk48"  "wk49"  "wk50"
[51] "wk51"  "wk52"  "wk53"  "wk54"  "wk55"  "wk56"  "wk57"  "wk58"  "wk59"  "wk60"
[61] "wk61"  "wk62"  "wk63"  "wk64"  "wk65"  "wk66"  "wk67"  "wk68"  "wk69"  "wk70"
[71] "wk71"  "wk72"  "wk73"  "wk74"  "wk75"  "wk76"

```

The regular expression `\d` in the function stands for any digit. Similarly, one could search only for columns that begin with the string “wk” and are followed by only one digit:

```
billboard %>%
  select(matches("wk\\d{1}")) %>%
  colnames()

[1] "wk1" "wk2" "wk3" "wk4" "wk5" "wk6" "wk7" "wk8" "wk9"
```

Here, the curly braces indicate that we are looking for only one digit, and the dollar sign \$ indicates that the column should end with that one digit. From this example it should be obvious that `matches` provides the most versatile way to select columns among the `tidyselect` functions.

Let’s try another example. Imagine you want to select columns starting with the letter x or y and then followed by the digits 1 to 2:

```
anscombe %>%
  select(matches("[xy][1-2]")) %>%
  glimpse()

Rows: 11
Columns: 4
$ x1 <dbl> 10, 8, 13, 9, 11, 14, 6, 4, 12, 7, 5
$ x2 <dbl> 10, 8, 13, 9, 11, 14, 6, 4, 12, 7, 5
$ y1 <dbl> 8.04, 6.95, 7.58, 8.81, 8.33, 9.96, 7.24, 4.26, 10.84, 4.82, 5.68
$ y2 <dbl> 9.14, 8.14, 8.74, 8.77, 9.26, 8.10, 6.13, 3.10, 9.13, 7.26, 4.74
```

The square brackets are used to search for sets of characters. The first bracket indicates that we are looking for the letter x or y. If we put a hyphen - between the characters, we are looking for a range of values. Here we are looking for numbers between 1 and 2.

3.6 Select columns with number ranges

Next, we have `num_range`. The function is useful if your column names follow a certain pattern. In the `anscombe` data frame the column names start with a letter and are then followed by a number. Let’s try use `num_range` with this data frame. And let’s try to find all columns that start with the letter x and are followed by the numbers 1 or 2:

```
anscombe %>%
  select(num_range("x", 1:2)) %>%
  glimpse()
```

```
Rows: 11
Columns: 2
$ x1 <dbl> 10, 8, 13, 9, 11, 14, 6, 4, 12, 7, 5
$ x2 <dbl> 10, 8, 13, 9, 11, 14, 6, 4, 12, 7, 5
```

The same idea can be applied to the billboard dataset. In this dataset we have a list of columns starting with the letters “wk” (for week) and the numbers 1 to 76:

```
billboard %>%
  select(num_range("wk", 1:15)) %>%
  glimpse()
```

```
Rows: 317
Columns: 15
$ wk1 <dbl> 87, 91, 81, 76, 57, 51, 97, 84, 59, 76, 84, 57, 50, 71, 79, 80, 9~
$ wk2 <dbl> 82, 87, 70, 76, 34, 39, 97, 62, 53, 76, 84, 47, 39, 51, 65, 78, 9~
$ wk3 <dbl> 72, 92, 68, 72, 25, 34, 96, 51, 38, 74, 75, 45, 30, 28, 53, 76, 9~
$ wk4 <dbl> 77, NA, 67, 69, 17, 26, 95, 41, 28, 69, 73, 29, 28, 18, 48, 77, 9~
$ wk5 <dbl> 87, NA, 66, 67, 17, 26, 100, 38, 21, 68, 73, 23, 21, 13, 45, 92, ~
$ wk6 <dbl> 94, NA, 57, 65, 31, 19, NA, 35, 18, 67, 69, 18, 19, 13, 36, NA, 9~
$ wk7 <dbl> 99, NA, 54, 55, 36, 2, NA, 35, 16, 61, 68, 11, 20, 11, 34, NA, 93~
$ wk8 <dbl> NA, NA, 53, 59, 49, 2, NA, 38, 14, 58, 65, 9, 17, 1, 29, NA, 96, ~
$ wk9 <dbl> NA, NA, 51, 62, 53, 3, NA, 38, 12, 57, 73, 9, 17, 1, 27, NA, NA, ~
$ wk10 <dbl> NA, NA, 51, 61, 57, 6, NA, 36, 10, 59, 83, 11, 17, 2, 30, NA, NA, ~
$ wk11 <dbl> NA, NA, 51, 61, 64, 7, NA, 37, 9, 66, 92, 1, 17, 2, 36, NA, 99, N~
$ wk12 <dbl> NA, NA, 51, 59, 70, 22, NA, 37, 8, 68, NA, 1, 3, 3, 37, NA, NA, 9~
$ wk13 <dbl> NA, NA, 47, 61, 75, 29, NA, 38, 6, 61, NA, 1, 3, 3, 39, NA, 96, N~
$ wk14 <dbl> NA, NA, 44, 66, 76, 36, NA, 49, 1, 67, NA, 1, 7, 4, 49, NA, 96, N~
$ wk15 <dbl> NA, NA, 38, 72, 78, 47, NA, 61, 2, 59, NA, 4, 10, 12, 57, NA, 99, ~
```

3.7 Selecting columns of a specific type

Finally, there is the `where` function. `where` is used when you want to select variables of a certain data type. For example, we could select character variables:

```

billboard %>%
  select(where(is.character)) %>%
  glimpse()

Rows: 317
Columns: 2
$ artist <chr> "2 Pac", "2Ge+her", "3 Doors Down", "3 Doors Down", "504 Boyz",~
$ track  <chr> "Baby Don't Cry (Keep...)", "The Hardest Part Of ...", "Kryptoni~
```

`where` was introduced in tidyselect in 2020 to avoid confusing error messages. So make sure that if you use a predicate function (e.g. `is.character`), you include it in `where`. Other predicate functions are:

- `is.double`
- `is.logical`
- `is.factor`
- `is.integer`

3.8 Combining selections

Before we conclude this tutorial, you should know that you can combine the different selection functions with the `&` and `|` operators. Suppose we want to select all columns that are of type character and that contain the letter l:

```

mpg %>%
  select(where(is.character) & contains("l")) %>%
  glimpse()
```

```

Rows: 234
Columns: 3
$ model <chr> "a4", "a4", "a4", "a4", "a4", "a4", "a4", "a4 quattro", "a4 quat~
$ fl    <chr> "p", ~
$ class <chr> "compact", "compact", "compact", "compact", "compact", "compact"~
```

Similarly, we can use the or operator `|` to select columns that satisfy one of several conditions:

```

mpg %>%
  select(where(is.character) | contains("l")) %>%
  glimpse()
```

```

Rows: 234
Columns: 8
$ manufacturer <chr> "audi", "audi", "audi", "audi", "audi", "audi", "~
$ model          <chr> "a4", "a4", "a4", "a4", "a4", "a4", "a4 quattro", "~
$ trans          <chr> "auto(15)", "manual(m5)", "manual(m6)", "auto(av)", "auto-
$ drv            <chr> "f", "f", "f", "f", "f", "f", "4", "4", "4", "4", "4~
$ fl             <chr> "p", "p~
$ class          <chr> "compact", "compact", "compact", "compact", "compact", "c~
$ displ          <dbl> 1.8, 1.8, 2.0, 2.0, 2.8, 2.8, 3.1, 1.8, 1.8, 2.0, 2.0, 2.~
$ cyl            <int> 4, 4, 4, 4, 6, 6, 6, 4, 4, 4, 4, 6, 6, 6, 6, 6, 8, 8, ~

```

Compared to our previous example, the selection results in 8 columns instead of 3 because we used the or operator.

This concludes our brief tour of the tidyselect functions. These functions are not the most exciting in this course, but they are fundamental to many things that we will cover later in this course.

Summary

Here is what you can take from this tutorial.

- The tidyselect functions provide you with versatile possibilities to select variables
- Tidyselect provides the following functions: `everything`, `last_col`, `starts_with`, `ends_with`, `contains`, `matches`, `num_range`, `where`
- All functions can be used with other functions as `select`. For example, `summarise` or `mutate`
- Any selection can be negated within `select`
- `everything` can be used to relocate variables
- `matches` is the most complicated function because it works with regular expressions

4 How to rename many column names at once

What will this tutorial cover?

In this tutorial you will learn how to rename many columns with the `rename_with` function. With `rename_with` we can apply functions to specific column names which results in more elegant and error-free code than `rename`.

Who do I have to thank?

For this trick, I referred on [the official documentation of dplyr](#).

Column names often contain spaces, special characters, or are written in a mixture of lower and upper case characters. Such poorly formatted column names can lead to numerous problems. Let's take these column names:

```
Age, Item 1, Item 2, $dollars, Where do you come from?  
23 , 4      , 8      , 45      , "Germany"
```

- The column names `Item 1` and `Item 2` contain spaces. If we had to refer to these columns in R, we would have to enclose them in single quotes: `Item 1`. This is much more complicated to write than simply `item_1`. The same thing happens with the last column. Here the column name is not just one word, but a whole sentence. This is not uncommon, especially when a CSV file is generated directly from a survey program or an Excel file.
- The column `$dollars` starts with a \$ sign. Column names, however, cannot start with a special character in R. We would have to remove the dollar sign from the column name.
- The first letter of some column names are written in upper case. This mustn't be a problem, but I find that it is generally easier to write column names in lower case.
- We also see a pattern. The column names `Item 1` and `Item 2` differ only by their number. So it would be nice if we could rename these column names at once.

We could easily solve these issues with the ‘`rename`’ function but this approach does not scale:

```
the_data_frame %>%
  rename(
    age      = Age,
    item_1   = `Item 1`,
    item_2   = `Item 2`,
    dollars  = `$dollars`,
    origin   = `Where do you come from?`
  )
```

Imagine how long it would take if you had 50 column names to change using the `rename` function. So obviously we need a better solution. And the solution is the `rename_with` function.

4.1 The `rename_with` function

The main difference between `rename` and `rename_with` is that `rename_with` changes the column names using a function. The three main arguments of the function are `.data`, `.fn` and `.cols`. `.data` stands for the data frame, `.fn` for the function to apply to the column names, and `.cols` for the columns to apply the function to.

```
mpg %>%
  rename_with(data = .,
              .fn = toupper,
              .cols = everything())
```

In this example I used the function `toupper` to convert all column names to uppercase. Here's a little trick to better understand what is happening here: Just imagine you want to replace or change strings in a vector of strings:

```
c("manufacturer", "model", "displ", "year") %>%
  toupper
```

[1]	"MANUFACTURER"	"MODEL"	"DISPL"	"YEAR"
-----	----------------	---------	---------	--------

Essentially, `rename_with` does nothing but use the column names as a vector of strings:

```
mpg %>%
  rename_with(.fn = toupper,
              .cols = everything()) %>%
  colnames()
```

```
[1] "MANUFACTURER" "MODEL"          "DISPL"           "YEAR"           "CYL"  
[6] "TRANS"          "DRV"            "CTY"             "HWY"            "FL"  
[11] "CLASS"
```

Let's look at another example. In our first tutorial, we learned how to use a specific naming convention for column names using the `make_clean_names` function from the `janitor` package. Since this is a function that works with a vector of strings, we can use this function to write all column names of the `iris` data frame in BigCamel notation:

```
# Make sure to install janitor first:  
#   install.packages("janitor")  
iris %>%  
  rename_with(~ janitor::make_clean_names(., case = "big_camel")) %>%  
  colnames()
```

```
[1] "SepalLength" "SepalWidth"  "PetalLength" "PetalWidth"  "Species"
```

From this example, we can see a few things:

- I did not explicitly mention the `.data` and `.cols` arguments. Since I use the pipe, I don't need to mention the `.data` argument explicitly. Also, if I don't mention the `.cols` argument explicitly, the function is applied to all column names.
- In this example, I used the tilde operator to indicate an anonymous function. This shortcut is needed whenever you need to call certain arguments of a function. In our case, this is the `case` argument of `make_clean_names`.

In summary, we can use the `toupper`, `tolower`, and `make_clean_names` functions to convert the column names to lowercase or uppercase, or to convert our column names to a particular naming convention.

4.2 How to use `rename_with` to replace characters

Another use case of `rename_with` is the replacement of characters. Suppose we want to replace all `e` characters with an underscore `_`:

```
mpg %>%  
  rename_with(~ gsub("e", "_", .)) %>%  
  colnames()
```

```
[1] "manufactur_r" "mod_l"          "displ"        "y_ar"        "cyl"  
[6] "trans"         "drv"           "cty"          "hwy"         "fl"  
[11] "class"
```

As you can see, I used the `gsub` function to replace a specific character. Alternatively, I could have used the `str_replace` function:

```
mpg %>%  
  rename_with(~ str_replace(., "e", "_")) %>%  
  colnames()
```

```
[1] "manufactur_r" "mod_l"          "displ"        "y_ar"        "cyl"  
[6] "trans"         "drv"           "cty"          "hwy"         "fl"  
[11] "class"
```

Let's look at a slightly more complicated example. The column names of the anscombe dataset consist of the letter x or y and the numbers 1 to 4:

```
anscombe %>%  
  colnames()
```

```
[1] "x1" "x2" "x3" "x4" "y1" "y2" "y3" "y4"
```

Suppose we want to insert an underscore between the letter and the number. A trick to solve this problem is to use the grouping function in `str_replace`. A group in the argument `pattern` is everything between two brackets.

```
anscombe %>%  
  rename_with(~ str_replace(., pattern = "(\\d+)",  
                           replacement = "_\\1")) %>%  
  colnames()
```

```
[1] "x_1" "x_2" "x_3" "x_4" "y_1" "y_2" "y_3" "y_4"
```

Let's unpack this code. With `pattern` we said that we are looking for a group of characters containing one or more digits (`\\d+`). `\\d+` is a regular expression. If you want to learn more about it, take a look at [the official documentation of stringr](#). With `replacement` we said that we want to put an underscore in front of this group. The group itself is specified by `\\1`. If we had two groups, the second group would be specified by `\\2`.

As you can see in the output, we have added an underscore between the letter and the number. Let's create a more complicated example with two groups. Suppose we want to replace only the column names `y1` and `y2` with `ypsiilon1_` and `ypsiilon2_`. If we only use the techniques we have learned so far, we need to create two groups. The first group is used to replace the `y` with `ypsiilon`. The second group is used to replace the number with the number and an underscore:

```
anscombe %>%
  rename_with(~ str_replace(., "(y)([1-2])",
                           "\\\1psilon\\\2_")) %>%
  colnames()

[1] "x1"          "x2"          "x3"          "x4"          "ypsiilon1_" "ypsiilon2_"
[7] "y3"          "y4"
```

As you can see, we renamed only two of the eight columns. This is because the pattern used in `str_replace` only applies to two variables: The variables that start with a `y` and are followed by the numbers 1 or 2. You can also see that we used two groups. The first group contains the letter `y`, the second group contains the number 1 or 2 (indicated by the square brackets `[1-2]`). In the replacement argument you will find these groups as `\\\1` and `\\\2`. If we would get rid of these groups in the replacement argument, they would be removed from the new column names (and it would throw an error because the column names would not be unique anymore).

4.3 How to rename variables for specific variables

Granted, shows the power of `rename_with`, but it is not the most elegant. So far we haven't talked about the third argument of `rename_with`: `.cols`. You can use `.cols` to specify which column names to apply the function to. And you can even use our tidyselect functions for that. So let's rewrite our previous example with the `.cols` argument:

```
anscombe %>%
  rename_with(~ str_replace(., "([:alpha:])([1-2])",
                           "\\\1psilon\\\2_"), c(y1, y2)) %>%
  colnames()

[1] "x1"          "x2"          "x3"          "x4"          "ypsiilon1_" "ypsiilon2_"
[7] "y3"          "y4"
```

Take a closer look at the end of the code. Here we have specified that the `str_replace` function should only be applied to the columns names `y1` and `y2`. Again, we have defined two groups: `([:alpha:]):` and `([1-2]):`. The first group searches for a single letter. The second group searches for the numbers 1 or 2. You can see that the output is the same, we just used a different method.

Similarly, you could use a `tidyselect` function and convert all numeric columns to uppercase:

```
mpg %>%
  rename_with(~ toupper(.), where(is.numeric)) %>%
  colnames()

[1] "manufacturer" "model"          "DISPL"        "YEAR"         "CYL"
[6] "trans"          "drv"            "CTY"          "Hwy"          "f1"
[11] "class"
```

Or you could replace a dot with an underscore for all columns that begin with the word Sepal:

```
iris %>%
  rename_with(~ str_replace(., "\\.", "_"),
             starts_with("Sepal")) %>%
  colnames()

[1] "Sepal_Length" "Sepal_Width"   "Petal.Length" "Petal.Width"  "Species"
```

Another useful function is `matches`. With `matches`, you can search for specific patterns in your column names and apply a function to the column names that match the pattern. In the next example, we used this technique to replace the dot with an underscore in all column names that end with “Width” or “width”:

```
iris %>%
  rename_with(~ str_replace(., "\\.", "_"),
             matches("[Ww]idth$")) %>%
  colnames()

[1] "Sepal.Length" "Sepal_Width"   "Petal.Length" "Petal_Width"  "Species"
```

Summary

Here is what you can take from this tutorial.

- `rename_with` allows us to change many column names using functions.
- `rename_with` has three important arguments: `.data` which stands for the data frame, `.fn` for the function to apply to the selected column names, and `.cols` which stands for the column names to apply the function to.
- The most common use cases for `rename_with` are converting column names to a specific naming convention, converting column names to lowercase or uppercase, or removing and replacing certain characters in the column names
- More complex character replacements can be achieved with the grouping function of `str_replace` in combination with the `.cols` argument of `rename_with`.

Part III

Improve creating and modifying variables

5 How to count with count, add_count, and add_tally

What will this tutorial cover?

In this tutorial you will learn how to count values with `count()`, `add_count()`, and `add_tally()`. Also, you will find out how to count with continuous variables.

Who do I have to thank?

For this post, I have to thank [David Robinson](#), from whom I learned about the decade trick. Thanks also go to Olivier Gimenez who wrote a [really nice article explaining this trick](#).

Counting is one of the most common tasks you do when working with data. Counting may sound simple, but it can get complicated quickly. Consider these examples:

- Sometimes we want to count with continuous variables. Suppose you have a year variable in your data frame that is of data type integer (e.g. 1982, 1945, 1990). You want to know the number of people for each decade. To do this, you must first convert your year variable to decades before you start counting.
- Often you want to count things per group (for example, the number of players on a particular sports team) and add the counts per group as a new variable to your data frame. You could use joins to do this, but could you do it with less code and more efficiently?
- Counting can also mean that you calculate the sum of a variable within groups (for example, the number of goals scored by a particular team during a season). Normally you would use `group_by` and `summarize` for this calculation. But then you should not forget to `ungroup`. Could you do this without the `group_by` function and with less code?

If you feel that you can't complete these tasks in a snap, read on. I'll introduce you to four tricks and functions that will help you accomplish these tasks effortlessly.

- The first trick lets you count with continuous variables (counting decades, for example).
- With the second trick, you can calculate the sum of a variable within groups without using `group_by`

- The third trick allows you to add the count of a variable as a new variable to your data frame.
- The fourth trick allows you to add a new variable to your data frame that represents the sum of a given variable.

5.1 How to count with continuous variables

Let's start with the first trick. The Starwars dataset from dplyr represents all Starwars characters in a data frame. The data frame has a variable `birth_year`. Suppose you want to know how many of these characters were born in a given decade, not a given year. To calculate decades from years, you need to know how integer division works. Example: I was born in 1986. I might ask myself: How many times does the number ten fit into the number 1986? I could divide 1986 by ten and get 198.6, but that's not the question I'm asking. I want to know how many times 10 fits into the number 1986. For this we need integer division. In R, integer division can be done with the `%/%` operator. If I apply this operator to my birth year, I get the number 198 (`1986 %/% 10`). To convert this number to my decade, I need to multiply this number by 10:

```
10 * (1986 %/% 10)
```

```
[1] 1980
```

 Note

Kudos go to David Robinson who as far as I know came up with this trick and [Olivier Gimenez](#), who wrote an excellent blog describing it.

Now that we know how to perform integer divisions and calculate decades from them, we can combine them with the counting function. Suppose we want to calculate how many of the Starwars characters were born in a given decade. This is how we would do it:

```
starwars %>%
  count(decade = 10 * (birth_year %/% 10),
        name = "characters_per_decade") %>%
  glimpse()
```

```
Rows: 16
Columns: 2
$ decade              <dbl> 0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, ~
$ characters_per_decade <int> 1, 3, 4, 4, 9, 6, 4, 2, 2, 3, 1, 1, 1, 1, 44
```

The real trick is that you create a new variable inside the count function. Up to now, you have probably put an existing variable into the count function. In this example, we have created a new variable `characters_per_decade` that is calculated from the variable `year_of_birth`. We also used the `name` argument to give the count column a more descriptive name.

Now that we know how we can create new variables inside `count`, we can think of new ways to create discrete variables. `ggplot2`, for example, has a [function that creates bins from continuous variables](#). Suppose we want to create bins in the range of 10 for the variable `birth_year`:

```
starwars %>%
  count(height_intervals = cut_width(height, 10))

# A tibble: 18 x 2
  height_intervals     n
  <fct>             <int>
  1 [65,75]            1
  2 (75,85]            1
  3 (85,95]            2
  4 (95,105]           3
  5 (105,115]          1
  6 (115,125]          1
  7 (135,145]          1
  8 (145,155]          2
  9 (155,165]          7
 10 (165,175]          14
 11 (175,185]          20
 12 (185,195]          12
 13 (195,205]          7
 14 (205,215]          3
 15 (215,225]          2
 16 (225,235]          3
 17 (255,265]          1
 18 <NA>                6
```

You can see that the bins each have a range of 10. Also, the bins are surrounded by square brackets and parentheses. A square bracket means that a number is included in the bin, a parenthesis means that the number is not included in the bin. In our second example, this would mean that the year 75 is included, but not the year 85.

5.2 How to calculate the sum of a variable based on groups without using group_by.

The second trick took me some time to understand. Intuitively, one would think that the `count` function counts the values of discrete variables: The number of players on a team, the number of cars, etc. However, `count` can also be used to calculate the sum of a variable for a particular group or groups. Let's first look at how this would be done without the trick. Let's say we want to calculate the sum of unemployed people in the US per year (I wouldn't trust these numbers here, this is just an example of using this technique). Using `group_by` and `summarise` you would do the following:

```
 economics %>%
  mutate(
    year = format(date, "%Y")
  ) %>%
  group_by(year) %>%
  summarise(sum_unemploy = sum(unemploy, na.rm = TRUE))

# A tibble: 49 x 2
# ... with 39 more rows
  year   sum_unemploy
  <chr>     <dbl>
1 1967      18074
2 1968      33569
3 1969      33962
4 1970      49528
5 1971      60260
6 1972      58510
7 1973      52312
8 1974      62080
9 1975      95275
10 1976     88778
# ... with 39 more rows
```

To achieve the same result with `count`, you need to know the argument `wt`. `wt` stands for weighted counts. While `count` calculates the frequency of values within a group without specifying the `wt` argument (`n = n()`), `wt` calculates the sum of a continuous variable for certain groups (`n = sum(<VARIABLE>)`):

```
 economics %>%
  count(year = format(date, "%Y"), wt = unemploy,
        name = "sum_unemploy")
```

```

# A tibble: 49 x 2
  year   sum_unemploy
  <chr>      <dbl>
1 1967        18074
2 1968        33569
3 1969        33962
4 1970        49528
5 1971        60260
6 1972        58510
7 1973        52312
8 1974        62080
9 1975        95275
10 1976       88778
# ... with 39 more rows

```

Again, you can see that we created a new variable on the fly. Also, we used the `wt` argument and set it to the `unemploy` variable. This technique has its advantages and disadvantages. On the positive side, we only need three lines of code instead of six. On the downside, the code is less explicit, and without knowing the inner workings of `count`, it's hard to tell that the function is calculating sums. However, it could be a new option in your toolbox to calculate sums.

5.3 How to add counts as a variable to your data frame

In the previous examples, you saw that the `count` creates a new data frame with the grouping variable and the frequency or sum variable. This is not always what you want. Sometimes you want to add counts to your existing data frame. Let's take the `mpg` data frame. The data frame contains statistics about cars from different manufacturers. Now suppose you want to count how many cars there are per manufacturer, and add those numbers to the `mpg` data frame. This can be done with `add_count()`:

```

mpg %>%
  add_count(manufacturer, name = "number_of_cars_by_manufacturer") %>%
  select(manufacturer, model, number_of_cars_by_manufacturer) %>%
  glimpse()

```

```

Rows: 234
Columns: 3
$ manufacturer      <chr> "audi", "audi", "audi", "audi", "audi", ~
$ model              <chr> "a4", "a4", "a4", "a4", "a4", "a4", ~
$ number_of_cars_by_manufacturer <int> 18, 18, 18, 18, 18, 18, 18, 18, 18, ~

```

As you can see, there are 18 cars from Audi. Why is this useful? Because you keep all the other columns of your data frame. For example, you can use the counts to calculate proportions (number of cars per manufacturer / total number of cars). Also, you avoid using joins to combine two data frames (the counts and the original dataset). Of course, add_count also works with weighted counts.

5.4 How to add a new variable to your data frame that contains the sum of a specific variable

add_tally() does something similar than add_count(). The only difference is that add_tally calculates the sum of a given variable instead of a count. For example, we could add a new variable to mpg that shows the sum of the variables per model.

```
mpg %>%
  group_by(model) %>%
  add_tally(wt = displ, name = "sum_display_per_model") %>%
  select(manufacturer, model, sum_display_per_model) %>%
  glimpse()
```

```
Rows: 234
Columns: 3
Groups: model [38]
$ manufacturer      <chr> "audi", "audi", "audi", "audi", "audi", "audi", ~
$ model              <chr> "a4", "a4", "a4", "a4", "a4", "a4", "a4 qu~
$ sum_display_per_model <dbl> 16.3, 16.3, 16.3, 16.3, 16.3, 16.3, 16.3, 19.4, ~
```

Again, you can see that we calculated the sum by providing the argument `wt` with a continuous variable. The result is a new variable called `sum_display_per_model`. You could get the same result with `add_count` and using `displ` for the argument `wt`. Note that `add_tally` has no argument for grouping the data. You must accomplish this with `group_by`.

i Summary

Here is what you can take from this tutorial.

- With the integer division trick you convert years to decades.
- With `add_count` you can add a count variable to an existing data frame
- With the `wt` argument in `count`, `add_count` and `add_tally` you can calculate the sum of a variable (per group if you want to)

6 How to use extract to create multiple columns from one column

What will this tutorial cover?

In this tutorial you will learn how to extract a character column into multiple columns using `extract`. The `extract` function is basically the `separate` function with super powers and works with groups instead of separators.

Who do I have to thank?

I have to thank two people in particular for this tutorial. First, many thanks to [Tom Henry](#) who covered this topic in his fantastic YouTube [video tidyverse tips & tricks](#). Thanks also to [Mark Dulhunty](#) who showed me in this [post](#) that `extract` is much more powerful than I initially thought. Thanks again to the team working on `stringr`, who taught me about [the concept of non-grouping parentheses](#).

`extract` is the more powerful version of `separate`. The `separate` function allows you to split a character variable into multiple variables. Suppose we want to split the `variable` column in this dataset into two columns:

```
tibble(  
  variable = c("a-b", "a-d", "b-c", "d-e")  
) %>%  
  separate(  
    variable,  
    into = c("a", "b"),  
    sep = "-",  
    remove = FALSE  
)  
  
# A tibble: 4 x 3  
  variable     a     b  
  <chr>       <chr> <chr>  
1 a-b         a     b
```

```

2 a-d      a      d
3 b-c      b      c
4 d-e      d      e

```

This approach reaches its limits quite quickly. Especially if we don't have a clear separator to distinguish the columns we want to create. For these use cases we have `extract`.

The key difference between `separate` and `extract` is that `extract` works with groups within its regular expressions. We will see many examples of this in a minute. For now, let's just say that a group is indicated by two parentheses in regular expressions: `()`. We define groups in `extract` to tell the function which parts of a column should represent a new column.

6.1 How to extract a simple character column

6.1.1 A hyphen

Let's start with the simplest example, a column separated by a hyphen:

```

tibble(
  variable = c("a-b", "a-d", "b-c", "d-e")
) %>%
  extract(
    col = variable,
    into = c("a", "b"),
    regex = "[a-z]-[a-z]",
    remove = FALSE
)
#>

# A tibble: 4 x 3
#>   variable     a     b
#>   <chr>       <chr> <chr>
#> 1 a-b         a     b
#> 2 a-d         a     d
#> 3 b-c         b     c
#> 4 d-e         d     e

```

`extract` takes a few arguments:

- `col` specifies the character column to be split into several columns.
- `into` specifies the name of the columns to be created

- `regex` defines the regular expression in which we capture the groups that will represent the new columns
- `remove` tells the function if the original column should be removed (by default `TRUE`)

The interesting thing about this code are the two groups: `([a-z])` and `([a-z])`. They are separated by a hyphen (`-`). Each captured group is converted into a new column. So instead of thinking of the separator in `separate` with `extract`, we think of groups.

6.1.2 A letter and a number

Let's try this concept on another example. In this data frame, we want to extract two columns from one column. The first one should be the first letter, the second one the number. In fact, in this example we don't even have a separator anymore:

```
tibble(
  variable = c("x1", "x2", "y1", "y2")
)

# A tibble: 4 x 1
  variable
  <chr>
1 x1
2 x2
3 y1
4 y2
```

This is how we would extract both columns with `extract`:

```
tibble(
  variable = c("x1", "x2", "y1", "y2")
) %>%
  extract(
    variable,
    into = c("letter", "number"),
    regex = "([xy])(\\d)",
    remove = FALSE
)

# A tibble: 4 x 3
  variable letter number
  <chr>     <chr>   <chr>
```

```

1 x1      x      1
2 x2      x      2
3 y1      y      1
4 y2      y      2

```

Again, we have defined two groups for the two columns. The first group captures the letters `x` or `y` marked by square brackets (`([xy])`), the second group captures a single number (`(\\d)`).

6.1.3 First name and last name

Now suppose you want to extract the first and last names of famous people:

```

tibble(
  variable = c("David Jude Heyworth Law", "Elton Hercules John",
               "Angelina Jolie Voight", "Jennifer Shrader Lawrence")
) %>%
  extract(
    variable,
    into = c("first_name", "last_name"),
    regex = "(\\w+) .* (\\w+)",
    remove = FALSE
)

# A tibble: 4 x 3
#>   variable           first_name last_name
#>   <chr>              <chr>     <chr>
#> 1 David Jude Heyworth Law    David      Law
#> 2 Elton Hercules John     Elton      John
#> 3 Angelina Jolie Voight   Angelina   Voight
#> 4 Jennifer Shrader Lawrence Jennifer Lawrence

```

What we want to say with the regular expression is the following:

- The first group captures at least 1 letter (`(\\w+)`).
- The column is then followed by a space, and all characters in between are followed by another space: `.*`
- The last group again contains at least 1 letter: `(\\w+)`

You can see that this also works for people who have more than one middle name. This is because the regular expression only captures the last word preceded by a space.

6.2 How to extract more complicated character column

6.2.1 Non-grouping parentheses

To extract columns that are more complicated and confusing, we need to learn the concept of **non-grouping parentheses**. Non-grouping parentheses define groups that are not captured. In other words, these groups are not converted into a new column. But they allow us to extract columns that have some inconsistencies.

Let's take this example: Here we want to create two columns separated by a `->` (this example could also be solved with `separate`, but we will get to more complicated examples in a moment):

```
tibble(
  variable = c("x -> 1",
               "y -> 2",
               "p-> 34")
) %>%
  extract(
    variable,
    into = c("letter", "number"),
    remove = FALSE,
    regex = "([a-z])(?: ?-> ?)(\\d+)?"
  )

# A tibble: 3 x 3
  variable letter number
  <chr>     <chr>   <chr>
1 x -> 1   x       1
2 y -> 2   y       2
3 p-> 34   p       34
```

The most important part here is this: `(?: ?-> ?)`. This is called a non-grouping parenthesis. A non-grouping parenthesis is defined by a group that starts with a question mark and a colon: `(?:)`. The advantage of this method is that we can solve column separation problems caused by messy or inconsistent variables. Let's see what this means in our next example.

6.2.2 Inconsistent separators

In this example, we want to extract columns from a variable that has an arrow as a separator (`->`). However, we might sometimes find two or more arrows in the character string or no arrow at all:

```

df <- tibble(
  variable = c("x ->-> 1",
              "y -> 2",
              "p-> 34",
              "f 4")
)

df %>%
  extract(
    variable,
    into = c("letter", "number"),
    remove = FALSE,
    regex = "[a-z] ?(?:->){}{0,} ?(\d+)?"
  )

# A tibble: 4 x 3
  letter number
  <chr>   <chr>  <chr>
1 x        1
2 y        2
3 p        34
4 f        4

```

Our non-grouping parenthesis looks for any number of arrows between our two new variables: $(?:->){}{0,}$. The 0 indicates that we can also find a character string that does not contain an arrow. Therefore, we are able to extract the last value that lacks the arrow: `f 4`. The comma `{0,}` indicates that the arrow can be repeated infinitely often.

6.2.3 Non-perfect pattern with unnecessary characters

Let us create an even more complicated example. Suppose that we expect to find some typos after the arrow `->aslkdfj`. This problem can also be solved with non-grouping parentheses:

```

df <- tibble(
  variable = c("x ->aslkdfj 1",
              "y-> 2",
              "p 34",
              "8")
)

df %>%

```

```

extract(
  variable,
  into = c("letter", "number"),
  remove = FALSE,
  regex = "[a-z]? ?(?:->\\w*)? ?(\\d+)"
)

# A tibble: 4 x 3
  variable     letter number
  <chr>       <chr>   <chr>
1 x ->aslkdfj 1 "x"    1
2 y-> 2        "y"    2
3 p 34         "p"    34
4 8             ""     8

```

A few things have been changed here. First, I added the regex `\w*` to the non-grouping parenthesis. This regex searches for any number of letters. The asterisk `*` indicates that we expect zero to an infinite number of letters.

You can also see that we may not even find a letter for the first newly created column. Here the question mark `([a-z])?` indicates that the first letter is optional.

6.2.4 A hard example

Let's conclude this tutorial with a tricky example. Imagine that our column actually represents four different columns with the following structure:

- First column: A single number (e.g. 3).
- Separator: A period (e.g. `.`)
- Second column: A number of any length (e.g. 10).
- Separator: An equal sign with or without an enclosing space (e.g. `=`).
- Fourth column: Any number of letters (e.g. `AX`).
- Separator: A colon (e.g. `:`).
- Fourth column: A number (e.g. 40)

Let's first look at how this might work, and then go through the regex in more detail:

```

tibble(
  value = c("3.10 = AX",
            "3.1345 = AX:_40",
            "3.8983 =:$15",
            ".873 = PFS:4")) %>%

```

```

extract(
  value,
  into = c("v0", "v2", "v3", "v4"),
  regex = "(\\d)?\\. (\\d+) ?= ?(?: (\\w+)? :?) ?(?: [?$_]*)(\\d+)?",
  remove = FALSE
)

# A tibble: 4 x 5
#> #>   value     v0    v2    v3    v4
#> #>   <chr>    <chr> <chr> <chr> <chr>
#> 1 3.10 = AX      "3"    10    "AX"   ""
#> 2 3.1345 = AX:?:40 "3"    1345   "AX"   "40"
#> 3 3.8983 =:$15   "3"    8983   ""     "15"
#> 4 .873 = PFS:4    ""    873    "PFS"  "4"

```

Here is a decomposition of the regex:

- $(\\d)?$: The first column is an optional number indicated by the question mark (0 or 1).
- $\\.$: The number is then followed by a period. The period is not optional. There can be only one period.
- $(\\d+)$?: The second column is a required number, which can be followed by a white space.
- $=$?: The second column is followed by an equal sign, which can be followed by one white space
- $(?: (\\w+)? :?)$?: Then we have a non-grouping parenthesis in combination with a group. This actually works. What we want to say here is that the third column could be missing in the column (indicated by the last question mark) and that the third variable is a word ($\\w$). Also the third column can be followed by a colon (?:).
- $(?: [?$_]*)$: A number of typos may occur after the optional colon. Especially the characters ?, _, or \$. There may be zero or many of these characters (indicated by the *).
- $(\\d+)?$: The last column is an optional number.

From this example you should see that `extract` can be a very powerful function when you need to work with very messy columns. The most important part is always the regex. I hope we could strengthen your regex muscle a little bit with this tutorial.

Summary

Here is what you can take from this tutorial.

- While `separate` works with separators, `extract` works with groups.

- A group `(())` in the regex argument in `extract` represents new columns that will be created.
- Non-grouping parentheses can be used to work with messy columns and are not converted to new columns.

7 How to anonymize columns

What will this tutorial cover?

In this tutorial we will cover six different techniques for anonymizing columns. We will find out how to use the function `fct_anon`, how to replace names with random names, how to mask values, how to group numeric variables, how to remove house numbers from street names, and how to encode and decode values.

Who do I have to thank?

For this course, I had to do some reading about anonymization. In particular, I relied on the following blog posts:

- <https://www.record-evolution.de/en/blog/data-anonymization-techniques-and-best-practices-a-quick-guide/>
- <https://satoricyber.com/data-masking/data-anonymization-use-cases-and-6-common-techniques/>
- <https://www.ssl2buy.com/wiki/symmetric-vs-asymmetric-encryption-what-are-differences>

No one wants to make their bank account password or home address public. When such data breaches occur, it is costly and potentially dangerous for those affected. For example in 2020, [private data of Finnish people in psychological care was hacked](#) and published. It even went so far that the hackers forced the victims to pay money to prevent their personal data from being disclosed.

To avoid such data breaches, we as users of R need to think twice about how we handle sensitive data. Therefore, in this tutorial we will cover some techniques to anonymize or pseudoanonymize your data. We will cover these topics:

- Anonymization of a factor column with `fct_anon`
- How to replace names with random names
- Masking values in a column
- Creating groups of numeric variables (e.g. age)
- Removing house numbers from street names
- Encrypting and decrypting columns with `encryptr`

7.1 What is the difference between pseudonymization and anonymization?

Before we begin, we need to talk about pseudonymization and anonymization. The following techniques can be used for pseudoanonymization and anonymization. The difference between the two is that pseudonymization is reversible, while anonymization is not (at least with current technical capabilities). All of the following techniques allow pseudonymization. Not all of them are suitable for anonymization (especially encryption).

The EU defines pseudonymization as follows:

“the processing of personal data in such a manner that the personal data can no longer be attributed to a specific data subject without the use of additional information provided that such additional information is kept separately and is subject to technical and organisational measures to ensure that the personal data are not attributed to an identified or identifiable natural person.” (https://edps.europa.eu/system/files/2021-04/21-04-27_aepd-edps_anonymisation_en_5.pdf)

By this definition, pseudonymization is reversible and requires additional information to reverse the process. Thus, for the following techniques, whether your data is anonymized or pseudoanonymized depends on your actions.

7.2 How to anonymize a factor column with fct_anon

Sometimes you want to make your data completely anonymous so that other people can't see sensitive information. For example, take a person's religion or gender. A simple function to anonymize such discrete data is `fct_anon`. The function takes two arguments. The factor you want to anonymize, and the prefix you put in front of the anonymized factor. Suppose we want to anonymize the factor levels of the `relig` column in the `gss_cat` data frame:

```
levels(gss_cat$relig)
```



```
[1] "No answer"           "Don't know"
[3] "Inter-nondenominational" "Native american"
[5] "Christian"           "Orthodox-christian"
[7] "Moslem/islam"         "Other eastern"
[9] "Hinduism"             "Buddhism"
[11] "Other"                "None"
[13] "Jewish"                "Catholic"
[15] "Protestant"           "Not applicable"
```

With `fct_anon` we can convert these levels into numeric values and add a prefix to them:

```
gss_cat %>%
  mutate(
    relig = fct_anon(relig, prefix = "religion_")
  ) %>%
  glimpse()
```

```
Rows: 21,483
Columns: 9
$ year      <int> 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 20~
$ marital   <fct> Never married, Divorced, Widowed, Never married, Divorced, Mar~
$ age        <int> 26, 48, 67, 39, 25, 25, 36, 44, 44, 47, 53, 52, 52, 51, 52, 40~
$ race       <fct> White, White, White, White, White, White, White, White, ~
$ rincome    <fct> $8000 to 9999, $8000 to 9999, Not applicable, Not applicable, ~
$ partyid   <fct> "Ind,near rep", "Not str republican", "Independent", "Ind,near~
$ relig      <fct> religion_07, religion_07, religion_07, religion_09, religion_1~
$ denom      <fct> "Southern baptist", "Baptist-dk which", "No denomination", "No~
$ tvhours    <int> 12, NA, 2, 4, 1, NA, 3, NA, 0, 3, 2, NA, 1, NA, 1, 7, NA, 3, 3~
```

Note that the numbers are generated randomly. So, each time you run this code, you will get a different set of numbers. If you need to use a factor as an id column, it is better to use other techniques that preserve the levels as unique identifiers (for example, hashing).

7.3 How to replace names with random names

Names are also sensitive data. To anonymize names, you can simply replace them with random names. This can be done with the `randomNames` function from the `randomNames` package:

```
library(randomNames)

presidential %>%
  mutate(
    name = randomNames(nrow(.),
                        sample.with.replacement = FALSE)
  )

# A tibble: 11 x 4
  name          start      end      party
  <fct>     <dbl> <dbl> <dbl>
```

	<chr>	<date>	<date>	<chr>
1	Jensen, Joseph	1953-01-20	1961-01-20	Republican
2	Valenzuela, Hubert	1961-01-20	1963-11-22	Democratic
3	Lor, Bu Sun	1963-11-22	1969-01-20	Democratic
4	Pico, Danzig	1969-01-20	1974-08-09	Republican
5	Tran, David	1974-08-09	1977-01-20	Republican
6	Abdi, Chelsea	1977-01-20	1981-01-20	Democratic
7	Tuttle, Alison	1981-01-20	1989-01-20	Republican
8	Suh, Lauren	1989-01-20	1993-01-20	Republican
9	el-Fayad, Saalim	1993-01-20	2001-01-20	Democratic
10	el-Khalili, Manaahil	2001-01-20	2009-01-20	Republican
11	Peterson, Mary	2009-01-20	2017-01-20	Democratic

In this example we have overwritten the column name with a list of random names and told the function that no name should occur more than once (`sample.with.replacement = FALSE`). Again, you get a different set of names each time you run the function.

If we want to be more specific about how the names are generated, we can provide some additional information to the function. For example, we can specify the ethnicity of the generated names, the order of their first and last names, and a separator that separates the first name from the last:

```
presidential %>%
  mutate(
    name = randomNames(nrow(.),
      sample.with.replacement = FALSE,
      ethnicity = c(1, 2, 4),
      name.order = "first.last",
      name.sep = " ")
  )
```

```
# A tibble: 11 x 4
  name          start      end      party
  <chr>     <date>    <date>    <chr>
1 Antoinette Bayles 1953-01-20 1961-01-20 Republican
2 Brandon Dinh     1961-01-20 1963-11-22 Democratic
3 Karen Blanco-Araujo 1963-11-22 1969-01-20 Democratic
4 Carsen Lynch     1969-01-20 1974-08-09 Republican
5 Shadae Lisette Lin 1974-08-09 1977-01-20 Republican
6 Erika Luna       1977-01-20 1981-01-20 Democratic
7 Autumn Hubbs     1981-01-20 1989-01-20 Republican
8 Nur Xiong        1989-01-20 1993-01-20 Republican
```

9	Amber Horner	1993-01-20	2001-01-20	Democratic
10	Alyssa Puryear	2001-01-20	2009-01-20	Republican
11	Ayush Mohan	2009-01-20	2017-01-20	Democratic

If you are wondering what the names in the ethnicity vector mean, take a look at the documentation. Just hit `?randomNames` in your console and scroll down to ethnicity. For example, 1 stands for “American Indian or Native Alaskan”.

7.4 How to mask values (e.g. credit card numbers)

Another common use case is the masking of values. Credit card numbers are a prime example. You certainly don’t want to reveal a person’s credit card number. Masking is a technique that hides some characters of a string. Mostly by “X”s.

Suppose we want to mask the last digit of the height of Starwars characters. This is how we would do it in Tidyverse:

```
starwars %>%
  mutate(
    height = map_chr(height, ~ str_replace(.x, ".\"", "X"))
  )

# A tibble: 87 x 14
  name      height   mass hair_~1 skin_~2 eye_c~3 birth~4 sex   gender homew~5
  <chr>     <chr>   <dbl> <chr>   <chr>   <chr>   <dbl> <chr> <chr>   <chr>
1 Luke Skywa~ 17X      77 blond   fair    blue     19 male   masculin~ Tatooi~ 
2 C-3PO       16X      75 <NA>    gold    yellow  112 none   masculin~ Tatooi~ 
3 R2-D2        9X      32 <NA>    white,~ red    33 none   masculin~ Naboo  
4 Darth Vader 20X     136 none    white   yellow  41.9 male   masculin~ Tatooi~ 
5 Leia Organa 15X     49 brown   light   brown   19 female feminin~ Aldera~ 
6 Owen Lars    17X     120 brown,~ light   blue    52 male   masculin~ Tatooi~ 
7 Beru White~ 16X      75 brown   light   blue    47 female feminin~ Tatooi~ 
8 R5-D4        9X      32 <NA>    white,~ red    NA none   masculin~ Tatooi~ 
9 Biggs Dark~ 18X      84 black   light   brown   24 male   masculin~ Tatooi~ 
10 Obi-Wan Ke~ 18X     77 auburn~ fair    blue-g~  57 male   masculin~ Stewjon
# ... with 77 more rows, 4 more variables: species <chr>, films <list>,
#   vehicles <list>, starships <list>, and abbreviated variable names
#   1: hair_color, 2: skin_color, 3: eye_color, 4: birth_year, 5: homeworld
```

The trick here lies in the `str_replace` function. The `.x` stands for the piped variable (in this case `height`). Then I provide a regular expression that searches for the last character of the string (`.$`). This character should then be replaced by an `X`.

A more complicated example arises when we want to mask more than one character. So let's return to our credit card example. Here is a data frame with credit card numbers:

```
ccards <- tibble(  
  creditcards = c(  
    36555224524299,  
    36350489667466,  
    36002887965170,  
    5447552069207504,  
    2221002654361034,  
    5127699386148536)  
)
```

Let's convert the first 10 characters of these credit card numbers to "X"s:

```
ccards %>%  
  mutate(  
    creditcards_masked = map_chr(creditcards, ~ str_replace(.x, "^.{10}",  
                                              replacement = strrep("X", 10)))  
)  
  
# A tibble: 6 x 2  
  creditcards creditcards_masked  
  <dbl> <chr>  
1 3.66e13 XXXXXXXXXX4299  
2 3.64e13 XXXXXXXXXX7466  
3 3.60e13 XXXXXXXXXX5170  
4 5.45e15 XXXXXXXXXXXX207504  
5 2.22e15 XXXXXXXXXXXX361034  
6 5.13e15 XXXXXXXXXXXX148536
```

This code is a little more complicated than the first one. The regular expression `^.{10}` indicates that we are looking for the first 10 characters of the string. We replace this pattern with 10 "X"s, specified by `strrep("X", 10)`. The function `strrep` is a basic function of R, which simply repeats a series of characters:

```
strrep("X", 10)
```

```
[1] "XXXXXXXXXX"
```

Similarly, we could replace the last 5 characters with “X”s:

```
ccards %>%
  mutate(
    creditcards = map_chr(creditcards, ~ str_replace(.x, "\\d{5}$",
                                                    replacement = strrep("X", 5)))
  )

# A tibble: 6 x 2
  creditcards creditcars
  <dbl> <chr>
1 3.66e13 365552245XXXXX
2 3.64e13 363504896XXXXX
3 3.60e13 360028879XXXXX
4 5.45e15 54475520692XXXXX
5 2.22e15 22210026543XXXXX
6 5.13e15 51276993861XXXXX
```

In the regular expression `\\d{5}$` we look for the last five digits of a string.

7.5 How to turn ages into groups

Another common technique for anonymizing data is to divide it into groups. Suppose we want to divide the ages of people into groups.

Let's first calculate the age of our Starwars characters again:

```
(age_starwars <- starwars %>%
  mutate(age = as.integer(format(Sys.Date(), "%Y")) - birth_year) %>%
  select(name, age) %>%
  drop_na(age))

# A tibble: 43 x 2
  name           age
  <chr>         <dbl>
1 Luke Skywalker 2004
2 C-3PO          1911
3 R2-D2          1990
```

```

4 Darth Vader      1981.
5 Leia Organa     2004
6 Owen Lars        1971
7 Beru Whitesun lars 1976
8 Biggs Darklighter 1999
9 Obi-Wan Kenobi    1966
10 Anakin Skywalker 1981.
# ... with 33 more rows

```

Let us now go through four techniques we can use to group this age variable. With the function `cut_width` we can create groups of arbitrary width from a numeric variable. So let's group age into groups of 10 years:

```

age_starwars %>%
  mutate(
    age_groups = cut_width(age, 10)
  )

# A tibble: 43 x 3
  name          age age_groups
  <chr>        <dbl> <fct>
1 Luke Skywalker 2004  (1995,2005]
2 C-3PO           1911  (1905,1915]
3 R2-D2            1990  (1985,1995]
4 Darth Vader     1981. (1975,1985]
5 Leia Organa     2004  (1995,2005]
6 Owen Lars        1971  (1965,1975]
7 Beru Whitesun lars 1976  (1975,1985]
8 Biggs Darklighter 1999  (1995,2005]
9 Obi-Wan Kenobi    1966  (1965,1975]
10 Anakin Skywalker 1981. (1975,1985]
# ... with 33 more rows

```

The round bracket means that a number is not included in the set. The square bracket means that a number is included in the set.

The function `cut_number` creates a certain number of sets. For example, we could say that the age column should be grouped into 10 groups:

```

age_starwars %>%
  mutate(
    age_groups = cut_number(age, 10)
  )

```

```

)
# A tibble: 43 x 3
  name              age age_groups
  <chr>            <dbl> <fct>
1 Luke Skywalker    2004  (2002,2015]
2 C-3PO             1911  [1127,1923]
3 R2-D2             1990  (1982,1992]
4 Darth Vader       1981. (1977,1982]
5 Leia Organa       2004  (2002,2015]
6 Owen Lars          1971  (1966,1971]
7 Beru Whitesun lars 1976  (1971,1977]
8 Biggs Darklighter 1999  (1992,2002]
9 Obi-Wan Kenobi     1966  (1966,1971]
10 Anakin Skywalker  1981. (1977,1982]
# ... with 33 more rows

```

Note, however, that the width of each group varies. For example, Luke Skywalker is in the set between 2011 and 2014 (a difference of 13 years) and Darth Vader is in the set between 1976 and 1981 (a difference of 5 years).

Then we can convert the age to millennia or decades. For example, Luke Skywalker is 2003 years old (at least in this data frame; I honestly don't know much about Starwars, so be gentle :)), so his age should fall in the millennia 2000. That's how we would do it with [David Robinson's](#) decade trick:

```

age_starwars %>%
  mutate(
    millennium = 1000 * (age %/% 1000)
  )

```

```

# A tibble: 43 x 3
  name              age millennium
  <chr>            <dbl>      <dbl>
1 Luke Skywalker    2004        2000
2 C-3PO             1911        1000
3 R2-D2             1990        1000
4 Darth Vader       1981.       1000
5 Leia Organa       2004        2000
6 Owen Lars          1971        1000
7 Beru Whitesun lars 1976        1000

```

```

8 Biggs Darklighter 1999      1000
9 Obi-Wan Kenobi     1966      1000
10 Anakin Skywalker   1981.    1000
# ... with 33 more rows

```

In a similar way, we could convert ages into decades:

```

age_starwars %>%
  mutate(
    decade = 10 * (age %% 10)
  )

# A tibble: 43 x 3
  name          age decade
  <chr>        <dbl>  <dbl>
1 Luke Skywalker 2004  2000
2 C-3PO          1911  1910
3 R2-D2          1990  1990
4 Darth Vader    1981. 1980
5 Leia Organa    2004  2000
6 Owen Lars      1971  1970
7 Beru Whitesun lars 1976  1970
8 Biggs Darklighter 1999  1990
9 Obi-Wan Kenobi 1966  1960
10 Anakin Skywalker 1981. 1980
# ... with 33 more rows

```

7.6 How to remove house numbers from street names

Then we have street names and especially street numbers that sometimes need to be anonymized. Suppose we want to remove the street numbers of these streets:

```

street_names <- tibble(
  street_name = c("Bromley Lanes 34",
                 "Woodsgate Avenue 12",
                 "Ardconnel Terrace 99",
                 "Gipsy Birches 45",
                 "Legate Close 8",
                 "Stevenson Oval 9",
                 "St Leonard's Boulevard 112",

```

```

    "Copper Chare 435",
    "Glastonbury Glebe 82",
    "Southern Way 91")
)

```

To remove them, we can use the `str_remove_all` function from the `stringr` package:

```

street_names %>%
  mutate(
    street_names_no_number = str_remove_all(street_name, "\\d")
  )

# A tibble: 10 x 2
  street_name      street_names_no_number
  <chr>            <chr>
1 Bromley Lanes 34 "Bromley Lanes "
2 Woodsgate Avenue 12 "Woodsgate Avenue "
3 Ardconnel Terrace 99 "Ardconnel Terrace "
4 Gipsy Birches 45 "Gipsy Birches "
5 Legate Close 8   "Legate Close "
6 Stevenson Oval 9 "Stevenson Oval "
7 St Leonard's Boulevard 112 "St Leonard's Boulevard "
8 Copper Chare 435 "Copper Chare "
9 Glastonbury Glebe 82 "Glastonbury Glebe "
10 Southern Way 91  "Southern Way "

```

With the regular expression `\\d` we can remove all digits from a string.

7.7 How to encrypt and decrypt columns

Finally, we can anonymize each column by encrypting it. Encryption is a complicated and vast topic. I can't and won't go into detail here, but let me give you a brief introduction to how it works in R.

When we encrypt a column, we convert the values of a column into another form, which we call ciphertext. The ciphertext is not readable by humans, but it can be converted back to the original value. There are two forms of encryption. Symmetric encryption, where a single key is used to encrypt and decrypt a value, and asymmetric encryption, where two keys are used to encrypt and decrypt a value. A key is plaintext that translates between the two representations. Once you have the key in symmetric encryption, you can decrypt values. To decrypt values in asymmetric encryption, you need the public key and the private key.

The public key is as it says public, so open to anyone. The private key is a key you should not share with anyone. Only when you have both, can you decrypt a value. Also, private key cannot be guessed from the public key. To add another level of security, the private key also sometimes has a passphrase (or password) to it.

So much for the short theory, let's encrypt a column in R. For example, suppose you have this table of usernames and passwords:

```
users <- tibble(  
  name = c("Alexander", "Marie", "John"),  
  password = c(12345, "8$43_45*", "becker23#")  
)
```

We can encrypt this data with the package `encrptr`. First we need to load the package and create the private and public keys using the `genkeys` function:

```
library(encrptr)  
  
# genkeys() generates a public and private key pair  
# Passphrase: 456#7  
genkeys()
```

The function prompted us to provide a passphrase for the private key. This passphrase and the private key should not be shared with anyone!

Once we have the passphrase, we can encrypt our columns. Let's encrypt the password column:

```
users_encrypted <- users %>%  
  encrypt(password)  
  
users_encrypted %>%  
  glimpse()  
  
Rows: 3  
Columns: 2  
$ name      <chr> "Alexander", "Marie", "John"  
$ password <chr> "5b084bd19feef7b413b7a5ec4417f4c28b06e1f6ba8c91..."
```

As you can see, the passwords have been converted into a long string. It is not possible to decrypt the passwords from this string. To decrypt the column, we simply use the `decrypt` function:

```
users_encrypted %>% decrypt(password)
```

You must provide the passphrase to decrypt the column. Also, this works only if the R file is in the same directory as the public and private keys.

Summary

Here is what you can take from this tutorial.

- Anonymization and pseudoanonymization are two different techniques. Anonymization is not reversible, pseudoanonymization is.
- Whether your data is anonymized or pseudoanonymized depends on how you handle the data.
- With asymmetric encryption, you have a public key and a private key. Only with both keys you can decrypt a message.

8 How to lump factor levels

What will this tutorial cover?

In this tutorial, we will talk about four functions that allow you to lump factors together:

`fct_lump_min`, `fct_lump_prop`, `fct_lump_n`, `fct_lump_lowfreq`.

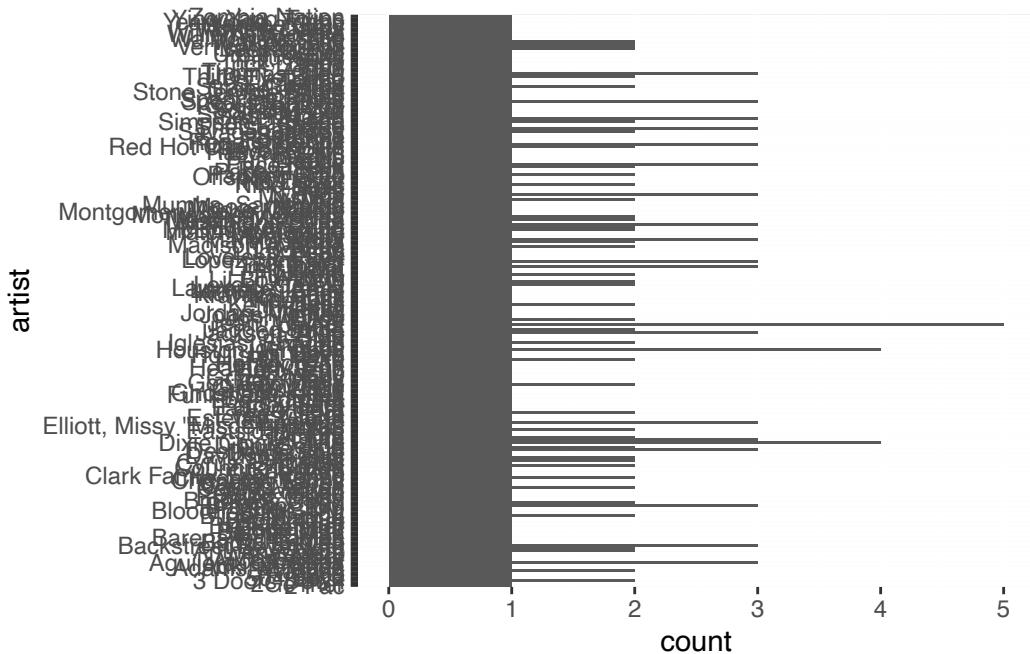
Who do I have to thank?

For this tutorial I relied on the official [forcats documentation](#)

A factor is a data structure in R that allows you to create a set of categories. We call these categories levels. It is well known in the psychological literature that we can only store a certain number of things in our working memory. Therefore, to help people make sense of categories, we shouldn't show too many of them. This is where the strength of lumping factors shows.

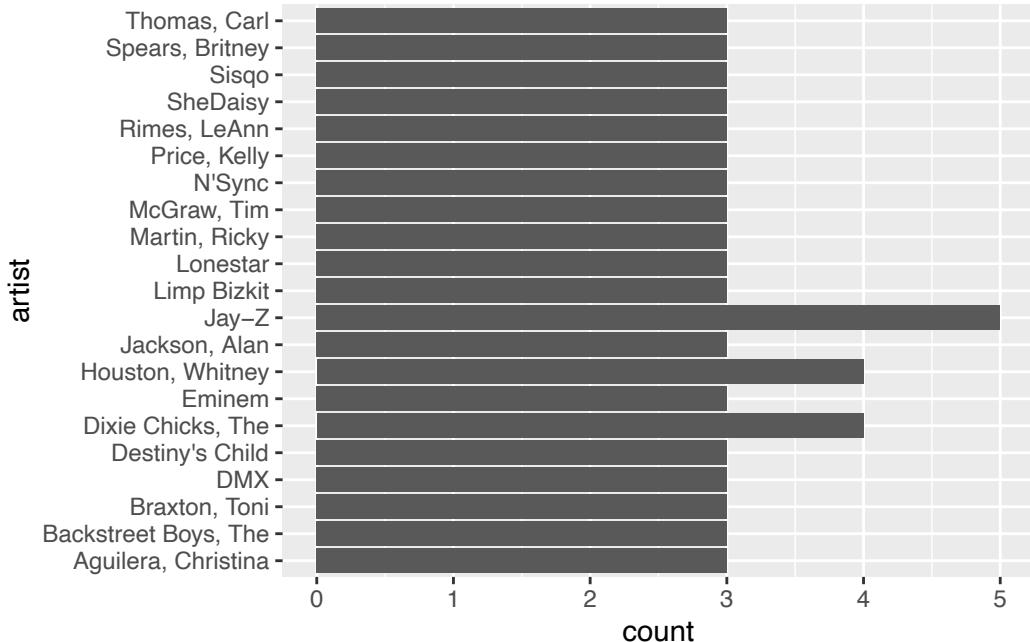
Lumping is nothing more than combining factor levels into a new and larger category. Let's say we want to visualize how many musicians made it to the top 100 in 2000. Here is a bar chart that shows what this looks like:

```
billboard %>%
  ggplot(aes(y = artist)) +
  geom_bar()
```



This visualization is completely incomprehensible. `fct_lump` comes to our rescue:

```
billboard %>%
  mutate(artist = fct_lump(as_factor(artist), 10)) %>%
  filter(artist != "Other") %>%
  ggplot(aes(y = artist)) +
  geom_bar()
```



We provide `fct_lump` with a number. The number indicates, well, what? 10 certainly does not indicate the remaining levels that were not categorized under “Other”. Nor is 10 the levels we lumped together. As it turns out, `fct_lump` is a pretty obscure function because it chooses different methods based on its arguments. The tidyverse team no longer recommends the use of this function.

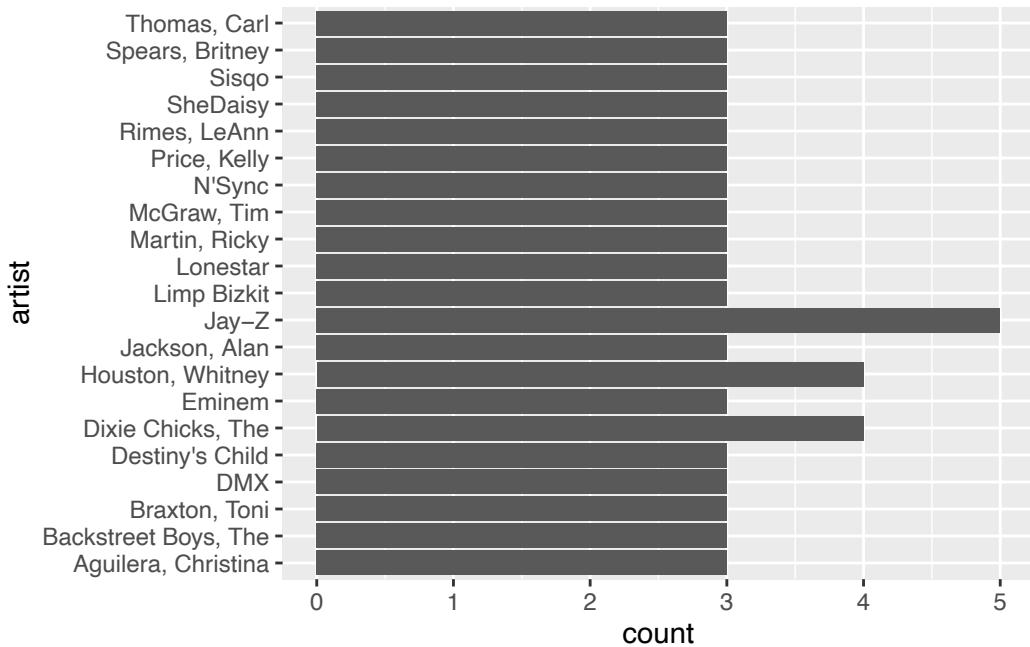
That’s why four new functions were created in 2020, which we’ll go through in this tutorial:

- `fct_lump_min`: lump levels that do not occur more than min times.
- `fct_lump_n`: lumps n of the most or least frequently occurring levels
- `fct_lump_prop`: lumps of levels that occur at most n times * prop
- `fct_lump_lowfreq`: lumps the least frequent levels

8.1 `fct_lump_min`: How to lump levels that occur no more than min times

Let’s start with the simplest function. `fct_lump_min` summarizes all levels that do not appear more than min times. Let’s stay with our billboard example. Let’s assume we want to lump together all levels (or musicians) that made it into the top 100 less than three times in 2000:

```
billboard %>%
  mutate(artist = fct_lump_min(as_factor(artist), 3)) %>%
  filter(artist != "Other") %>%
  ggplot(aes(y = artist)) +
  geom_bar()
```



Clearly, all the other musicians had at least three songs that made it into the Top 100.

Let's try another example. The dataset `gss_cat` contains data on the General Social Survey. This survey contains, among other things, data on the marital status, age or stated income of persons in the years 2000 to 2014. The column `income` stands for the stated income:

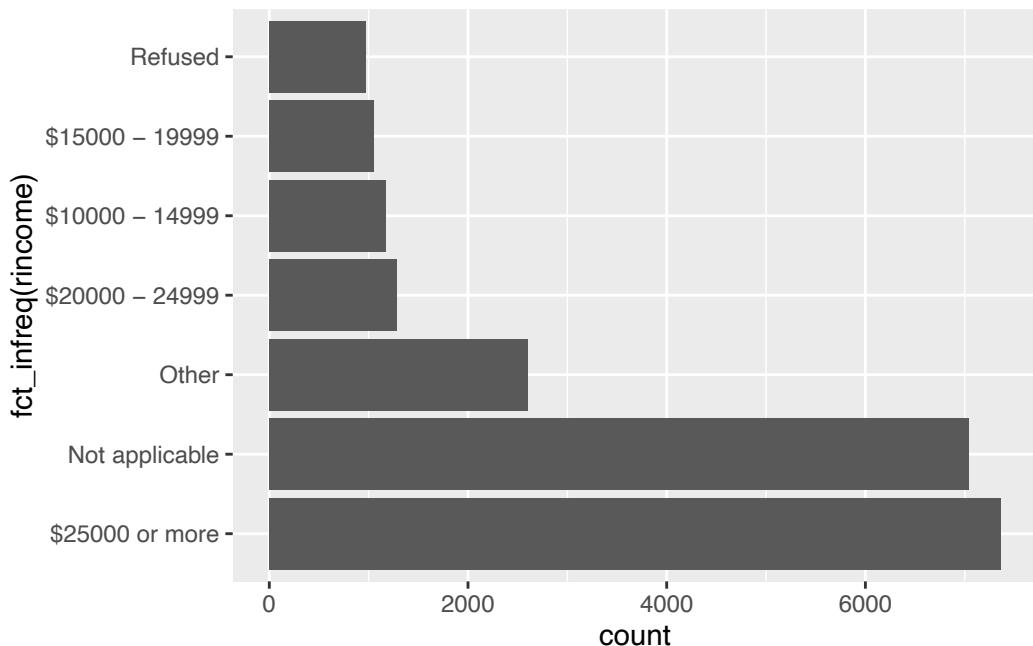
```
table(gss_cat$rincome)
```

No answer	183	Don't know	267	Refused	975	\$25000 or more	7363	\$20000 - 24999	1283
\$15000 - 19999	1048	\$10000 - 14999	1168	\$8000 to 9999	340	\$7000 to 7999	188	\$6000 to 6999	215
\$5000 to 5999	227	\$4000 to 4999	226	\$3000 to 3999	276	\$1000 to 2999	395	Lt \$1000	286

```
Not applicable  
7043
```

Now suppose we want to lump together all incomes that occur less than 600 times:

```
gss_cat %>%  
  mutate(rincome = fct_lump_min(rincome, 600)) %>%  
  ggplot(aes(y = fct_infreq(rincome))) +  
  geom_bar()
```



This time I kept the “Other” level. As you can see, six levels occurred more than 600 times and were therefore not lumped together.

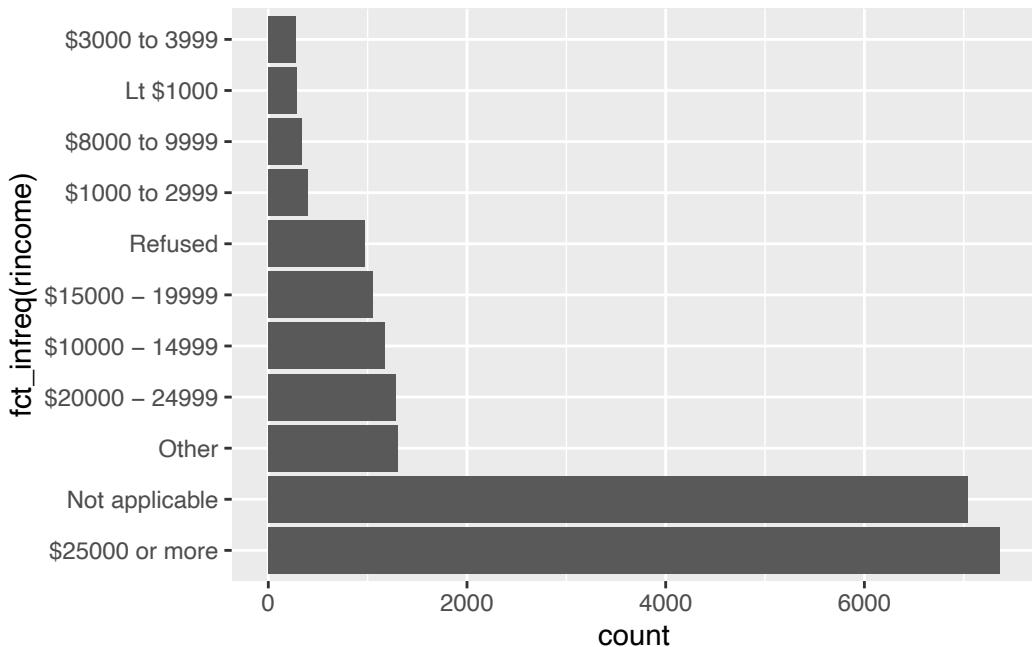
8.2 `fct_lump_n`: Lumps n of the most or least frequent levels

Compared to `fct_lump_min`, `fct_lump_n` is not about the number of levels. Instead, it simply keeps the most frequent levels or the least frequent levels. For example, in our survey example, we might say that we want to lump all levels except the 10 most frequent:

```

gss_cat %>%
  mutate(rincome = fct_lump_n(rincome, n = 10)) %>%
  ggplot(aes(y = fct_infreq(rincome))) +
  geom_bar()

```



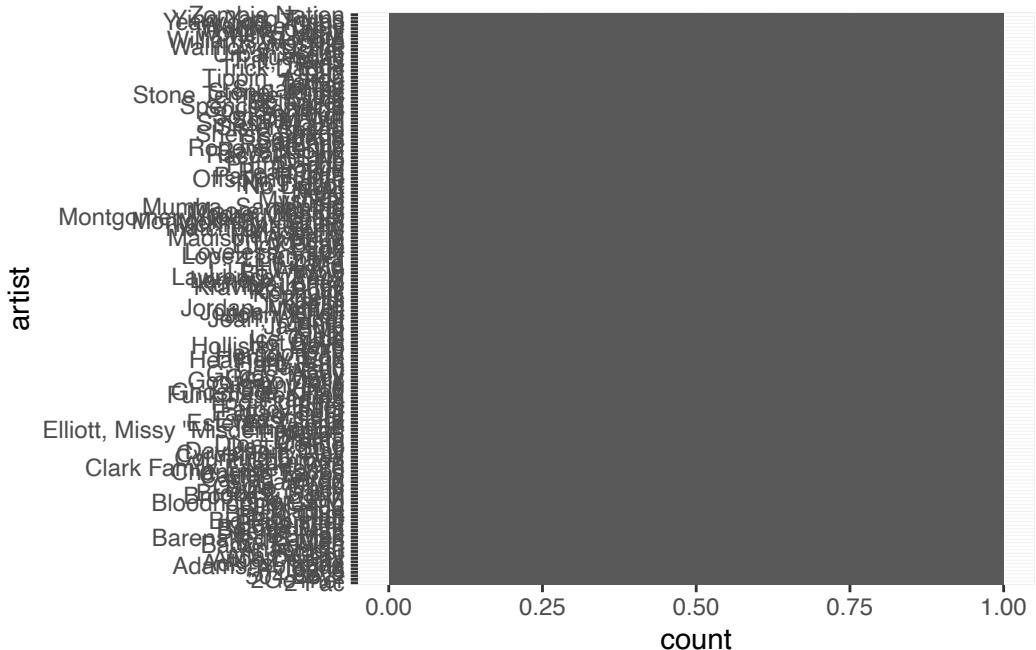
This gives a total of 11 levels (including the `Other` level).

If you specify the `n` argument with a negative number, the function will lump all the levels that occur most often (exactly the opposite of what a positive number does). For example, we could lump together the 5 musicians with the most songs in the Top 100.

```

billboard %>%
  mutate(artist = fct_lump_n(artist, n = -5)) %>%
  filter(artist != "Other") %>%
  ggplot(aes(y = artist)) +
  geom_bar()

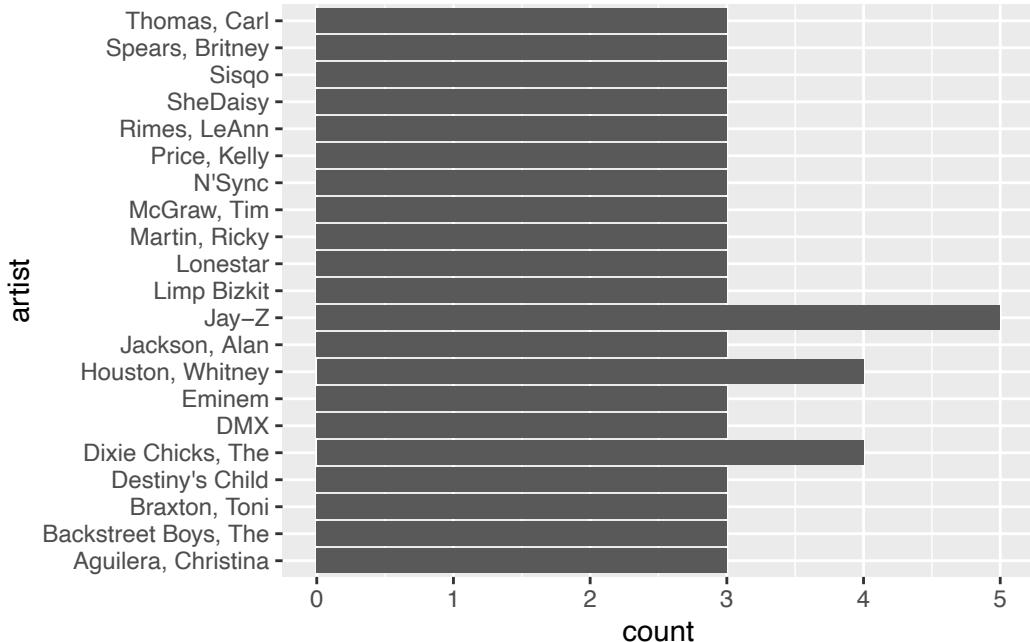
```



This is not a bar chart you want to design, but it proofs the point.

Let's try another example. Let's say we want to lump all levels together except for the 5 most common:

```
billboard %>%
  mutate(artist = fct_lump_n(artist, 5)) %>%
  filter(artist != "Other") %>%
  ggplot(aes(y = artist)) +
  geom_bar()
```

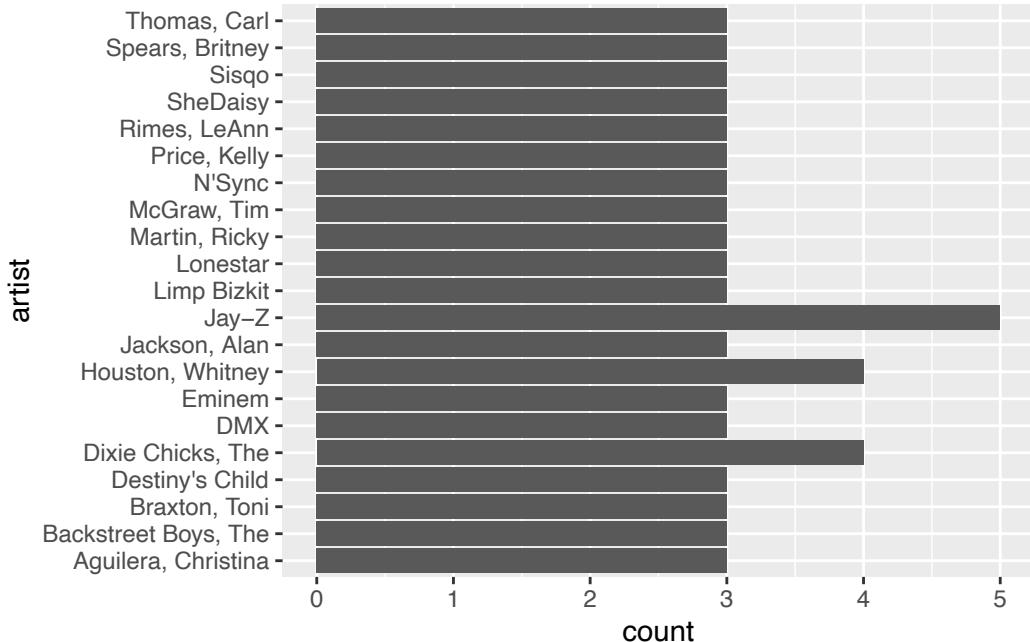


Clearly, these are not five levels. What went wrong? It turns out that many levels occur three times. So we have to decide what to do with the levels that occur the same number of times. The most common three levels are Jay-Z, Whitney Houston and The Dixie Chicks. But what should be the 4th and 5th most frequent levels?

If you don't give the function any additional information, `fct_lump_n` will show you all the levels whose number falls below the last level, which is clearly one of the most frequent levels.

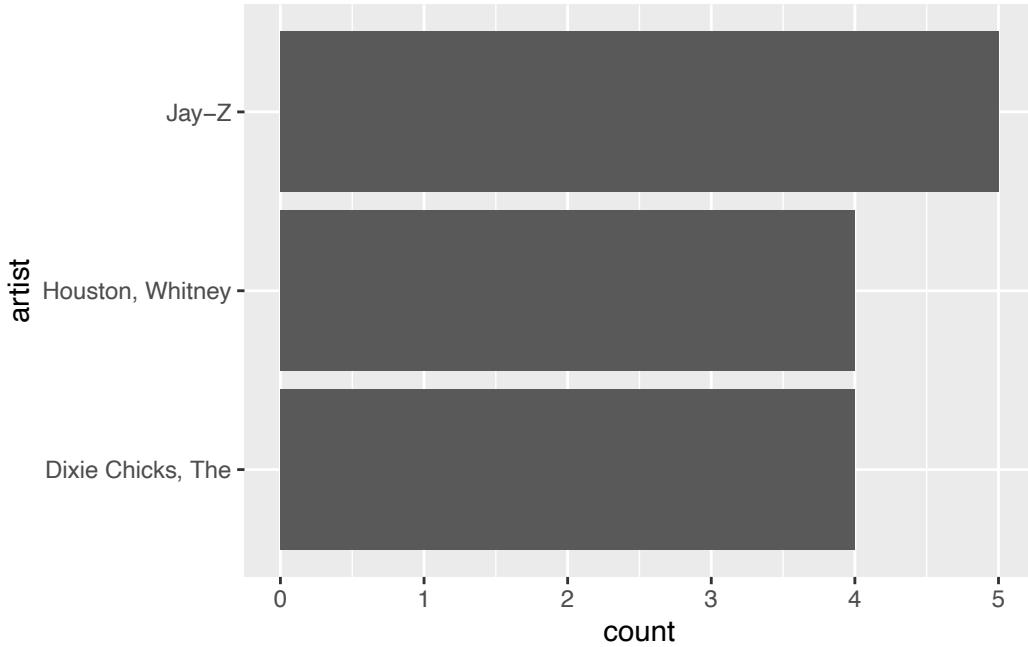
You can change this behavior with the `ties.method` argument. The default argument is `min`, which we have just seen:

```
billboard %>%
  mutate(artist = fct_lump_n(artist, 5, ties.method = "min")) %>%
  filter(artist != "Other") %>%
  ggplot(aes(y = artist)) +
  geom_bar()
```



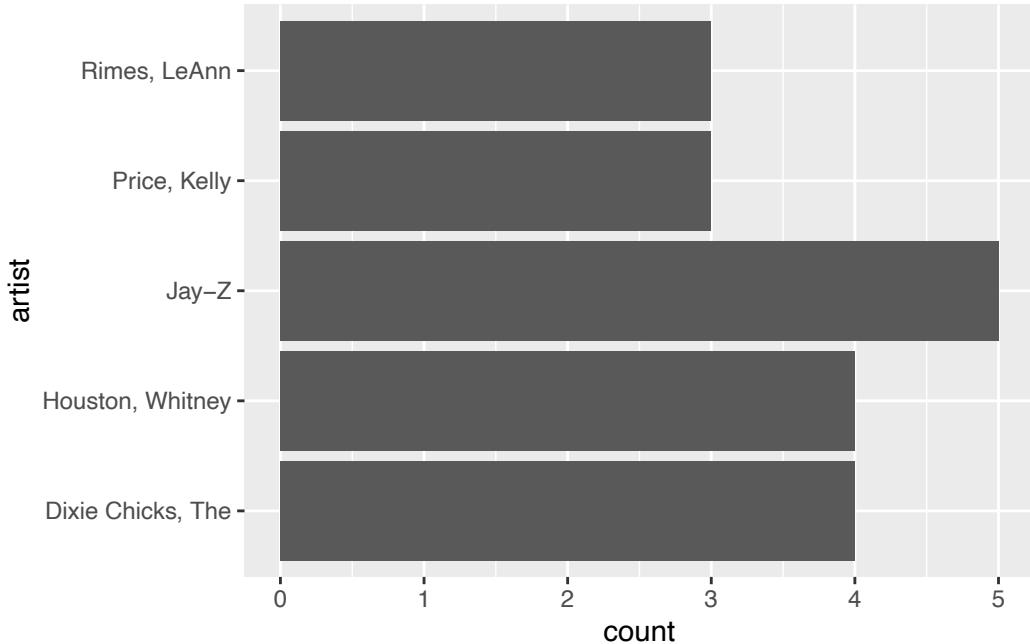
The other options are “average”, “first”, “last”, “random” and “max”. We’ll not go through them all in detail here, but let’s take a look at two of them. “max” removes all levels that cannot be uniquely identified.

```
billboard %>%
  mutate(artist = fct_lump_n(artist, 5, ties.method = "max")) %>%
  filter(artist != "Other") %>%
  ggplot(aes(y = artist)) +
  geom_bar()
```



"random" randomly selects the levels that cannot be uniquely identified as the most frequent levels:

```
billboard %>%
  mutate(artist = fct_lump_n(artist, 5, ties.method = "random")) %>%
  filter(artist != "Other") %>%
  ggplot(aes(y = artist)) +
  geom_bar()
```



You will probably see another set of steps, but that is a feature and not a bug.

8.3 fct_lump_prop: Lumps levels that appear no more than n * prop times

The next function is a bit more complicated. To understand what it does, we need a bit of math.

First we count how many times all levels occur in total. Let's try it with our `gss_cat` data frame:

```
(total_count_income <- gss_cat %>% count(rincome) %>% sum(.n))
```

```
[1] 21483
```

We have data of 21483 incomes. Next, we choose a specific income range and that is the range between “\$20000 - 24999”. This range occurs so often:

```
(count_one_range <- gss_cat$rincome[gss_cat$rincome == "$20000 - 24999"] %>% length)
```

```
[1] 1283
```

This figure represents about 6% of all data points on people's incomes:

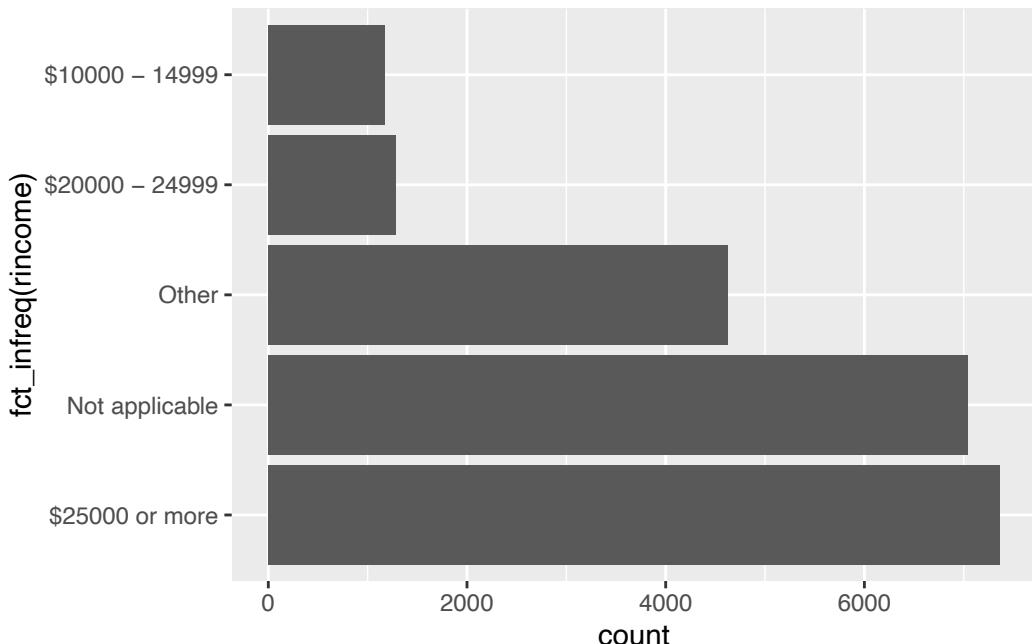
```
count_one_range / total_count_income
```

```
[1] 0.05972164
```

And this is the number we are looking for in `fct_lump_prop`. It represents the percentage at which a particular level occurs within the total number of levels.

Now suppose we want to lump together all the levels that occur in less than 5% of all counts:

```
gss_cat %>%
  mutate(rincome = fct_lump_prop(rincome, .05)) %>%
  ggplot(aes(y = fct_infreq(rincome))) +
  geom_bar()
```



Let's prove this with some Tidyverse functions:

```

gss_cat %>%
  count(rincome, name = "count_per_income_range") %>%
  select(rincome, count_per_income_range) %>%
  mutate(
    total_count_income = sum(count_per_income_range),
    percentage = count_per_income_range / total_count_income
  ) %>%
  filter(percentage >= .05)

# A tibble: 4 x 4
  rincome      count_per_income_range total_count_income percentage
  <fct>                <int>            <int>        <dbl>
1 $25000 or more       7363             21483        0.343
2 $20000 - 24999       1283             21483        0.0597
3 $10000 - 14999       1168             21483        0.0544
4 Not applicable        7043             21483        0.328

```

As you can see, all four levels occur more often than 6%.

8.4 fct_lump_lowfreq: Lumps the least frequent levels

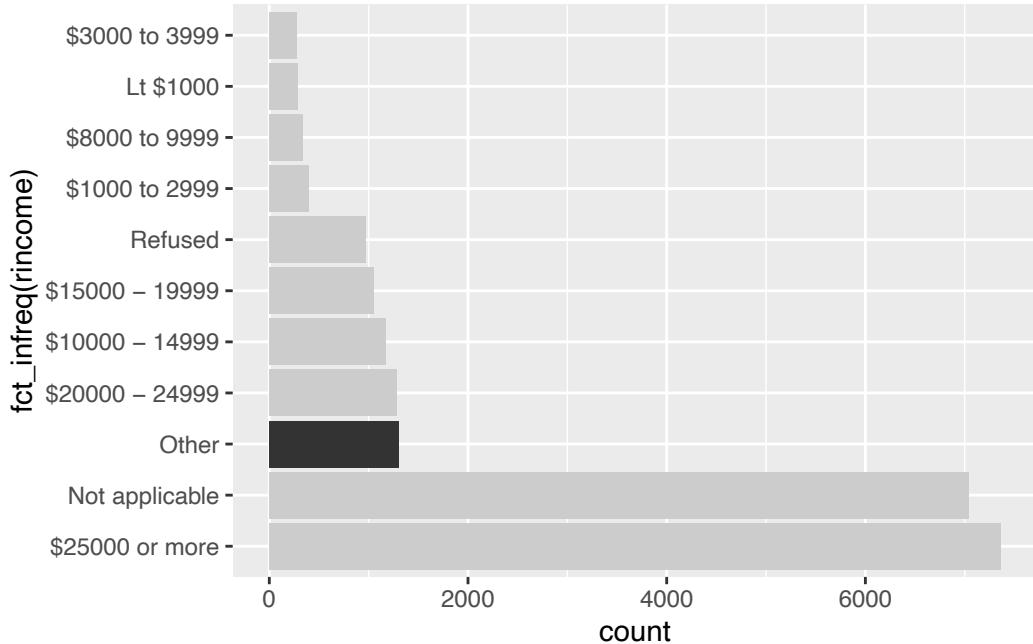
The last function is quite nifty. It has no additional arguments except `other_level`, which is used to specify the name of the “other” level.

This is how it works. You may have realized that some of these four functions cause the `Other` level not to be the least common level:

```

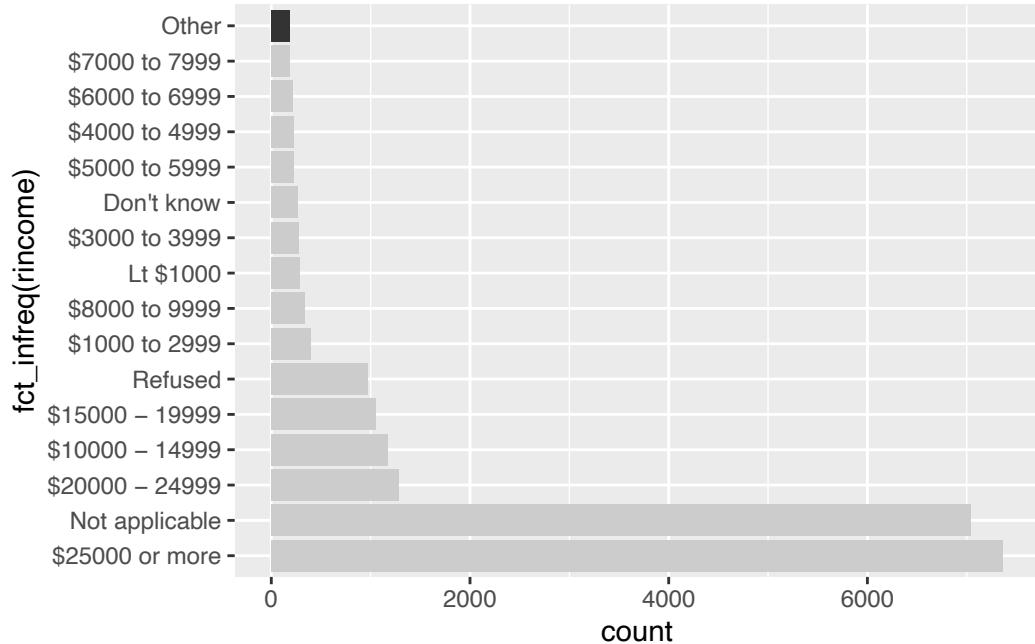
gss_cat %>%
  mutate(
    rincome = fct_lump_n(rincome, n = 10),
    color_coding_rincome = ifelse(rincome == "Other", "a", "b")
  ) %>%
  ggplot(aes(y = fct_infreq(rincome),
             fill = color_coding_rincome)) +
  scale_fill_manual(values = c("grey20", "grey80")) +
  geom_bar(show.legend = FALSE)

```



`fct_lump_lowfreq` simply ensures that so many levels are grouped together that the “Other” is still the least frequent level. Let’s try the same example:

```
gss_cat %>%
  mutate(
    rincome = fct_lump_lowfreq(rincome),
    color_coding_rincome = ifelse(rincome == "Other", "a", "b")
  ) %>%
  ggplot(aes(y = fct_infreq(rincome),
             fill = color_coding_rincome)) +
  scale_fill_manual(values = c("grey20", "grey80")) +
  geom_bar(show.legend = FALSE)
```



This is an amazingly simple but effective idea.

i Summary

Here is what you can take from this tutorial.

- Do not use `fct_lump` anymore. Use the other factor functions instead
- If you need to visualize factors quickly, use `fct_lump_lowfreq` to find the best size of “Other”.
- With `fct_lump_min` you lump all levels that occur less than min times, with `fct_lump_n` you keep the most frequent n levels.

9 How to order factor levels

What will this tutorial cover?

In this tutorial you will learn about four functions that allow you to order factor levels: `fct_relevel`, `fct_infreq`, `fct_reorder`, and `fct_reorder2`.

Who do I have to thank?

I have Claus O. Wilke to thank for this tutorial. He wrote [a popular presentation](#) explaining the topic excellently (also check out [his tweet](#)). My thanks also go to the makers of the `datavizpyr` website, who wrote a great article about the `fct_reorder` function.

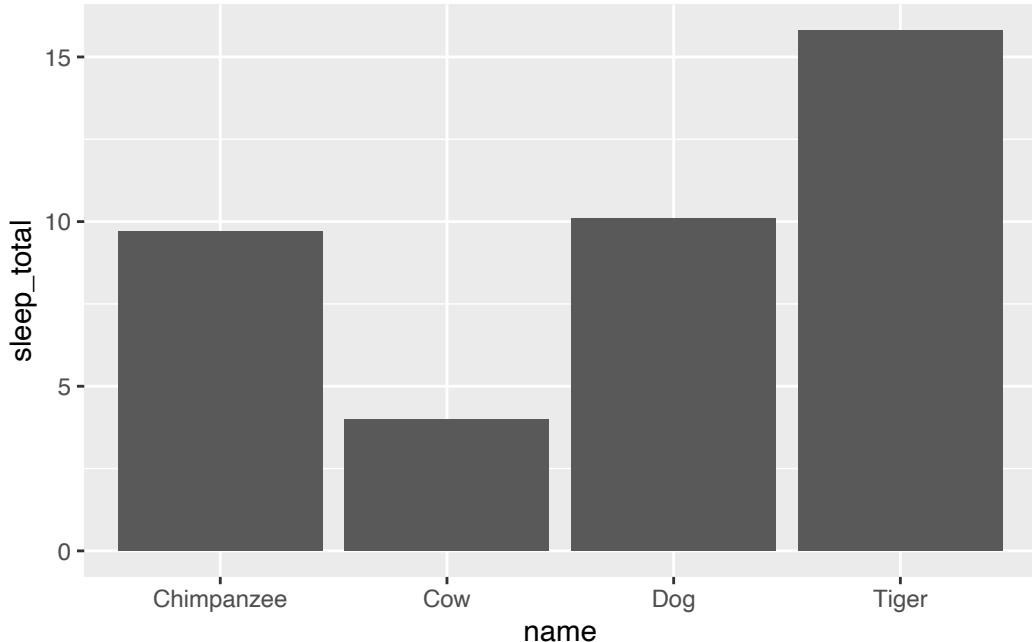
A factor is an ordered data structure in R. What is ordered in a factor are its levels. Usually, you need to know the order of levels when you try to visualize factor columns. Let's say you want to order the bars in your bar chart. Or you want to order the facets in your chart. Or you want to order the lines in your line chart. In this tutorial, we will delve into four functions from the `forcats` package that allow us to order factor levels:

- `fct_relevel`: Order the levels manually
- `fct_infreq`: Order the levels based on how frequently each level occurs
- `fct_reorder`: Order the levels based on the values of a numeric variable
- `fct_reorder2`: Order the levels based on the values of two numeric variables

9.1 How to order factor levels manually

Suppose you created this bar chart showing how many hours cows, dogs, tigers, and chimpanzees sleep per day:

```
msleep %>%
  filter(name %in% c("Cow", "Dog", "Tiger", "Chimpanzee")) %>%
  ggplot(aes(x = name, y = sleep_total)) +
  geom_col()
```



Let's say you want to put cows and dogs first in the bar chart. To make this task a little easier for us, let me first show how this works without a visualization. Here is the same column as a factor:

```
animal_factor <- as.factor(c("Cow", "Dog", "Tiger", "Chimpanzee"))
```

You can see how the factor levels are arranged by running the `levels` function:

```
levels(animal_factor)
```

```
[1] "Chimpanzee" "Cow"          "Dog"          "Tiger"
```

If you look at our bar chart again, you will notice that the output has the same order as the bars in the bar chart. We can change this order manually by using the function `fct_relevel`. It literally says what it does. “re”, according to the [Webster Dictionary](#), means to place something in front. So, in essence, we are putting some levels in front of others with `fct_relevel`. Let's try this:

```
fct_relevel(.f = animal_factor,
            "Cow", "Dog")
```

```
[1] Cow          Dog          Tiger         Chimpanzee  
Levels: Cow Dog Chimpanzee Tiger
```

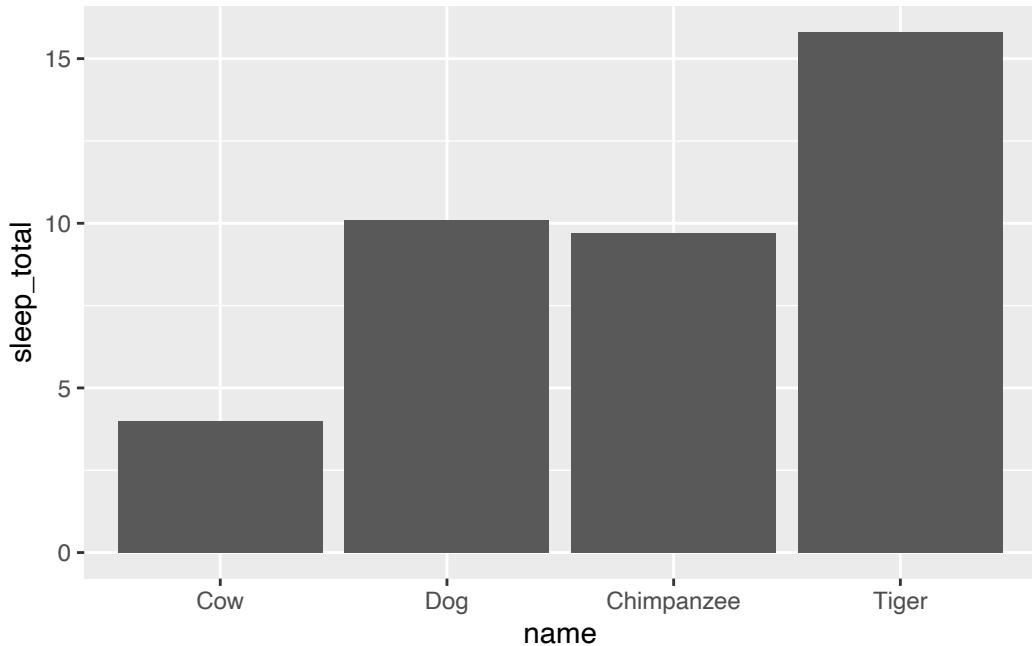
The first argument is the factor column. Then you list the levels you want to place first. You could also do this with a vector:

```
fct_relevel(.f = animal_factor,  
            c("Cow", "Dog"))
```

```
[1] Cow          Dog          Tiger         Chimpanzee  
Levels: Cow Dog Chimpanzee Tiger
```

Let's try this in our visualization:

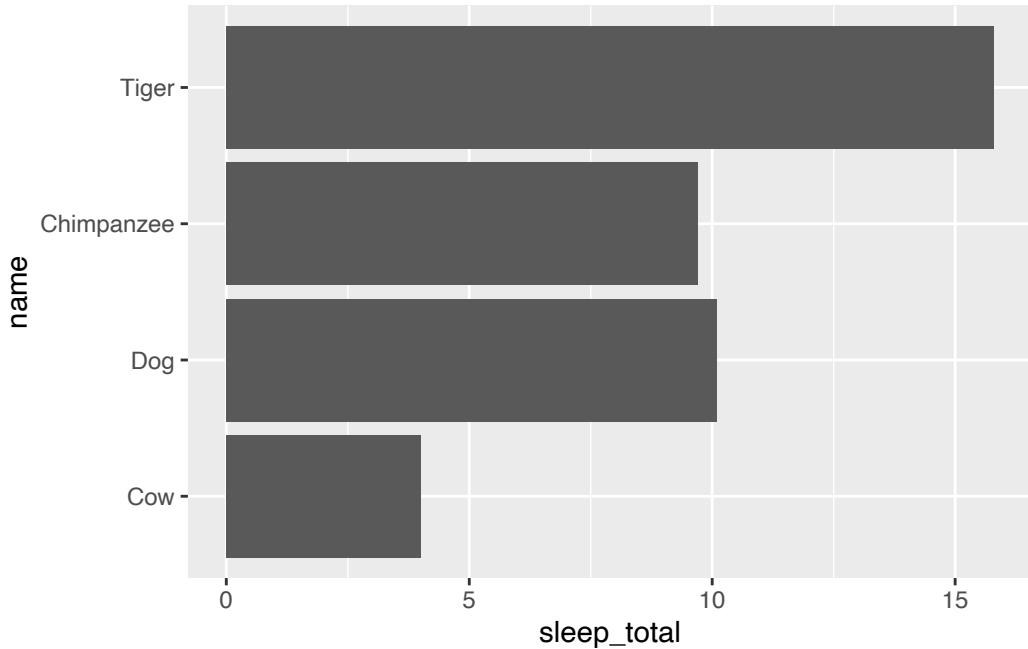
```
msleep %>%  
  dplyr::filter(name %in% c("Cow", "Dog", "Tiger", "Chimpanzee")) %>%  
  mutate(  
    name = fct_relevel(name, c("Cow", "Dog"))  
  ) %>%  
  ggplot(aes(x = name, y = sleep_total)) +  
  geom_col()
```



You may wonder what scheme the other levels are ordered by. By default by alphabetical order. In our case, the “C” in “Chimpanzee” comes before the “T” in “Tiger”.

If we put the factor on the y-axis and the sleep hours on the x-axis, we can see that the values are arranged from bottom to top and not from top to bottom:

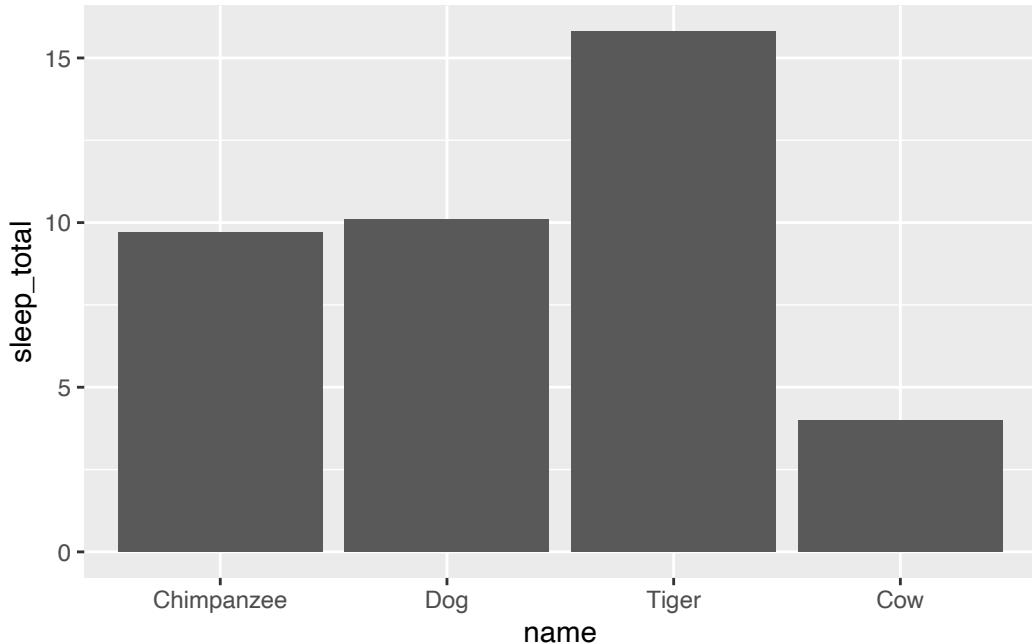
```
msleep %>%
  filter(name %in% c("Cow", "Dog", "Tiger", "Chimpanzee")) %>%
  mutate(
    name = fct_relevel(name, c("Cow", "Dog")))
) %>%
  ggplot(aes(x = sleep_total, y = name)) +
  geom_col()
```



Here is another interesting trick. Instead of placing the levels to the front, we can place them wherever we like with the `after` argument. Suppose we want to place the Cow after the Tiger:

```
msleep %>%
  filter(name %in% c("Cow", "Dog", "Tiger", "Chimpanzee")) %>%
  mutate(
    name = fct_relevel(name, "Cow", after = 3))
```

```
) %>%
ggplot(aes(x = name, y = sleep_total)) +
geom_col()
```



I wish you could use a string for the `after` argument, but you must specify a number. Your levels will be placed at the $n + 1$ position of this number. In our case the layer `cow` is placed at the 4th position (3th + 1).

9.2 How to order the levels based on how frequently each level occurs

For the next trick, we must first note that it will hardly work with our previous data set. The reason for this is that each level in our factor column occurs only once:

```
msleep %>%
  filter(name %in% c("Cow", "Dog", "Tiger", "Chimpanzee")) %>%
  count(name)
```

```
# A tibble: 4 x 2
```

```
name      n
<chr>    <int>
1 Chimpanzee   1
2 Cow          1
3 Dog          1
4 Tiger        1
```

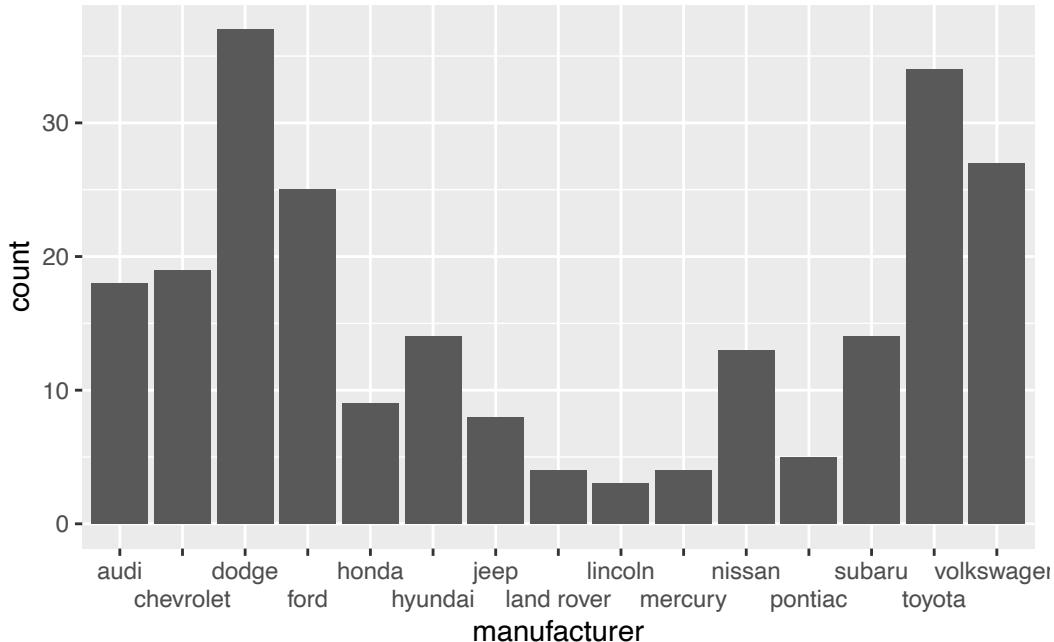
You can't really order the levels if they don't differ in frequency. So let's find a data frame where the factor levels differ. Let us take the data frame `mpg` with the `manufacturer` column:

```
mpg %>%
  count(manufacturer)

# A tibble: 15 x 2
  manufacturer     n
  <chr>       <int>
1 audi           18
2 chevrolet      19
3 dodge          37
4 ford           25
5 honda          9
6 hyundai        14
7 jeep           8
8 land rover     4
9 lincoln         3
10 mercury        4
11 nissan         13
12 pontiac        5
13 subaru         14
14 toyota          34
15 volkswagen     27
```

Now suppose we want to show how many times each manufacturer appears in the data frame:

```
mpg %>%
  ggplot(aes(x = manufacturer)) +
  geom_bar() +
  scale_x_discrete(guide = guide_axis(n.dodge = 2))
```



Again, let's first illustrate the new function with a simple example. Here again is our column manufacturer factor:

```
manufacturer_factor <- as.factor(mpg$manufacturer)
levels(manufacturer_factor)

[1] "audi"      "chevrolet" "dodge"     "ford"      "honda"
[6] "hyundai"   "jeep"      "land rover" "lincoln"   "mercury"
[11] "nissan"    "pontiac"   "subaru"   "toyota"    "volkswagen"
```

Again, you can see that they are in alphabetical order. With the function `fct_infreq` we can change the order according to how frequent each level occurs:

```
manufacturer_factor %>% fct_infreq %>%
  levels

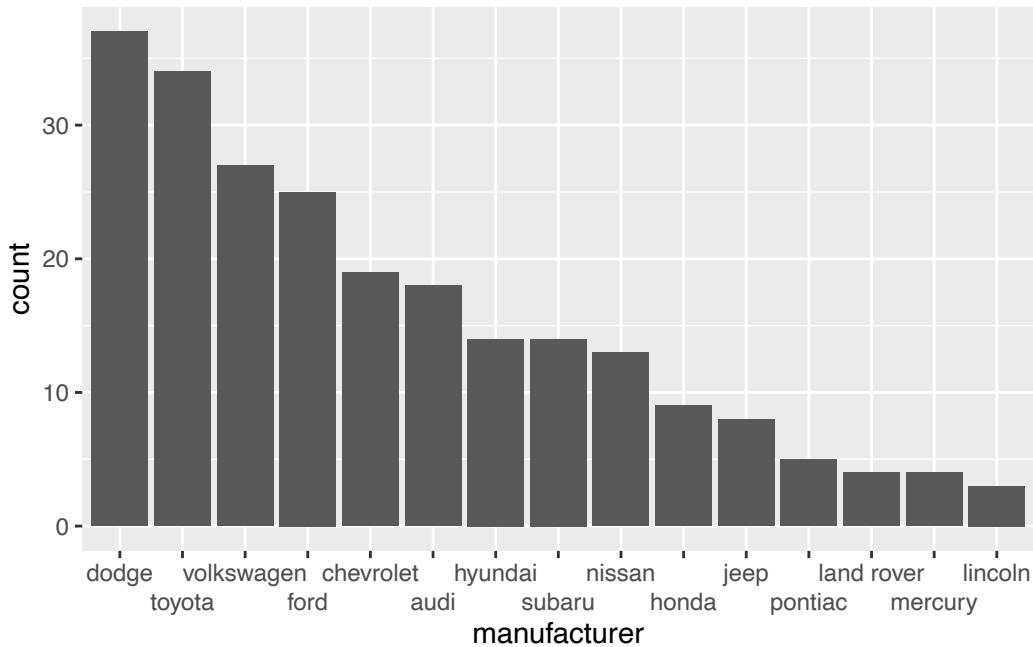
[1] "dodge"      "toyota"     "volkswagen" "ford"      "chevrolet"
[6] "audi"       "hyundai"    "subaru"     "nissan"    "honda"
[11] "jeep"       "pontiac"    "land rover"  "mercury"   "lincoln"
```

To check this, we can visualize the bar chart again:

```

mpg %>%
  mutate(manufacturer = fct_infreq(manufacturer)) %>%
  ggplot(aes(x = manufacturer)) +
  geom_bar() +
  scale_x_discrete(guide = guide_axis(n.dodge = 2))

```

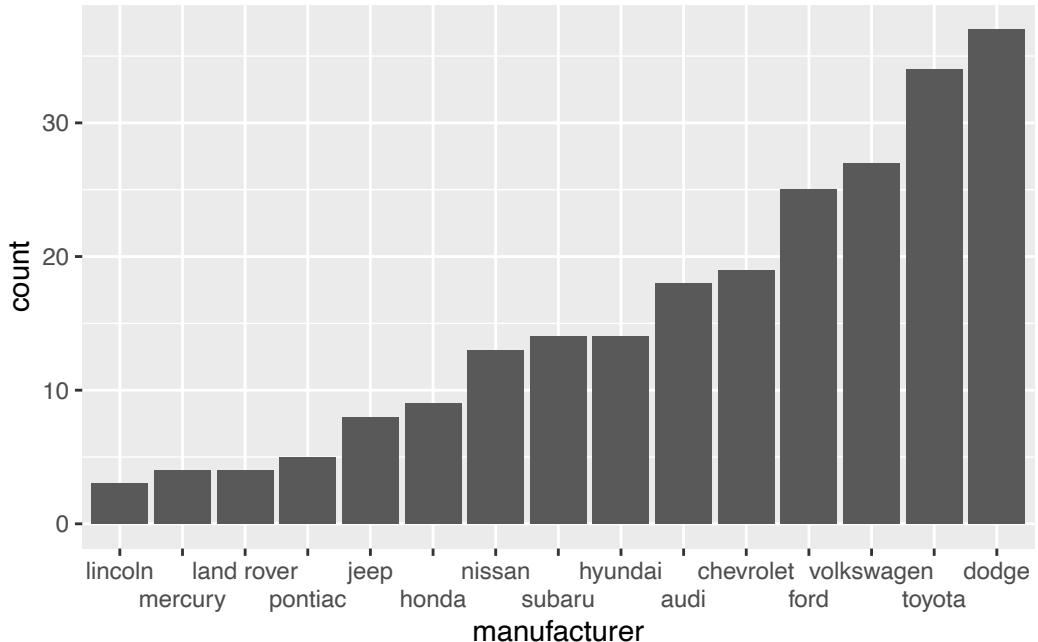


If you want to reverse the order of the levels, you can use the `fct_rev` function:

```

mpg %>%
  mutate(manufacturer = fct_infreq(manufacturer) %>% fct_rev) %>%
  ggplot(aes(x = manufacturer)) +
  geom_bar() +
  scale_x_discrete(guide = guide_axis(n.dodge = 2))

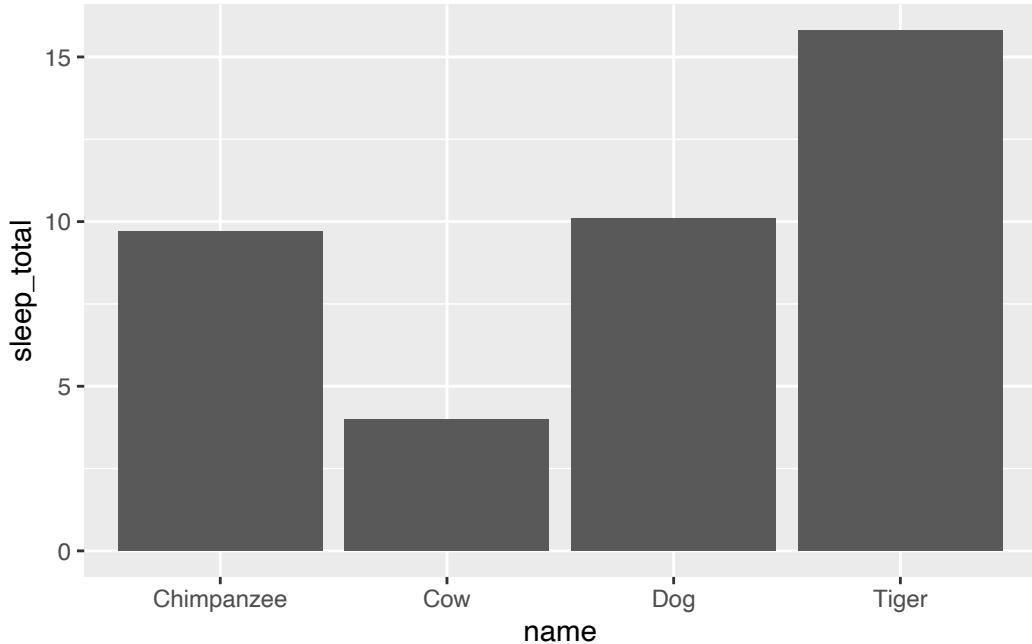
```



9.3 How to order the levels based on the values of a numeric variable

So far, we have changed the order of the levels based only on the information from the factor column. Next, let's discuss how we can order the levels of a factor column based on another numeric variable. Here is a typical use case for this: you create a bar chart with a discrete variable on the x-axis and a continuous variable on the y-axis:

```
msleep %>%
  filter(name %in% c("Cow", "Dog", "Tiger", "Chimpanzee")) %>%
  ggplot(aes(x = name, y = sleep_total)) +
  geom_col()
```



We can see that each factor level is associated with the continuous variable `sleep_total`:

```
(sleep_data <- msleep %>%
  filter(name %in% c("Cow", "Dog", "Tiger", "Chimpanzee")) %>%
  select(name, sleep_total))
```

```
# A tibble: 4 x 2
  name      sleep_total
  <chr>     <dbl>
1 Cow          4
2 Dog         10.1
3 Chimpanzee  9.7
4 Tiger        15.8
```

The function `fct_reorder` allows to order the levels based on another continuous variable. In our case `sleep_total`. Let's try it out:

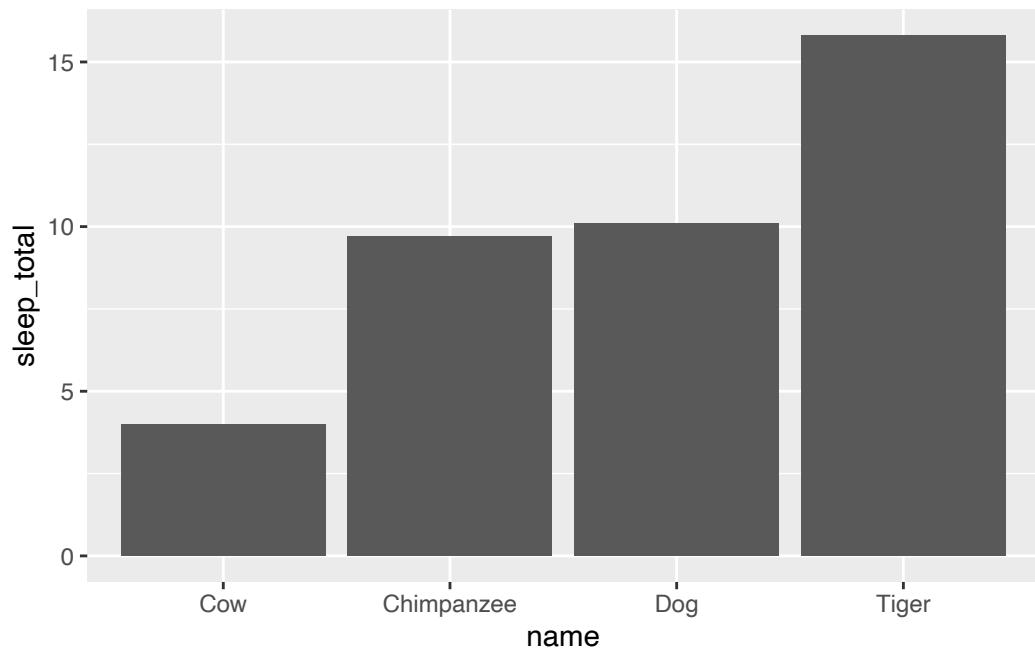
```
sleep_data %>%
  mutate(
    name = as.factor(name) %>% fct_reorder(sleep_total)
  ) %>%
```

```
pull(name)
```

```
[1] Cow      Dog      Chimpanzee Tiger  
Levels: Cow Chimpanzee Dog Tiger
```

Cows apparently have the fewest hours of sleep, followed by dogs, chimpanzees and tigers.
Let's visualize this:

```
msleep %>%  
  filter(name %in% c("Cow", "Dog", "Tiger", "Chimpanzee")) %>%  
  mutate(  
    name = as.factor(name) %>% fct_reorder(sleep_total)  
  ) %>%  
  ggplot(aes(x = name, y = sleep_total)) +  
  geom_col()
```

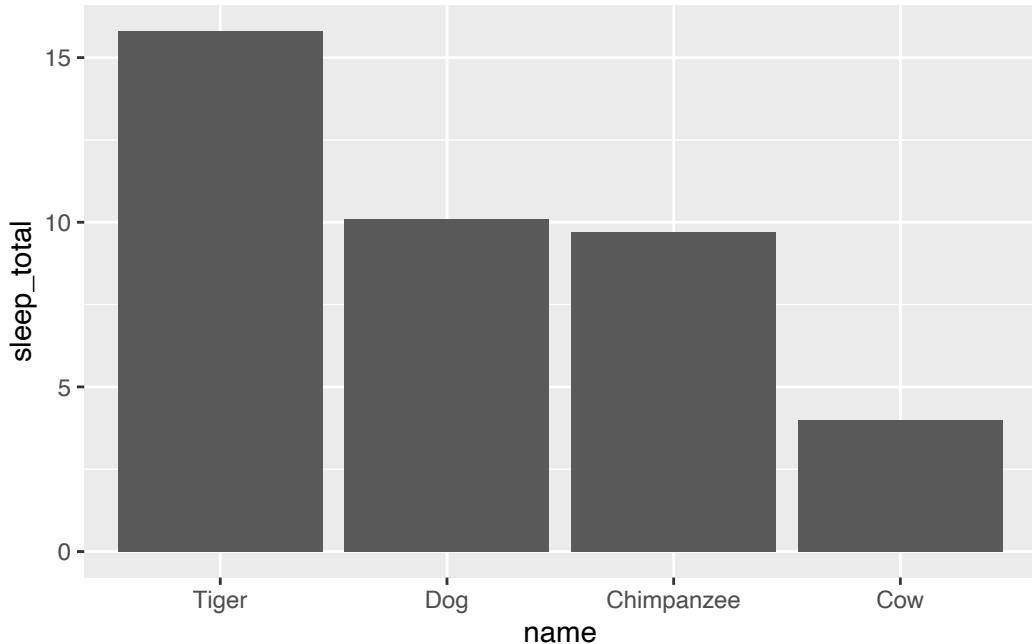


That is correct. We can reverse the order in two ways. First, by setting the `.desc` argument to `TRUE`:

```

msleep %>%
  filter(name %in% c("Cow", "Dog", "Tiger", "Chimpanzee")) %>%
  mutate(
    name = as.factor(name) %>%
      fct_reorder(sleep_total, .desc = TRUE)
  ) %>%
  ggplot(aes(x = name, y = sleep_total)) +
  geom_col()

```

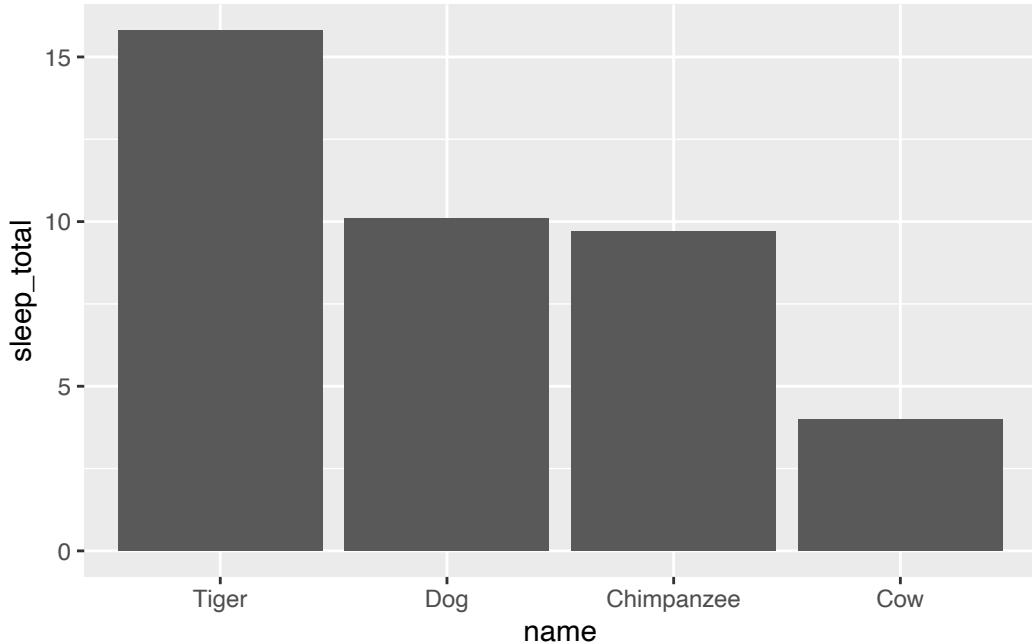


Then with the function `fct_rev`:

```

msleep %>%
  filter(name %in% c("Cow", "Dog", "Tiger", "Chimpanzee")) %>%
  mutate(
    name = as.factor(name) %>%
      fct_reorder(sleep_total) %>%
      fct_rev
  ) %>%
  ggplot(aes(x = name, y = sleep_total)) +
  geom_col()

```



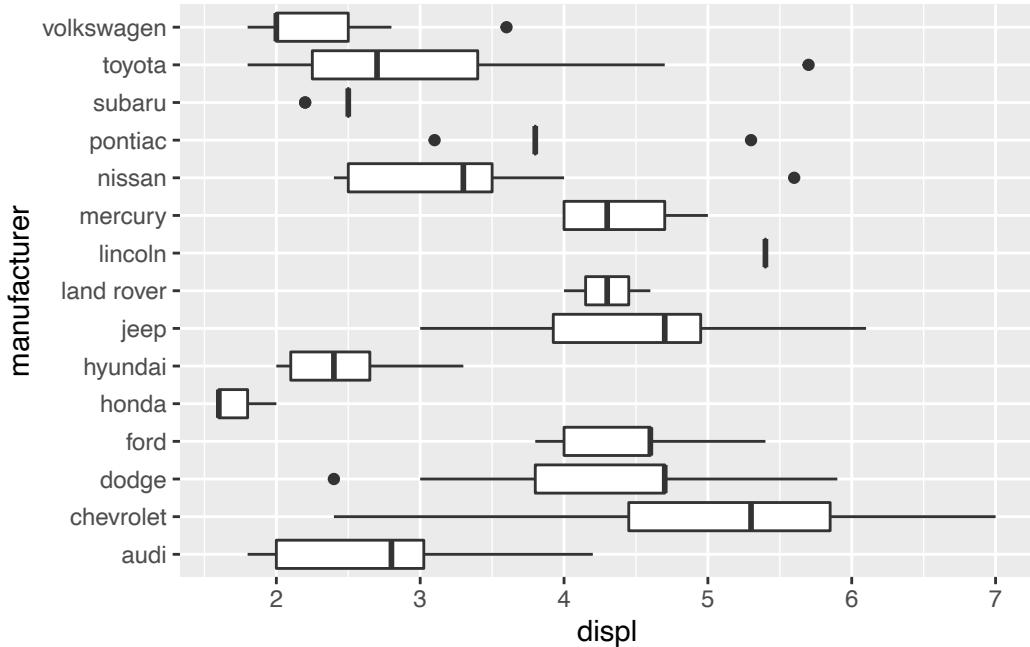
If you take a look at the documentation, you will find the `.fun` argument. The documentation says the following:

“n summary function. It should take one vector for `fct_reorder`, and two vectors for `fct_reorder2`, and return a single value.”

By default, this argument is set to the function `median`. Frankly, I didn’t understand this argument at first.

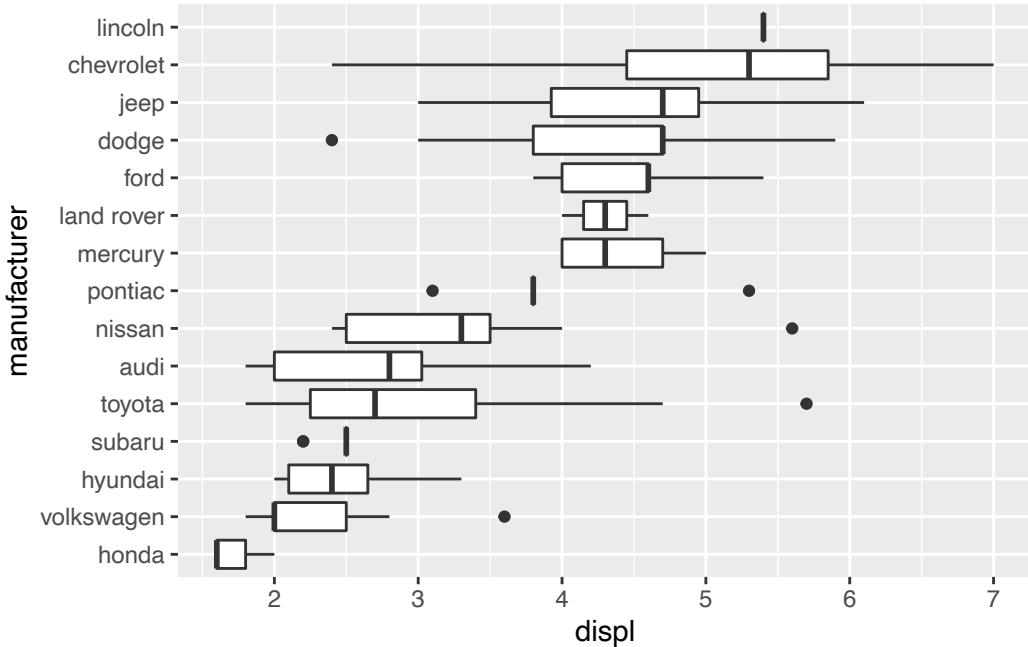
The first thing you need to know is that the `.fun` argument is useful when you have many values for each factor level. This is always the case when you create a boxplot or a violin diagram. Like this:

```
mpg %>%
  ggplot(aes(x = displ, y = manufacturer)) +
  geom_boxplot()
```



Before we explain the argument in detail, let's order the levels with the function `fct_reorder`:

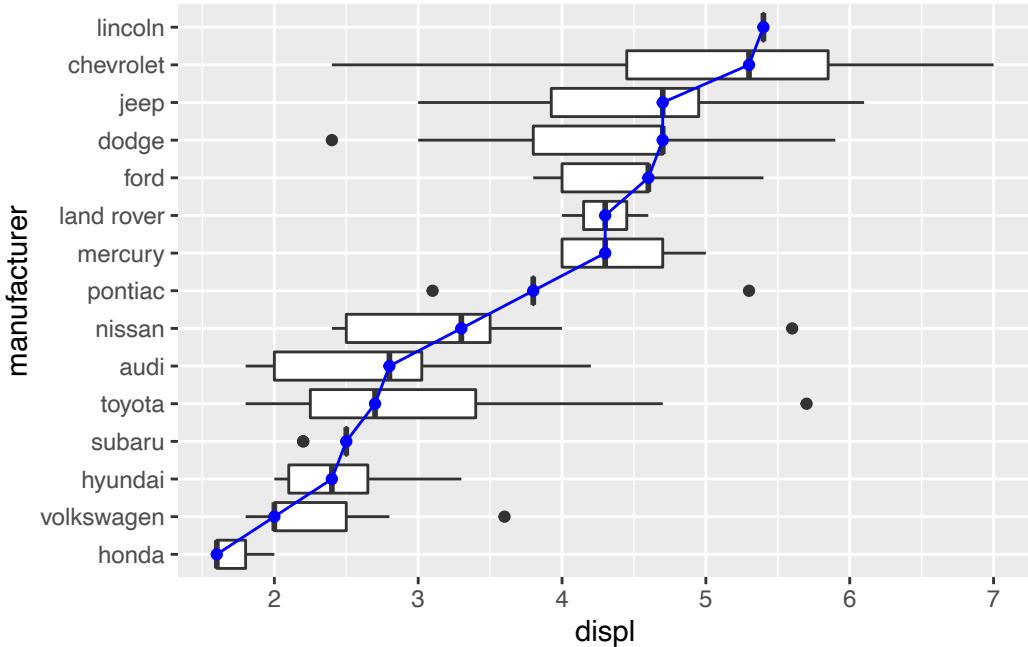
```
mpg %>%
  mutate(
    manufacturer = fct_reorder(as.factor(manufacturer),
                                displ)
  ) %>%
  ggplot(aes(x = displ, y = manufacturer)) +
  geom_boxplot()
```



Clearly, the levels have been ordered. But how? The higher the level, the larger the value of `displ`. I told you that by default the levels are ordered by the median values of the continuous variable.

We can illustrate this by adding a line chart to the plot:

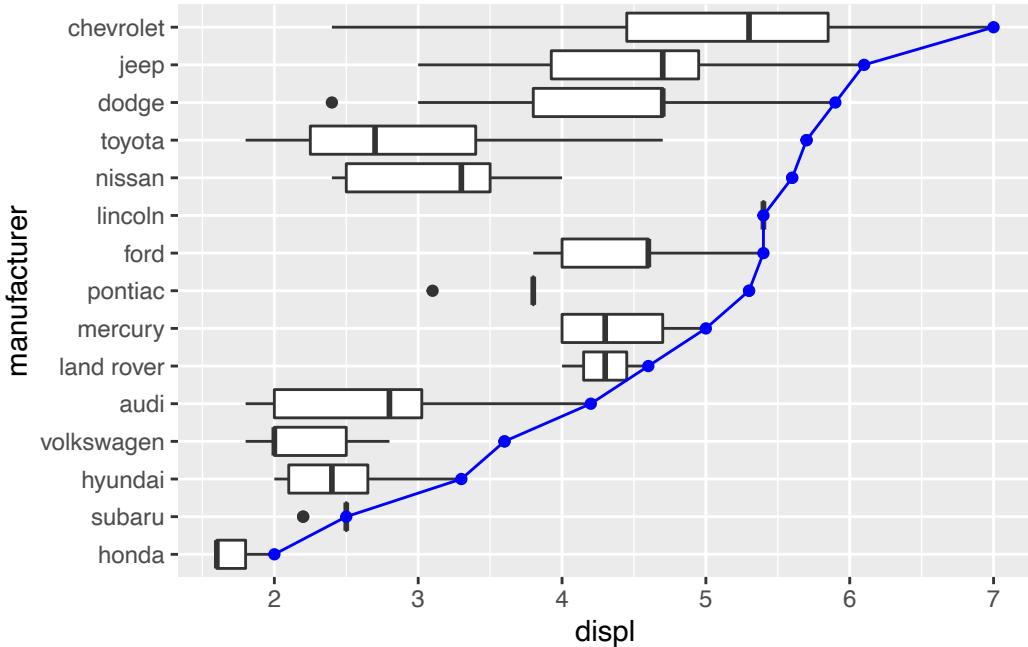
```
mpg %>%
  mutate(
    manufacturer = fct_reorder(as.factor(manufacturer),
                                displ)
  ) %>%
  ggplot(aes(x = displ, y = manufacturer)) +
  geom_boxplot() +
  stat_summary(geom = "point", fun = "median", color = "blue") +
  stat_summary(geom = "line", fun = "median", color = "blue", group = 1)
```



Some of you may know that the middle line in boxplots represents the median value. This is exactly what we see here.

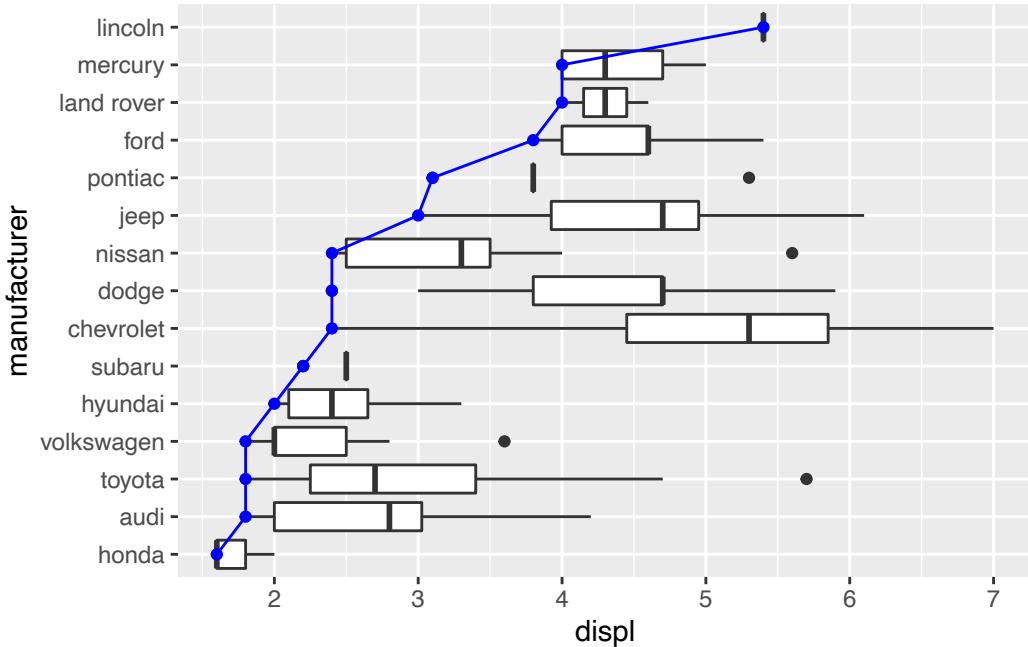
Say we want to change the order according to the maximum value of the continuous variable (see `.fun = max`):

```
mpg %>%
  mutate(
    manufacturer = fct_reorder(as.factor(manufacturer),
                                displ, .fun = max)
  ) %>%
  ggplot(aes(x = displ, y = manufacturer)) +
  geom_boxplot() +
  stat_summary(geom = "point", fun = "max", color = "blue") +
  stat_summary(geom = "line", fun = "max", color = "blue", group = 1)
```



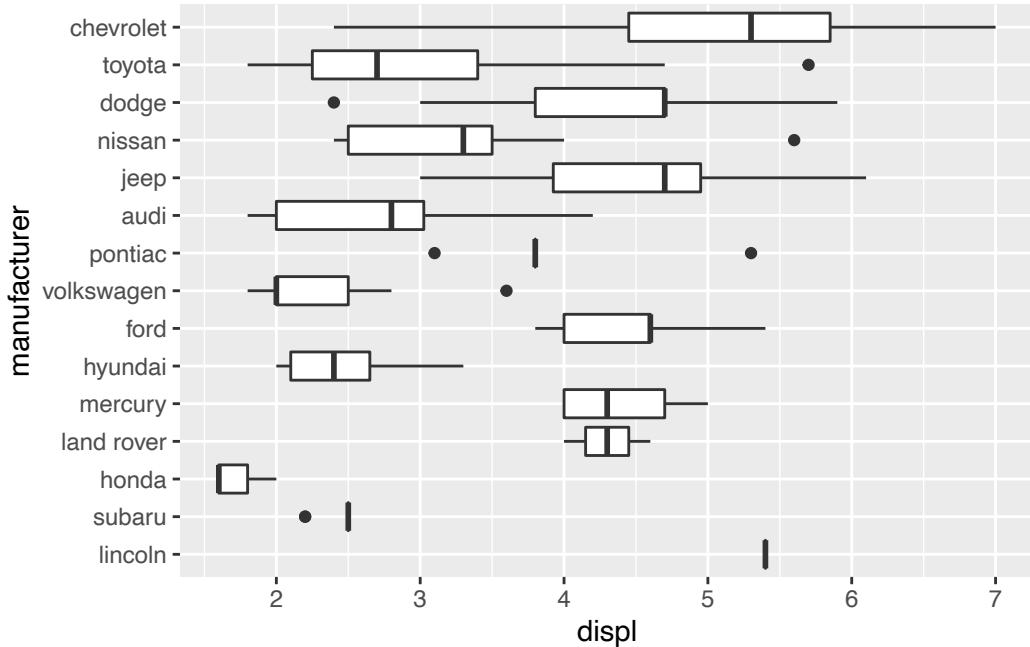
Or the minimal value:

```
mpg %>%
  mutate(
    manufacturer = fct_reorder(as.factor(manufacturer),
                                displ, .fun = min)
  ) %>%
  ggplot(aes(x = displ, y = manufacturer)) +
  geom_boxplot() +
  stat_summary(geom = "point", fun = "min", color = "blue") +
  stat_summary(geom = "line", fun = "min", color = "blue", group = 1)
```



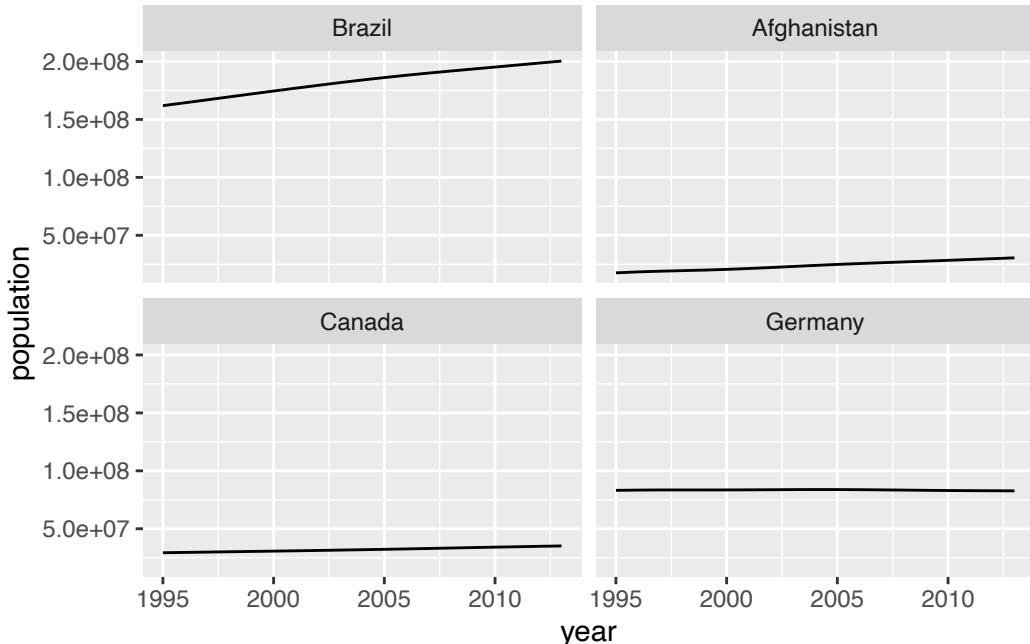
Similarly, you could order the levels by the range between the maximum and minimum values of each level:

```
mpg %>%
  mutate(
    manufacturer = fct_reorder(as.factor(manufacturer),
                                displ,
                                .fun = function(x) max(x) - min(x))
  ) %>%
  ggplot(aes(x = displ, y = manufacturer)) +
  geom_boxplot()
```



We could do something similar with facets. For example, suppose we want to rank countries by the amount of population growth from 1995 to 2013:

```
population %>%
  filter(country %in% c("Afghanistan", "Germany", "Brazil",
                        "Canada")) %>%
  mutate(country = fct_reorder(country,
                               population,
                               .fun = function(x) min(x) - max(x)))
) %>%
  ggplot(aes(x = year, y = population)) +
  geom_line() +
  facet_wrap(vars(country))
```



Obviously, Brazil had the largest increase compared to the other countries.

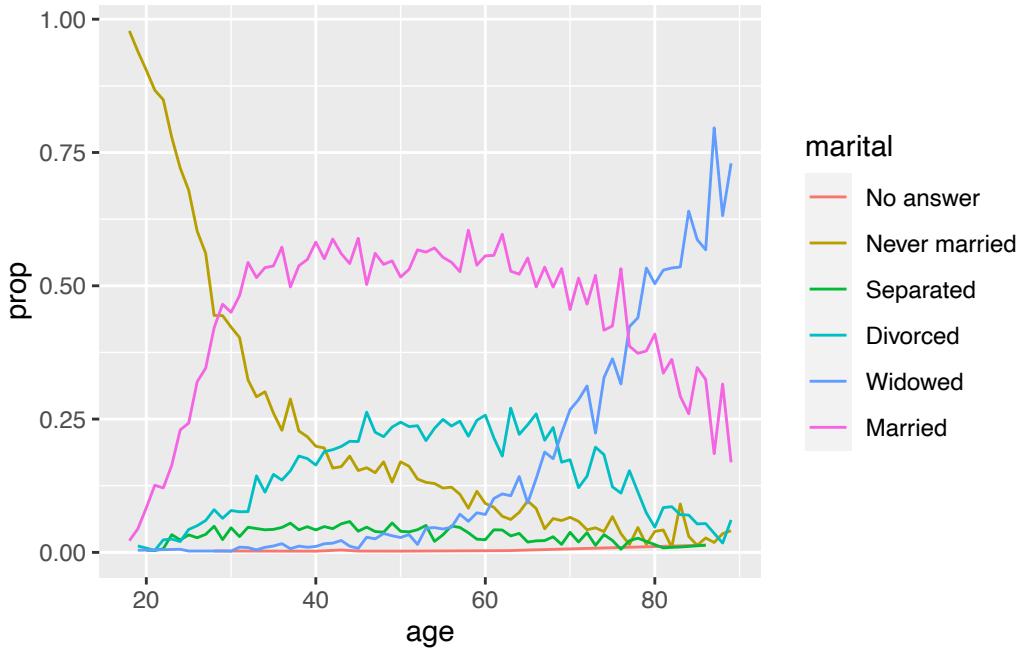
9.4 How to order levels based on the values of two numeric variables

Have you ever created a line chart and found that the end of your lines are not aligned with your legend text? Here is an example of this (which I adapted from [this stackoverflow post](#)):

```
marital_status_per_age <- gss_cat %>%
  count(age, marital) %>%
  group_by(age) %>%
  mutate(
    prop = n / sum(n)
  ) %>%
  ungroup()

marital_status_per_age %>%
  ggplot(aes(x = age, y = prop, color = marital)) +
  stat_summary(geom = "line", fun = mean)
```

Warning: Removed 6 rows containing non-finite values (stat_summary).



As you can see the blue line ends at the top but the first item in the legend is the “No answer” category. We can improve the order of the legend (aka the levels) by using the function `fct_reorder2`.

In essence, the function does the following: It finds the largest values of one variable at the largest value of another variable. In this case, we are looking for the largest value of `prop` within the largest value of `age`:

```
marital_status_per_age %>%
  group_by(marital) %>%
  slice_max(age) %>%
  ungroup() %>%
  arrange(desc(prop))
```

```
# A tibble: 6 x 4
  age marital      n    prop
  <int> <fct>     <int>  <dbl>
1    89 Widowed   108  0.730
2    89 Married    25  0.169
```

```

3   89 Divorced      9 0.0608
4   89 Never married 6 0.0405
5   86 No answer     1 0.0135
6   86 Separated     1 0.0135

```

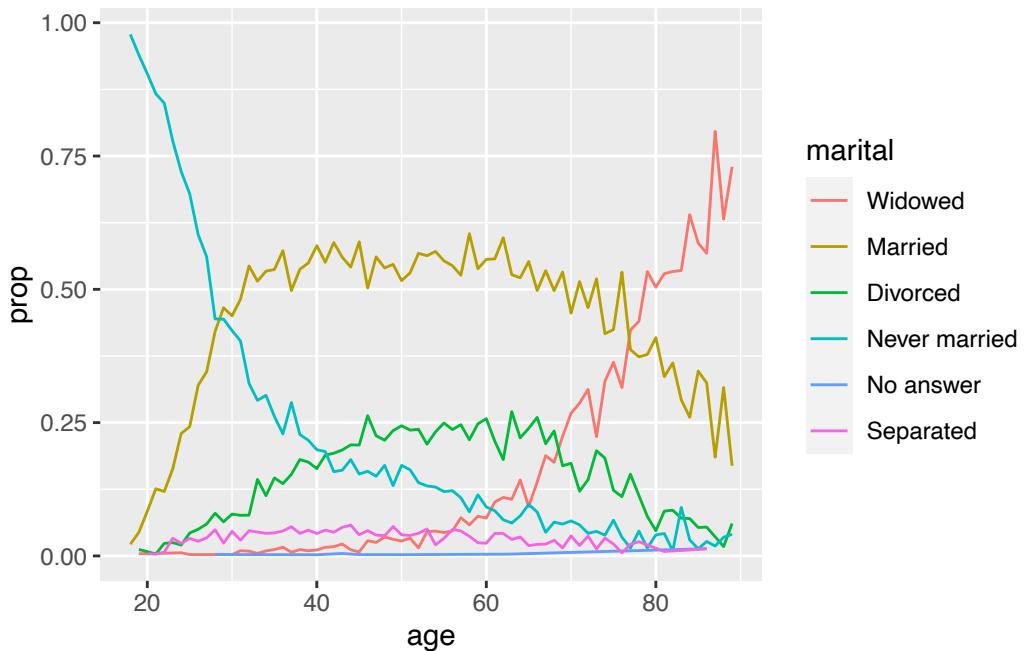
For example, the largest value of `prop` for the largest value of `age` is 0.73. This level should, therefore, be at the top of our legend. Let's see how this works:

```

marital_status_per_age %>%
  mutate(
    marital = as.factor(marital) %>%
      fct_reorder2(age, prop)
  ) %>%
  ggplot(aes(x = age, y = prop, color = marital)) +
  stat_summary(geom = "line", fun = mean)

```

Warning: Removed 6 rows containing non-finite values (stat_summary).



The first column is the reference variable (i.e., `age`). The second column determines the order of the levels.

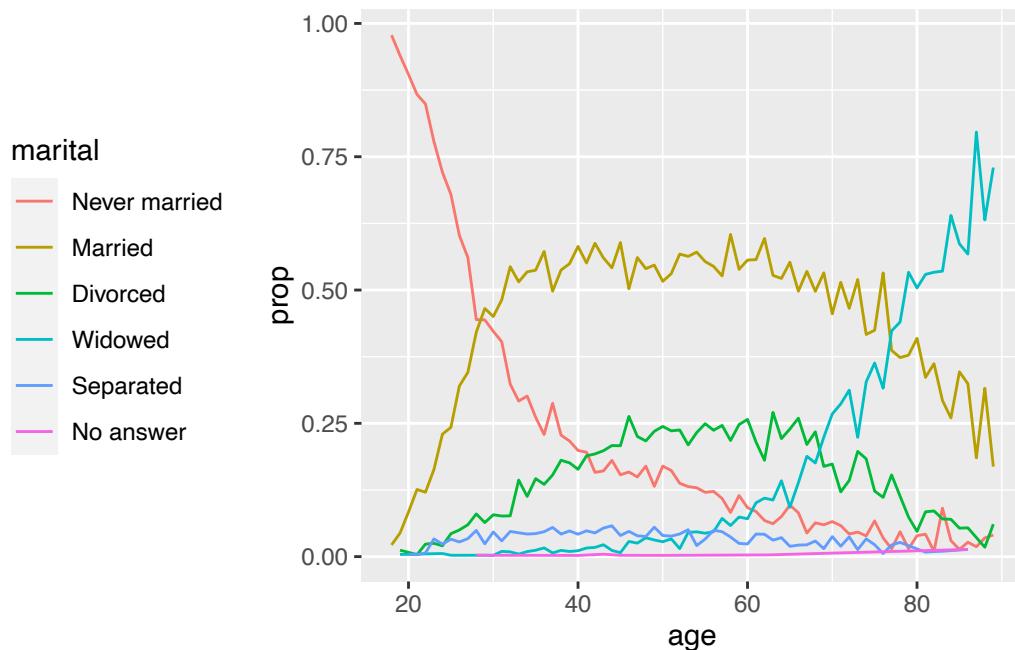
We can reverse the order by setting the `.fun` argument to `first2`:

```

marital_status_per_age %>%
  mutate(
    marital = as.factor(marital) %>%
      fct_reorder2(age, prop, .fun = first2)
  ) %>%
  ggplot(aes(x = age, y = prop, color = marital)) +
  stat_summary(geom = "line", fun = mean) +
  theme(
    legend.position = "left"
  )

```

Warning: Removed 6 rows containing non-finite values (stat_summary).



And this concludes our tour through ordering factor levels.

i Summary

Here's what you can take away from this tutorial.

- The functions `fct_relevel` and `fct_infreq` order the levels only according to the information provided by the factor column itself.

- The functions `fct_reorder` and `fct_reorder2` order levels based on other continuous variables.
- Use `fct_reorder2` if you want to align your legend text to the endpoints of your lines in a line chart.
- Use the `.fun` argument in `fct_reorder` whenever you want to order geometric objects that visualize many values (e.g. boxplots or violinplots)

10 How to apply a function across many columns

What will this tutorial cover?

In this tutorial you will learn how to apply one or more functions to many columns. You will learn to explain the structure of the function `across` and you will be able to use the function for six use cases.

Who do I have to thank?

For this tutorial, I have to thank Rebecca Barter, who wrote [a wonderful blog post about the `across` function](#). I have adapted a few of her examples for this tutorial.

One of the credos of programming is “Don’t repeat yourself”. We have seen in previous tutorials that many of us fall victim to this principle quite often. Fortunately, the tidyverse team has developed a set of functions that make it easier not to repeat ourselves. We have seen it with the `rename_with` function, which allows us to rename many columns at once. In this tutorial, we’ll take this idea further and figure out how to use the `across` function to apply a function to many columns.

We will talk about six use cases of the `across` function. These cases are tied to two dplyr functions: `summarise` and `mutate`:

- `summarise`: How to calculate summary statistics across many columns
- `summarise`: How to calculate the number of distinct values across many columns
- `mutate`: How to change the variable type across many columns
- `mutate`: How to normalize many columns
- `mutate`: How to impute values across many columns
- `mutate`: How to replace characters across many columns

Each use case will work with this general structure:

```
<DFRAME> %>%
  <DPLYR VERB>(
    across(
      .cols  = <SELECTION OF COLUMNS>,
      .fns   = <FUNCTION TO BE APPLIED TO EACH COLUMN>,
      .names = <NAME OF THE GENERATED COLUMNS>
    )
  )
```

A couple of things are important here:

- The function `across` only works inside dplyr verbs (e.g. `mutate`)
- The function has three important arguments: `.cols` stands for the column to apply a function to. You can use the tidyselect functions here; `.fns` stands for the function(s) that will be applied to these columns; `.names` is used whenever you want to change the names of the selected columns.
- The `.fns` argument takes three different values: (1) A simple function (e.g. `mean`). (2) A purrr-style lambda function (e.g. `~ mean(.x, na.rm = TRUE)`). You use these lambda functions if you need to change the arguments of a function. (3) A list of functions (e.g. `list(mean = mean, sd = sd)`). The list can also be combined with lambda functions (e.g. `list(mean = mean(.x, na.rm = TRUE), sd = sd(.x, na.rm = TRUE))`).

For the following examples, I will stick to this structure so that you can easily see the differences between the code snippets. Let's start with the combination of `across` and `summarise`.

10.1 across and summarise

10.1.1 How to calculate summary statistics across many columns

Here is a typical example of how many people calculate summary statistics with dplyr:

```
mpg %>%
  group_by(manufacturer) %>%
  summarise(
    mean_displ = mean(displ, na.rm = TRUE),
    mean_cty   = mean(cty, na.rm = TRUE),
  )
```

```
# A tibble: 15 x 3
  manufacturer mean_displ mean_cty
  <chr>          <dbl>    <dbl>
1 audi            2.54     17.6
2 chevrolet       5.06     15.0
3 dodge           4.38     13.1
4 ford            4.54     14.0
5 honda           1.71     24.4
6 hyundai         2.43     18.6
7 jeep             4.58     13.5
8 land rover      4.3      11.5
9 lincoln          5.4      11.3
10 mercury         4.4      13.2
11 nissan          3.27     18.1
12 pontiac         3.96     17.0
13 subaru          2.46     19.3
14 toyota           2.95     18.5
15 volkswagen      2.26     20.9
```

This approach works, but it is not scalable. Imagine you had to calculate the mean and standard deviation of dozens of columns.

Instead you can use the `across` function to get the same result:

```
mpg %>%
  group_by(manufacturer) %>%
  summarise(
    across(
      .cols = c("displ", "cty"),
      .fns = ~ mean(.x, na.rm = TRUE),
      .names = "mean_{.col}"
    )
  )
```

```
# A tibble: 15 x 3
  manufacturer mean_displ mean_cty
  <chr>          <dbl>    <dbl>
1 audi            2.54     17.6
2 chevrolet       5.06     15.0
3 dodge           4.38     13.1
4 ford            4.54     14.0
5 honda           1.71     24.4
```

6	hyundai	2.43	18.6
7	jeep	4.58	13.5
8	land rover	4.3	11.5
9	lincoln	5.4	11.3
10	mercury	4.4	13.2
11	nissan	3.27	18.1
12	pontiac	3.96	17
13	subaru	2.46	19.3
14	toyota	2.95	18.5
15	volkswagen	2.26	20.9

With the argument `.cols` (`.cols = c("displ", "cty")`) we told across that we want to apply a function to these two columns. With the argument `.fns` (`.fns = ~ mean(.x, na.rm = TRUE)`) we told across that we want to calculate the mean of the two columns. We also want to remove NAs from each column. Using the `.names` argument (`.names = "mean_{.col}"`), we told across that we want the summarised columns to have the following structure: Start with the string `mean_` and glue the name of the column to this string (`{.col}`).

Suppose, we would like to also calculate the standard deviation of each column. In this case, we need to provide a list to the `.fns` argument instead of a purr-style lambda function:

```
mpg %>%
  group_by(manufacturer) %>%
  summarise(
    across(
      .cols = c("displ", "cty"),
      .fns = list(mean = mean, sd = sd),
      .names = "{.fn}_{.col}"
    )
  )

# A tibble: 15 x 5
  manufacturer mean_displ sd_displ mean_cty sd_cty
  <chr>          <dbl>     <dbl>     <dbl>     <dbl>
1 audi            2.54      0.673     17.6     1.97
2 chevrolet       5.06      1.37      15        2.92
3 dodge           4.38      0.868     13.1     2.49
4 ford            4.54      0.541     14        1.91
5 honda           1.71      0.145     24.4     1.94
6 hyundai         2.43      0.365     18.6     1.50
7 jeep            4.58      1.02      13.5     2.51
8 land rover      4.3       0.258     11.5     0.577
```

9 lincoln	5.4	0	11.3	0.577
10 mercury	4.4	0.490	13.2	0.5
11 nissan	3.27	0.864	18.1	3.43
12 pontiac	3.96	0.808	17	1
13 subaru	2.46	0.109	19.3	0.914
14 toyota	2.95	0.931	18.5	4.05
15 volkswagen	2.26	0.443	20.9	4.56

Two things have changed. First, we created a list with two functions (`list(mean = mean, sd = sd)`). As I told you at the beginning of this tutorial, you can also combine the list with purr-style lambda functions: `list(mean = ~ mean(.x, na.rm = TRUE), sd = ~ sd(.x, na.rm = TRUE))`). Second, we changed the string of the `.names` argument. Instead of `mean_{.col}` we use the string `{.fn}_{.col}`. `{.fn}` and `{.col}` are special symbols and stand for the function and the column. Since we have more than one function, we need to have the flexibility to create column names that combine the name of the column with the name of the function applied to the column.

Once we have this structure, we can add as many columns and functions as we need:

```
mpg %>%
  group_by(manufacturer) %>%
  summarise(
    across(
      .cols = where(is.numeric),
      .fns = list(mean = mean, sd = sd,
                  median = median),
      .names = "{.fn}_{.col}"
    )
  ) %>%
  glimpse()
```

```
Rows: 15
Columns: 16
$ manufacturer <chr> "audi", "chevrolet", "dodge", "ford", "honda", "hyundai", ~
$ mean_displ   <dbl> 2.544444, 5.063158, 4.378378, 4.536000, 1.711111, 2.42857~
$ sd_displ     <dbl> 0.6732032, 1.3704057, 0.8679910, 0.5407402, 0.1452966, 0.~
$ median_displ <dbl> 2.8, 5.3, 4.7, 4.6, 1.6, 2.4, 4.7, 4.3, 5.4, 4.3, 3.3, 3.~
$ mean_year    <dbl> 2003.500, 2004.684, 2004.108, 2002.600, 2003.000, 2004.14~
$ sd_year       <dbl> 4.630462, 4.460352, 4.520225, 4.500000, 4.743416, 4.62197~
$ median_year   <dbl> 2003.5, 2008.0, 2008.0, 1999.0, 1999.0, 2008.0, 2008.0, 2~
$ mean_cyl     <dbl> 5.222222, 7.263158, 7.081081, 7.200000, 4.000000, 4.85714~
$ sd_cyl        <dbl> 1.2153700, 1.3679711, 1.1150082, 1.0000000, 0.0000000, 1.~
```

```

$ median_cyl <dbl> 6, 8, 8, 8, 4, 4, 8, 8, 8, 7, 6, 6, 4, 4, 4
$ mean_cty   <dbl> 17.61111, 15.00000, 13.13514, 14.00000, 24.44444, 18.6428-
$ sd_cty     <dbl> 1.9745108, 2.9249881, 2.4850907, 1.9148542, 1.9436506, 1.-
$ median_cty <dbl> 17.5, 15.0, 13.0, 14.0, 24.0, 18.5, 14.0, 11.5, 11.0, 13.-
$ mean_hwy   <dbl> 26.44444, 21.89474, 17.94595, 19.36000, 32.55556, 26.8571-
$ sd_hwy     <dbl> 2.175322, 5.108759, 3.574182, 3.327662, 2.554952, 2.17881-
$ median_hwy <dbl> 26.0, 23.0, 17.0, 18.0, 32.0, 26.5, 18.5, 16.5, 17.0, 18.-

```

See how we used a tidyselect function instead of a vector of column names? This gives us tremendous flexibility. Also note that we can easily compute four different summary statistics once we have a list.

Before we continue with the next use case, I would like to show you an example that the dplyr developers advise against. What I have not told you so far is that the function `across` can also take additional arguments. For example the `na.rm` argument of the `mean` function:

```

mpg %>%
  group_by(manufacturer) %>%
  summarise(
    across(
      .cols = c("displ", "cty"),
      .fns = mean,
      .names = "mean_{.col}",
      na.rm = TRUE
    )
  )

```

```

Warning: There was 1 warning in `summarise()` .
i In argument: `across(...)` .
i In group 1: `manufacturer = "audi"` .
Caused by warning:
! The `...` argument of `across()` is deprecated as of dplyr 1.1.0.
Supply arguments directly to ` .fns` through an anonymous function instead.

```

```

# Previously
across(a:b, mean, na.rm = TRUE)

# Now
across(a:b, \((x) mean(x, na.rm = TRUE))
```

A tibble: 15 x 3
 manufacturer mean_displ mean_cty

	<chr>	<dbl>	<dbl>
1	audi	2.54	17.6
2	chevrolet	5.06	15
3	dodge	4.38	13.1
4	ford	4.54	14
5	honda	1.71	24.4
6	hyundai	2.43	18.6
7	jeep	4.58	13.5
8	land rover	4.3	11.5
9	lincoln	5.4	11.3
10	mercury	4.4	13.2
11	nissan	3.27	18.1
12	pontiac	3.96	17
13	subaru	2.46	19.3
14	toyota	2.95	18.5
15	volkswagen	2.26	20.9

There are two reasons not to do this: First, it causes problems in timing the evaluation. In other words, it can lead to errors. Second, it decouples the arguments from the functions to which they are applied. The best thing you can do is not to do this.

10.1.2 How to calculate the number of distinct values across many columns

This tip comes from [Rebecca Barter](#). First of all, you can be quite creative with `across` and `summarise`, because you can calculate any summary statistic for many columns. The mean, standard deviation or median are just obvious examples. Making the problem smaller and easier to digest, you might simply ask yourself, “What summary statistics can I compute from a vector?” One such statistic is the number of distinct values in a vector: How many people do I have? From how many states do these people come from? How many manufacturers are in the data?

```
mpg %>%
  summarise(
    across(
      .cols = everything(),
      .fns  = n_distinct
    )
  )

# A tibble: 1 x 11
  manufacturer model displ  year   cyl trans   drv   cty   hwy     fl class
  <fct>        <fct>  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
```

```

<int> <int> <int> <int> <int> <int> <int> <int> <int> <int>
1      15     38     35     2      4     10     3     21     27     5     7

```

A look at the results shows that the data includes 15 car manufacturers and 38 different car models.

10.2 across and mutate

10.2.1 How to change the variable type across many columns

Suppose some of your character columns need to be transferred to a factor. Let's first see how this works with `across` and `mutate` and then go through the example in more detail:

```

mpg %>%
  mutate(
    across(
      .cols = where(is.character),
      .fns  = as_factor
    )
  ) %>%
  select(where(is.factor) | where(is.character)) %>%
  glimpse()

```

```

Rows: 234
Columns: 6
$ manufacturer <fct> audi, audi, audi, audi, audi, audi, audi, audi, audi,
$ model           <fct> a4, a4, a4, a4, a4, a4 quattro, a4 quattro, a4 qu-
$ trans            <fct> auto(15), manual(m5), manual(m6), auto(av), auto(15), man-
$ drv              <fct> f, f, f, f, f, f, 4, 4, 4, 4, 4, 4, 4, 4, 4, r, ~
$ fl               <fct> p, r, ~
$ class             <fct> compact, compact, compact, compact, compact, com-

```

With `mutate` you normally specify the new column to be created or overwritten. With `across` you don't have to do that. Without specifying the `.names` argument, the names of your columns remain the same, you just apply the function(s) to those columns. In this example, we applied the function `as_factor` to each character column. We could have also changed the names of the columns and created new columns instead:

```

mpg %>%
  mutate(
    across(
      .cols = where(is.character),
      .fns = as_factor,
      .names = "{.col}_as_factor"
    )
  ) %>%
  select(where(is.factor) | where(is.character)) %>%
  glimpse()

```

```

Rows: 234
Columns: 12
$ manufacturer_as_factor <fct> audi, audi, audi, audi, audi, audi, audi, ~
$ model_as_factor         <fct> a4, a4, a4, a4, a4, a4 quattro, a4 quat~
$ trans_as_factor          <fct> auto(15), manual(m5), manual(m6), auto(av), aut~
$ drv_as_factor            <fct> f, f, f, f, f, f, 4, 4, 4, 4, 4, 4, 4, ~
$ fl_as_factor             <fct> p, ~
$ class_as_factor          <fct> compact, compact, compact, compact, compact, co~
$ manufacturer              <chr> "audi", "audi", "audi", "audi", "audi", "audi", ~
$ model                      <chr> "a4", "a4", "a4", "a4", "a4", "a4", "a4", "a4 q~
$ trans                      <chr> "auto(15)", "manual(m5)", "manual(m6)", "auto(a~
$ drv                         <chr> "f", "f", "f", "f", "f", "f", "4", "4", "4", "4~
$ fl                           <chr> "p", "p", "p", "p", "p", "p", "p", "p", "p", "p~
$ class                        <chr> "compact", "compact", "compact", "compact", "co~

```

10.2.2 How to normalize many columns

Statisticians and scientists often need to normalize their data. Suppose you want to normalize your data so that each column has a mean of 0 and a standard deviation of 1:

```

scaled_columns <- mpg %>%
  transmute(
    across(
      .cols = where(is.numeric),
      .fns = scale,
    )
  )

scaled_columns %>%

```

```
glimpse()
```

```
Rows: 234
Columns: 5
$ displ <dbl[,1]> <matrix[26 x 1]>
$ year  <dbl[,1]> <matrix[26 x 1]>
$ cyl   <dbl[,1]> <matrix[26 x 1]>
$ cty   <dbl[,1]> <matrix[26 x 1]>
$ hwy   <dbl[,1]> <matrix[26 x 1]>
```

Two things: First, I used the function `transmute` to keep only the columns to which the function `scale` was applied. Second, the result is not a set of vectors, but matrices. Let's have a look at one of these columns:

```
scaled_columns$displ %>% head
```

```
[,1]
[1,] -1.2939999
[2,] -1.2939999
[3,] -1.1391962
[4,] -1.1391962
[5,] -0.5199816
[6,] -0.5199816
```

This is a matrix with one column and many rows. To return only the vector of this column, we can index it:

```
scaled_columns$displ[,1] %>% head
```

```
[1] -1.2939999 -1.2939999 -1.1391962 -1.1391962 -0.5199816 -0.5199816
```

The value before the comma indicates the rows we want to select, the value after the comma indicates the columns. To fix our problem, we need to add this syntax to our `scale` function:

```
mpg %>%
  transmute(
    across(
      .cols = where(is.numeric),
      .fns  = ~ scale(.)[,1],
```

```

    .unpack = TRUE
  )
) %>%
glimpse()

Rows: 234
Columns: 5
$ displ <dbl> -1.2939999, -1.2939999, -1.1391962, -1.1391962, -0.5199816, -0.5~
$ year  <dbl> -0.997861, -0.997861, 0.997861, 0.997861, -0.997861, -0.997861, ~
$ cyl   <dbl> -1.1721058, -1.1721058, -1.1721058, -1.1721058, 0.0689474, 0.068~
$ cty   <dbl> 0.26810155, 0.97299777, 0.73803236, 0.97299777, -0.20182926, 0.2~
$ hwy   <dbl> 0.93369639, 0.93369639, 1.26956872, 1.10163255, 0.42988788, 0.42~

scaled_columns <- mpg %>%
  transmute(
    across(
      .cols = where(is.numeric),
      .fns   = ~ scale(.)[,1],
      .unpack = TRUE
    )
  )

scaled_columns %>%
  glimpse()

Rows: 234
Columns: 5
$ displ <dbl> -1.2939999, -1.2939999, -1.1391962, -1.1391962, -0.5199816, -0.5~
$ year  <dbl> -0.997861, -0.997861, 0.997861, 0.997861, -0.997861, -0.997861, ~
$ cyl   <dbl> -1.1721058, -1.1721058, -1.1721058, -1.1721058, 0.0689474, 0.068~
$ cty   <dbl> 0.26810155, 0.97299777, 0.73803236, 0.97299777, -0.20182926, 0.2~
$ hwy   <dbl> 0.93369639, 0.93369639, 1.26956872, 1.10163255, 0.42988788, 0.42~

```

We can prove that the columns were scaled ($\text{mean} = 1$, $\text{sd} = 0$) with the `across summarise`:

```

scaled_columns %>%
  summarise(
    across(
      .cols = everything(),
      .fns   = list(mean = mean, sd = sd)

```

```
)  
) %>%  
glimpse()
```

```
Rows: 1  
Columns: 10  
$ displ_mean <dbl> 1.56392e-15  
$ displ_sd   <dbl> 1  
$ year_mean  <dbl> 5.693451e-18  
$ year_sd    <dbl> 1  
$ cyl_mean   <dbl> -1.684313e-17  
$ cyl_sd     <dbl> 1  
$ cty_mean   <dbl> 2.095605e-16  
$ cty_sd     <dbl> 1  
$ hwy_mean   <dbl> -2.097088e-16  
$ hwy_sd     <dbl> 1
```

10.2.3 How to impute values across many columns

Again, I have to thank [Rebecca Barter and her good blog post on the across function](#) for this trick. When we impute missing values, we replace them with substituted values. I am not an expert in this field, but I can show you how the method of imputation with `across` and `mutate` might work.

Suppose this is your data frame:

```
(dframe <- tibble(  
  group = c("a", "a", "a", "b", "b", "b"),  
  x      = c(3, 5, 4, NA, 4, 8),  
  y      = c(2, NA, 3, 1, 9, 7)  
))  
  
# A tibble: 6 x 3  
#>   group     x     y  
#>   <chr> <dbl> <dbl>  
#> 1 a         3     2  
#> 2 a         5     NA  
#> 3 a         4     3  
#> 4 b        NA     1  
#> 5 b         4     9  
#> 6 b         8     7
```

You have two columns and in both columns you have a missing value. You want to replace each missing value with the mean value of the respective column:

```
dframe %>%
  mutate(
    across(
      .cols = c(x, y), # or everything()
      .fns  = ~ ifelse(test = is.na(.),
                      yes  = mean(., na.rm = TRUE),
                      no   = .)
    )
  )

# A tibble: 6 x 3
  group     x     y
  <chr> <dbl> <dbl>
1 a         3     2
2 a         5     4.4
3 a         4     3
4 b         4.8   1
5 b         4     9
6 b         8     7
```

For each value in each column, we test if the value is an `NA`. If it is, we replace this value with the value of the column, if it is a real number, we keep it. We could just as well have used the vectorized if function `case_when`:

```
dframe %>%
  mutate(
    across(
      .cols = c(x, y), # or everything()
      .fns  = ~ case_when(
        is.na(.) ~ mean(., na.rm = TRUE),
        TRUE ~ .
      )
    )
  )

# A tibble: 6 x 3
  group     x     y
  <chr> <dbl> <dbl>
```

```

1 a      3      2
2 a      5      4.4
3 a      4      3
4 b      4.8    1
5 b      4      9
6 b      8      7

```

You can also impute the values within groups:

```

dframe %>%
  group_by(group) %>%
  mutate(
    across(
      .cols = c(x, y), # or everything()
      .fns  = ~ case_when(
        is.na(.) ~ mean(., na.rm = TRUE),
        TRUE ~ .
      )
    )
  ) %>%
  ungroup()

# A tibble: 6 x 3
  group     x     y
  <chr> <dbl> <dbl>
1 a         3     2
2 a         5     2.5
3 a         4     3
4 b         6     1
5 b         4     9
6 b         8     7

```

For example, the new value 6 from the column `x` is the mean of the values within the group “b” $((8 + 4)/2)$.

10.2.4 How to replace characters across many columns

Suppose you have the same typo in many columns:

```

typo_dframe <- tribble(
  ~pre_test,    ~post_test,
  "goud"       , "good",
  "medium"     , "good",
  "metium"     , "metium",
  "bad"        , "goud"
)

```

“goud” should be “good” and “metium” should be “medium”. Again, we can combine `mutate` with `across` to correct these typos across both columns:

```

(typo_corrected <- typo_dframe %>%
  mutate(
    across(
      .cols = everything(),
      .fns  = ~ case_when(
        str_detect(., "goud") ~ str_replace(., "goud", "good"),
        str_detect(., "metium") ~ str_replace(., "metium", "medium"),
        TRUE ~ .
      )
    )
  )))

```

	pre_test	post_test
1	good	good
2	medium	good
3	medium	medium
4	bad	good

Summary

Here's what you can take away from this tutorial.

- The function `across` has three important arguments: `.cols`, `.fns`, and `.names`
- `across` must be used inside dplyr functions.
- The most common dplyr functions that use `across` are `mutate` and `summarise`.
- `across` keeps the column names by default, you can change them with `.names`.

Part IV

Improve working with rows

11 How to filter rows based on multiple columns

What will this tutorial cover?

In this tutorial you will learn how to use the `if_any` and `if_all` functions to filter rows based on conditions across multiple columns. We will deal with three use cases: Filtering rows based on specific conditions, filtering rows with missing values and creating new columns with `case_when` and `if_any` / `if_all`.

Who do I have to thank?

I would like to thank the authors of the book [R for Epidemiology](#), from whom I took the example of showing `filter` only with logical vectors. I would also like to thank [Romain Francois](#) for his great Tidyverse blog post about the `if_any` and `if_all` functions.

Suppose you want to filter all rows from your data frame that contain a missing value. If you have only a few columns, you can solve this problem as follows:

```
df <- tibble(  
  a = c(1, 2, 3),  
  b = c(NA, 4, 8),  
  c = c(1, 4, 1)  
)  
  
df %>%  
  filter(!is.na(a) & !is.na(b) & !is.na(c))  
  
# A tibble: 2 x 3  
  a     b     c  
  <dbl> <dbl> <dbl>  
1     2     4     4  
2     3     8     1
```

The function `filter` basically says the following: Find the rows where neither a, nor b, nor c

is a missing value. To accomplish this, we repeat the code `!is.na` three times. It doesn't take much to imagine how error-prone this approach would be when you have dozens of rows.

To scale the solution to this problem, we need a way to automatically check a condition across multiple columns. It's no surprise that I say `across` here, because the functions we'll learn in a minute are basically a derivative of the `across` function from the previous tutorial. The two functions you will get to know in this tutorial are `if_any` and `if_all`.

What both functions have in common is that they produce TRUE or FALSE values. Let us have another look at `filter`. `filter` basically works with logical vectors. For example, suppose we want to remove the first and third rows of your data frame with logical values only:

```
df %>%
  filter(c(FALSE, TRUE, FALSE))

# A tibble: 1 x 3
  a     b     c
  <dbl> <dbl> <dbl>
1     2     4     4
```

When working with `filter`, these logical values are usually generated by checking a condition for a single column:

```
df %>%
  filter(a == 2)

# A tibble: 1 x 3
  a     b     c
  <dbl> <dbl> <dbl>
1     2     4     4
```

The trick with `if_any` and `if_all` is that they check a condition across multiple columns and return a TRUE or FALSE value for each row. Here is the difference between them:

- `if_any` indicates whether one of the selected columns fulfills a condition
- `if_all` indicates whether all selected columns satisfy a condition

The structure of these two functions is very similar to the `across` function. The following structure applies to both `if_any` and `if_all`:

```
<DFRAME> %>%
  filter(
    if_any(
      .cols = <SELECTION OF COLUMNS>,
      .fns  = <CONDITION TO BE CHECKED FOR EACH COLUMN>,
    )
  )
)
```

Both functions [were introduced in dplyr in February 2021](#). One reason for their introduction is that `across` was not feasible with `filter`.

Next, we will dive into three use cases of `if_any` and `if_all`.

11.1 How to filter rows based on a condition across multiple columns

Suppose you are working with the `billboard` data frame, which contains the rankings of songs over a period of 76 weeks. The columns “wk1” to “wk76” contain the rankings for each week. Let’s further assume that you want to filter out the songs that made it to #1 for at least one week. Here is how you would do this with `if_any`:

```
billboard %>%
  filter(
    if_any(
      .cols = contains("wk"),
      .fns  = ~ . == 1
    )
  )

# A tibble: 17 x 79
  artist track date.ent~1   wk1   wk2   wk3   wk4   wk5   wk6   wk7   wk8   wk9
  <chr>  <chr> <date>     <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1 Aaliy~ Try ~ 2000-03-18  59    53    38    28    21    18    16    14    12
2 Aguil~ Come~ 2000-08-05  57    47    45    29    23    18    11    9     9
3 Aguil~ What~ 1999-11-27  71    51    28    18    13    13    11    1     1
4 Carey~ Than~ 1999-12-11  82    68    50    50    41    37    26    22    22
5 Creed  With~ 2000-05-13  84    78    76    74    70    68    74    75    69
6 Desti~ Inde~ 2000-09-23  78    63    49    33    23    15    7     5     1
7 Desti~ Say ~ 1999-12-25  83    83    44    38    16    13    16    16    16
8 Igles~ Be W~ 2000-04-01  63    45    34    23    17    12    9     8     8
```

```

9 Janet Does~ 2000-06-17    59    52    43    30    29    22    15    10    10
10 Lones~ Amaz~ 1999-06-05   81    54    44    39    38    33    29    29    32
11 Madon~ Music 2000-08-12   41    23    18    14     2     1     1     1     1
12 N'Sync It's~ 2000-05-06   82    70    51    39    26    19    15     9     7
13 Santa~ Mari~ 2000-02-12   15     8     6     5     2     3     2     2     1
14 Savag~ I Kn~ 1999-10-23   71    48    43    31    20    13     7     6     4
15 Sisqo Inco~ 2000-06-24   77    66    61    61    61    55     2     1     1
16 Verti~ Ever~ 2000-01-22   70    61    53    46    40    33    31    26    22
17 match~ Bent  2000-04-29   60    37    29    24    22    21    18    16    13
# ... with 67 more variables: wk10 <dbl>, wk11 <dbl>, wk12 <dbl>, wk13 <dbl>,
#   wk14 <dbl>, wk15 <dbl>, wk16 <dbl>, wk17 <dbl>, wk18 <dbl>, wk19 <dbl>,
#   wk20 <dbl>, wk21 <dbl>, wk22 <dbl>, wk23 <dbl>, wk24 <dbl>, wk25 <dbl>,
#   wk26 <dbl>, wk27 <dbl>, wk28 <dbl>, wk29 <dbl>, wk30 <dbl>, wk31 <dbl>,
#   wk32 <dbl>, wk33 <dbl>, wk34 <dbl>, wk35 <dbl>, wk36 <dbl>, wk37 <dbl>,
#   wk38 <dbl>, wk39 <dbl>, wk40 <dbl>, wk41 <dbl>, wk42 <dbl>, wk43 <dbl>,
#   wk44 <dbl>, wk45 <dbl>, wk46 <dbl>, wk47 <dbl>, wk48 <dbl>, wk49 <dbl>, ...

```

A total of 17 songs made it to #1. If you look at the `.cols` argument again, you will notice that, similar to `across`, we can use tidyselect functions to select columns for which we want to check the condition.

Let's swap `if_any` with `if_all` and see what happens:

```

billboard %>%
  filter(
    if_all(
      .cols = contains("wk"),
      .fns  = ~ . == 1
    )
  )

# A tibble: 0 x 79
# ... with 79 variables: artist <chr>, track <chr>, date.entered <date>,
#   wk1 <dbl>, wk2 <dbl>, wk3 <dbl>, wk4 <dbl>, wk5 <dbl>, wk6 <dbl>,
#   wk7 <dbl>, wk8 <dbl>, wk9 <dbl>, wk10 <dbl>, wk11 <dbl>, wk12 <dbl>,
#   wk13 <dbl>, wk14 <dbl>, wk15 <dbl>, wk16 <dbl>, wk17 <dbl>, wk18 <dbl>,
#   wk19 <dbl>, wk20 <dbl>, wk21 <dbl>, wk22 <dbl>, wk23 <dbl>, wk24 <dbl>,
#   wk25 <dbl>, wk26 <dbl>, wk27 <dbl>, wk28 <dbl>, wk29 <dbl>, wk30 <dbl>,
#   wk31 <dbl>, wk32 <dbl>, wk33 <dbl>, wk34 <dbl>, wk35 <dbl>, wk36 <dbl>, ...

```

We get an empty data frame. That's because there is no song that has stayed at #1 for more than 76 weeks. In fact, most songs jumped out of the Top 100 soon after their release, leading to NAs in our data frame.

Here is a smarter way to use `if_all`. Let's say you want to filter those songs that stayed in the Top 50 for the first five weeks:

```
billboard %>%
  filter(
    if_all(
      .cols = matches("wk[1-5]$"),
      .fns  = ~ . <= 50
    )
  )

# A tibble: 13 x 79
  artist track date.ent~1   wk1   wk2   wk3   wk4   wk5   wk6   wk7   wk8   wk9
  <chr>  <chr> <date>     <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1 "Agui~ I Tu~ 2000-04-15  50    39    30    28    21    19    20    17    17
2 "Back~ Shap~ 2000-10-14  39    25    24    15    12    12    10    9     10
3 "Dixi~ Good~ 2000-03-18  40    29    24    24    20    20    20    19    38
4 "Elli~ Hot ~ 1999-11-27  36    21    13    9     7     7     5     7     7
5 "Guy" Danc~ 1999-12-18  46    29    19    22    36    44    58    58    68
6 "Lil ~ Boun~ 2000-08-19  48    35    24    24    20    20    20    20    22
7 "Mado~ Amer~ 2000-02-19  43    35    29    29    33    32    40    58    88
8 "Mado~ Music 2000-08-12  41    23    18    14    2     1     1     1     1
9 "Mart~ She ~ 2000-10-07  38    28    21    21    18    16    13    13    12
10 "N'Sy~ Bye ~ 2000-01-29  42    20    19    14    13    7     6     5     5
11 "No D~ Simp~ 2000-07-01  50    40    39    38    38    48    52    55    80
12 "Pink" Ther~ 2000-03-04  25    15    12    11    11    7     7     12    14
13 "Sant~ Mari~ 2000-02-12  15    8     6     5     2     3     2     2     1
# ... with 67 more variables: wk10 <dbl>, wk11 <dbl>, wk12 <dbl>, wk13 <dbl>,
#   wk14 <dbl>, wk15 <dbl>, wk16 <dbl>, wk17 <dbl>, wk18 <dbl>, wk19 <dbl>,
#   wk20 <dbl>, wk21 <dbl>, wk22 <dbl>, wk23 <dbl>, wk24 <dbl>, wk25 <dbl>,
#   wk26 <dbl>, wk27 <dbl>, wk28 <dbl>, wk29 <dbl>, wk30 <dbl>, wk31 <dbl>,
#   wk32 <dbl>, wk33 <dbl>, wk34 <dbl>, wk35 <dbl>, wk36 <dbl>, wk37 <dbl>,
#   wk38 <dbl>, wk39 <dbl>, wk40 <dbl>, wk41 <dbl>, wk42 <dbl>, wk43 <dbl>,
#   wk44 <dbl>, wk45 <dbl>, wk46 <dbl>, wk47 <dbl>, wk48 <dbl>, wk49 <dbl>, ...
```

First we used the function `matches` to check the condition in the columns “wk1” to “wk5”. Then we defined the condition itself, which searches for values less than or similar to 50.

11.2 How to filter rows that contain missing values

Another very useful use case is filtering rows based on missing values across multiple columns. The next data frame contains three missing values.

```
# This data frame comes from the tidyr documentation:  
# https://tidyr.tidyverse.org/reference/complete.html  
(df <- tibble(  
  item_name = c("a", "a", "b", "b"),  
  group     = c(1, NA, 1, 2),  
  value1    = c(1, NA, 3, 4),  
  value2    = c(4, 5, NA, 7)  
)  
  
# A tibble: 4 x 4  
#>   item_name  group value1 value2  
#>   <chr>       <dbl>  <dbl>  <dbl>  
#> 1 a            1      1      4  
#> 2 a            NA     NA      5  
#> 3 b            1      3      NA  
#> 4 b            2      4      7
```

Now lets keep all rows whose numeric columns do not contain missing value:

```
df %>%  
  filter(  
    if_all(  
      .cols = where(is.numeric),  
      .fns  = ~ !is.na(.)  
    )  
)  
  
# A tibble: 2 x 4  
#>   item_name  group value1 value2  
#>   <chr>       <dbl>  <dbl>  <dbl>  
#> 1 a            1      1      4  
#> 2 b            2      4      7
```

This leaves us with two rows.

11.3 How to create new columns based on conditions across multiple columns

The last example of `if_any` and `if_all` works with `mutate` instead of `filter`. It turns out that we can combine `case_when` with `if_any` / `if_all` to create a new column based on multiple column-spanning conditions. Suppose we want to create a new column that shows whether a song was #1 in the 76 weeks:

```
billboard %>%
  mutate(
    top_song = case_when(
      if_any(
        .cols = contains("wk"),
        .fns  = ~ . == 1
      ) ~ "top song",
      TRUE ~ "no top song"
    )
  ) %>%
  select(artist, track, top_song)
```



```
# A tibble: 317 x 3
  artist       track          top_song
  <chr>        <chr>         <chr>
1 2 Pac        Baby Don't Cry (Keep... no top song
2 2Ge+her     The Hardest Part Of ... no top song
3 3 Doors Down Kryptonite            no top song
4 3 Doors Down Loser                  no top song
5 504 Boyz    Wobble Wobble          no top song
6 98^0        Give Me Just One Nig... no top song
7 A*Teens     Dancing Queen          no top song
8 Aaliyah     I Don't Wanna         no top song
9 Aaliyah     Try Again             top song
10 Adams, Yolanda Open My Heart     no top song
# ... with 307 more rows
```

This works because the left side of the two-sided `case_when` formulas expects a logical value (`<LOGICAL VALUE> == <RIGHT HAND SIDE>`).

Summary

Here's what you can take away from this tutorial.

- `if_any` and `if_all` are similar to `across`, but are usually used with `filter` and `mutate`
- Both functions have a similar structure to `across`. The only significant difference is that the `.fns` argument checks a condition rather than (re)calculating values.
- Use `if_any` if you want to check whether the condition was met by at least one of the selected columns for a given row; use `if_all` if you want to check whether the condition was met by all the selected columns for a given row.

12 How to improve slicing rows

What will this tutorial cover?

In this tutorial you will learn about the functions `slice`, `slice_head`, `slice_tail`, `slice_max`, `slice_min` and `slice_sample`. All functions allow you to slice specific rows of your data frame.

Who do I have to thank?

Many thanks to [akrun](#) and [Dan Chaltiel](#) who discussed how to create bootstraps with `slice_sample` in this [stackoverflow question](#).

Both slicing and filtering allow you to remove or keep rows in a data frame. Essentially, you can use both to achieve the same result, but their approaches differ. While `filter` works with conditions (e.g. `displ > 17`), `slice` works with indices.

12.1 Overview of the `slice` function

Suppose we want to remove the first, second and third rows in the economic data frame (574 rows) with `slice`. The time series dataset shows some important economic variables in the US from 1967 to 2015 by month.

```
 economics %>%  
   slice(1, 2, 3)
```

```
# A tibble: 3 x 6  
 date      pce    pop psavert uempmed unemploy  
 <date>    <dbl>  <dbl>    <dbl>    <dbl>    <dbl>  
 1 1967-07-01 507. 198712    12.6     4.5    2944  
 2 1967-08-01 510. 198911    12.6     4.7    2945  
 3 1967-09-01 516. 199113    11.9     4.6    2958
```

`slice` keeps all rows for which you specify positive indices. Note that in R indexing starts with 1 and not with 0 as in most other programming languages. To make it more clear what rows `slice` keeps, let's add row numbers to our data frame:

```
 economics %>%
  rownames_to_column(var = "row_number")

# A tibble: 574 x 7
  row_number date      pce    pop psavert uempmed unemploy
  <chr>       <date>   <dbl>  <dbl>   <dbl>    <dbl>    <dbl>
1 1          1967-07-01 507. 198712 12.6     4.5     2944
2 2          1967-08-01 510. 198911 12.6     4.7     2945
3 3          1967-09-01 516. 199113 11.9     4.6     2958
4 4          1967-10-01 512. 199311 12.9     4.9     3143
5 5          1967-11-01 517. 199498 12.8     4.7     3066
6 6          1967-12-01 525. 199657 11.8     4.8     3018
7 7          1968-01-01 531. 199808 11.7     5.1     2878
8 8          1968-02-01 534. 199920 12.3     4.5     3001
9 9          1968-03-01 544. 200056 11.7     4.1     2877
10 10        1968-04-01 544. 200208 12.3     4.6     2709
# ... with 564 more rows
```

Let's then slice some arbitrary rows:

```
 economics %>%
  rownames_to_column(var = "row_number") %>%
  slice(c(4, 8, 10))

# A tibble: 3 x 7
  row_number date      pce    pop psavert uempmed unemploy
  <chr>       <date>   <dbl>  <dbl>   <dbl>    <dbl>    <dbl>
1 4          1967-10-01 512. 199311 12.9     4.9     3143
2 8          1968-02-01 534. 199920 12.3     4.5     3001
3 10         1968-04-01 544. 200208 12.3     4.6     2709
```

You can see two things: First, you can see that we have retained lines 4, 8, and 10, which map to our provided indices. Second, you can also provide a vector of indices instead of the comma-separated indices in the `slice` function.

To remove specific rows, we can use negative indices. Suppose, we want to remove the first row from our data frame:

```

economics %>%
  slice(-1)

# A tibble: 573 x 6
  date      pce    pop psavert uempmed unemploy
  <date>    <dbl>  <dbl>   <dbl>    <dbl>    <dbl>
1 1967-08-01 510. 198911  12.6     4.7    2945
2 1967-09-01 516. 199113  11.9     4.6    2958
3 1967-10-01 512. 199311  12.9     4.9    3143
4 1967-11-01 517. 199498  12.8     4.7    3066
5 1967-12-01 525. 199657  11.8     4.8    3018
6 1968-01-01 531. 199808  11.7     5.1    2878
7 1968-02-01 534. 199920  12.3     4.5    3001
8 1968-03-01 544. 200056  11.7     4.1    2877
9 1968-04-01 544. 200208  12.3     4.6    2709
10 1968-05-01 550. 200361  12       4.4    2740
# ... with 563 more rows

```

To remove the last row from our data frame, we need to determine the index of the last row. This is nothing else than the total number of rows in our data frame:

```

economics %>%
  slice(-nrow(.))

# A tibble: 573 x 6
  date      pce    pop psavert uempmed unemploy
  <date>    <dbl>  <dbl>   <dbl>    <dbl>    <dbl>
1 1967-07-01 507. 198712  12.6     4.5    2944
2 1967-08-01 510. 198911  12.6     4.7    2945
3 1967-09-01 516. 199113  11.9     4.6    2958
4 1967-10-01 512. 199311  12.9     4.9    3143
5 1967-11-01 517. 199498  12.8     4.7    3066
6 1967-12-01 525. 199657  11.8     4.8    3018
7 1968-01-01 531. 199808  11.7     5.1    2878
8 1968-02-01 534. 199920  12.3     4.5    3001
9 1968-03-01 544. 200056  11.7     4.1    2877
10 1968-04-01 544. 200208  12.3     4.6    2709
# ... with 563 more rows

```

The function `slice` is quickly explained. More interesting, however, are the helper functions `slice_head`, `slice_tail`, `slice_max`, `slice_min`, and `slice_sample`, which we will now

discuss in more detail. Essentially, all of these functions translate a semantic input (“give me the first 10 lines”) into indices.

12.2 How to slice off the top and bottom of a data frame

Imagine you have conducted a survey and the first two rows in your survey were test data.

```
survey_results <- tribble(
  ~id,    ~name,      ~pre,   ~post,
  1,      "Test",     4,      4,
  2,      "Test",     6,      8,
  3,      "Millner",  2,      9,
  4,      "Josh",     4,      7,
  5,      "Bob",      3,      4
)
```

Of course, you don’t want to do any calculations with the test data, so you need to get rid of them. `sample_head` does the job for you. The function allows you to slice the top n rows of your data frame.

```
survey_results %>%
  slice_head(
  n = 2
)
```

```
# A tibble: 2 x 4
  id name    pre  post
  <dbl> <chr> <dbl> <dbl>
1     1 Test     4     4
2     2 Test     6     8
```

Well, that’s not what we wanted. Instead of slicing the lines we need for our survey, we sliced the test results. Again, the slice function keeps the rows instead of removing them. One solution to this conundrum is to turn the problem around and slice off the tail of the data frame instead of its head:

```
survey_results %>%
  slice_tail(
  n = 3
)
```

```
# A tibble: 3 x 4
  id name      pre  post
  <dbl> <chr>    <dbl> <dbl>
1     3 Millner     2     9
2     4 Josh        4     7
3     5 Bob         3     4
```

This approach however is shaky. How do I know that I need to slice off the last three rows of the data frame? What if the data frame gets larger as the number of participants increases? The better solution is to write code that tells the function to slice off all rows except the first two. This is nothing more than the total number of rows minus 2:

```
survey_results %>%
  slice_tail(
  n = nrow(.) - 2
)

# A tibble: 3 x 4
  id name      pre  post
  <dbl> <chr>    <dbl> <dbl>
1     3 Millner     2     9
2     4 Josh        4     7
3     5 Bob         3     4
```

You could have solved the problem with `filter` as well, which might even be the more robust method:

```
survey_results %>%
  filter(name != "Test")

# A tibble: 3 x 4
  id name      pre  post
  <dbl> <chr>    <dbl> <dbl>
1     3 Millner     2     9
2     4 Josh        4     7
3     5 Bob         3     4
```

12.3 How to slice rows with the highest and lowest values in a given column

A common use case within the slice family is to slice rows that have the highest or lowest value within a column.

Finding these rows with `filter` would be tedious. To see how much, let's give it a try. Suppose we want to find the months in our data frame when unemployment was highest:

```
 economics %>%
  filter(unemploy >= sort(.unemploy, decreasing = TRUE)[10]) %>%
  arrange(desc(unemploy)) %>%
  select(date, unemploy)

# A tibble: 10 x 2
  date      unemploy
  <date>     <dbl>
1 2009-10-01    15352
2 2010-04-01    15325
3 2009-11-01    15219
4 2010-03-01    15202
5 2010-02-01    15113
6 2009-12-01    15098
7 2010-11-01    15081
8 2010-01-01    15046
9 2009-09-01    15009
10 2010-05-01   14849
```

The code inside the `filter` function is hard to read. What we do here is pull the `unemploy` column from the data frame, sort the values and get the tenth value of the sorted vector.

A much easier way to achieve the same result is to use `slice_max`:

```
 economics %>%
  slice_max(
    order_by = unemploy,
    n        = 10) %>%
  select(date, unemploy)

# A tibble: 10 x 2
  date      unemploy
  <date>     <dbl>
1 2009-10-01    15352
2 2010-04-01    15325
3 2009-11-01    15219
4 2010-03-01    15202
5 2010-02-01    15113
6 2009-12-01    15098
7 2010-11-01    15081
8 2010-01-01    15046
9 2009-09-01    15009
10 2010-05-01   14849
```

```

<date>      <dbl>
1 2009-10-01 15352
2 2010-04-01 15325
3 2009-11-01 15219
4 2010-03-01 15202
5 2010-02-01 15113
6 2009-12-01 15098
7 2010-11-01 15081
8 2010-01-01 15046
9 2009-09-01 15009
10 2010-05-01 14849

```

For the first argument `order_by` you specify the column for which the highest values should be taken. With `n` you specify how many of the rows with the highest values you want to keep.

If you are more interested in the percentage of rows with the highest value, you can use the argument `prop`. For example, let's slice the 10% of months with the highest unemployment rate:

```

economics %>%
  slice_max(
    order_by = unemploy,
    prop = .1
  )

# A tibble: 57 x 6
  date        pce      pop psavert uempmed unemploy
  <date>     <dbl>    <dbl>   <dbl>     <dbl>    <dbl>
1 2009-10-01 9932.  308189     5.4    18.9    15352
2 2010-04-01 10113.  309191     6.4    22.1    15325
3 2009-11-01 9940.   308418     5.9    19.8    15219
4 2010-03-01 10089.  309212     5.7    20.4    15202
5 2010-02-01 10031.  309027     5.8    19.9    15113
6 2009-12-01 9999.   308633     5.9    20.1    15098
7 2010-11-01 10355.  310596     6.6    21.0    15081
8 2010-01-01 10002.  308833     6.1    20.0    15046
9 2009-09-01 9883.   307946     5.9    17.8    15009
10 2010-05-01 10131   309369     7.0    22.3    14849
# ... with 47 more rows

```

Similarly, you can keep the rows with the lowest values in a given column. For example, let's find the three months when the unemployment rate was lowest between 1967 and 2015:

```

economics %>%
  slice_min(
    order_by = unemploy,
    n        = 3
  )

# A tibble: 3 x 6
  date      pce    pop psavert uempmed unemploy
  <date>    <dbl>   <dbl>    <dbl>    <dbl>    <dbl>
1 1968-12-01 576. 201621    11.1     4.4    2685
2 1968-09-01 568. 201095    10.6     4.6    2686
3 1968-10-01 572. 201290    10.8     4.8    2689

```

Given the absolute numbers, this was a long time ago.

12.4 How to combine the slice functions with group_by

The `slice` functions become especially powerful when combined with `group_by`. Suppose you want to find each month in the year when the unemployment rate was highest. The trick is that any function called after `group_by` is only applied to the subgroups.

```

library(lubridate)

(highest_unemploy_per_month <- economics %>%
  group_by(year = year(date)) %>%
  slice_max(
    order_by = unemploy,
    n        = 1
  ) %>%
  ungroup())

# A tibble: 49 x 7
  date      pce    pop psavert uempmed unemploy year
  <date>    <dbl>   <dbl>    <dbl>    <dbl>    <dbl> <dbl>
1 1967-10-01 512. 199311    12.9     4.9    3143  1967
2 1968-02-01 534. 199920    12.3     4.5    3001  1968
3 1969-10-01 618. 203302    11.4     4.5    3049  1969
4 1970-12-01 666. 206238    13.2     5.9    5076  1970
5 1971-11-01 721. 208555    13.1     6.4    5161  1971

```

```

6 1972-03-01 749. 209212    11.8    6.6    5038 1972
7 1973-12-01 877. 212785    14.8    4.7    4489 1973
8 1974-12-01 962. 214782    14      5.7    6636 1974
9 1975-05-01 1019. 215523   17.3    9.4    8433 1975
10 1976-11-01 1189  218834   11.4    8.4    7620 1976
# ... with 39 more rows

```

A couple of things happened here. First, I loaded the lubridate package. If you have one of the latest versions of the tidyverse package, lubridate should have already been loaded with `library(tidyverse)` (see [this tweet by Hadley Wickham](#)). I then grouped the `economics` data frame in years (`group_by(year = year(date))`). Yes, you can create new columns inside `group_by`. The function `year` from the lubridate package allows me to extract the year from a date column:

```
year(Sys.Date())
```

```
[1] 2023
```

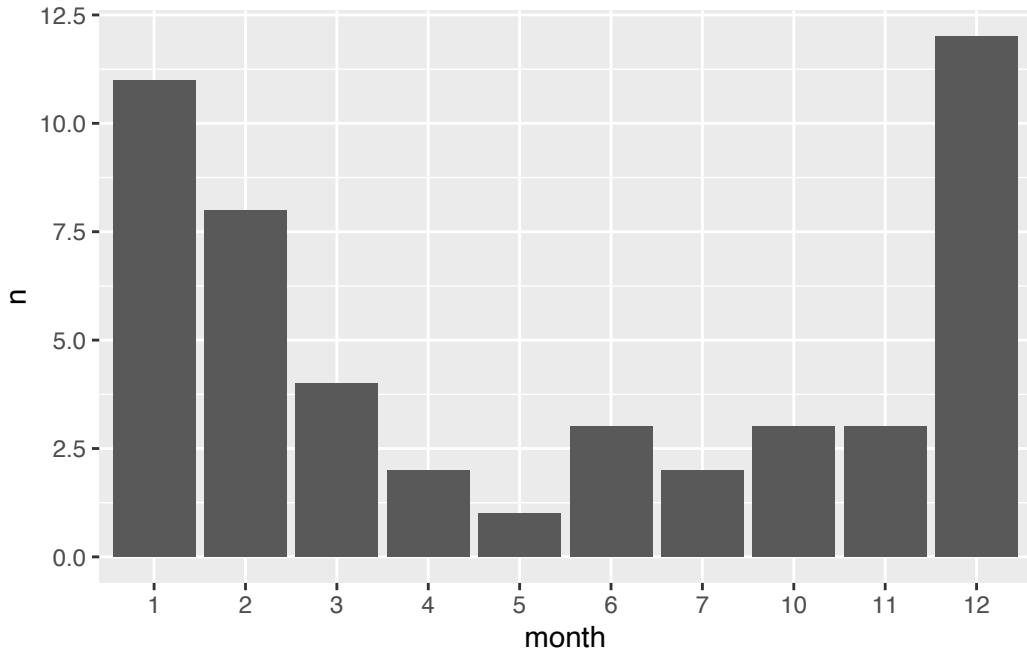
Now that I have grouped the data by year, I can slice the months with the highest unemployment rate within each year (`slice_max(order_by = unemploy, n = 1)`). Again, this works because each function after `group_by` is applied only to the specific groups. At the end we terminate the grouping function with `ungroup`. Otherwise we would not apply the next functions to the whole data frame, but to the individual groups.

This data could be used, for example, to show in which months the unemployment rate is highest:

```

highest_unemploy_per_month %>%
  mutate(
    month = month(date) %>% as.factor
  ) %>%
  count(month) %>%
  ggplot(aes(x = month, y = n)) +
  geom_col()

```



12.5 How to create bootstraps with slice_sample

Another useful function is `slice_sample`. It randomly selects rows from your data frame. You define how many should be selected. Let's, for example, slice 20 rows from our data frame:

```
 economics %>%
  slice_sample(n = 20)
```

```
# A tibble: 20 x 6
  date          pce      pop psavert uempmed unemploy
  <date>     <dbl>    <dbl>   <dbl>    <dbl>    <dbl>
1 2002-03-01  7237.  287190     5.9     8.4    8304
2 2015-03-01 12158. 320231     7.4    12      8504
3 1983-11-01  2366.  235078    10.3    9.3    9499
4 2011-10-01 10753. 312644     6.8    20.6   13594
5 1993-06-01  4440   259963     7.8    8.3    9121
6 2009-12-01  9999.  308633     5.9    20.1   15098
7 2000-10-01  6888.  283201     4.6    6.1    5534
8 1971-07-01  699.   207661    13.8    6.2    5035
9 2012-04-01 10980. 313667     8.7    19.1   12646
10 1999-08-01 6326.  279602     4.7    6.5    5838
```

```

11 2000-05-01 6708. 281877      4.9      5.8      5758
12 2008-03-01 9959. 303907      4       8.7      7822
13 1997-11-01 5661. 274126      6.4      7.6      6308
14 1988-08-01 3368  245240      8.4      5.9      6843
15 1992-02-01 4100. 255448      9.8      8.2      9454
16 2006-08-01 9322. 299263      3.6      8.4      7091
17 2002-11-01 7460. 289106      5.7      9.3      8520
18 2012-01-01 10862. 313183.     8       20.8     12797
19 2010-02-01 10031. 309027      5.8      19.9     15113
20 1984-02-01 2404. 235527     11.7      8.3      8791

```

Since the lines are randomly selected, you will see different rows. Now what happens when we sample all rows from our data frame:

```

economics %>%
  slice_sample(prop = 1)

```

```

# A tibble: 574 x 6
  date        pce      pop psavert uempmed unemploy
  <date>    <dbl>    <dbl>   <dbl>    <dbl>    <dbl>
1 1974-05-01  922.  213513    12.8     4.6     4705
2 1997-08-01  5587  273237     6       7.8     6608
3 1976-06-01  1140.  217861    11.4     7.8     7322
4 1995-10-01  5014.  267456     7.1     8.2     7328
5 2011-01-01  10436. 310961.    7.4     21.5    14013
6 1978-06-01  1426.  222379     10      6      6028
7 1969-06-01  601.   202507    11.1     4.4     2816
8 1995-01-01  4851.  265044     7.5      8      7375
9 1997-07-01  5549.  272912     6.1     8.3     6655
10 1972-01-01  732.   208917    12.5     6.2     5019
# ... with 564 more rows

```

Nothing really changes. We will get the same data frame. Why? Because `slice_sample` by default samples without replacement. Once we have selected a row, we cannot select it again. Consequently, there will be no duplicate rows in our data frame. However, if we set the `replace` argument to `TRUE`, we will perform sampling with replacement:

```

set.seed(455)
(sample_with_replacement <- economics %>%
  slice_sample(prop = 1, replace = TRUE))

```

```

# A tibble: 574 x 6
  date      pce    pop psavert uempmed unemploy
  <date>    <dbl>   <dbl>   <dbl>    <dbl>    <dbl>
1 1968-01-01  531. 199808    11.7     5.1    2878
2 1968-07-01  563. 200706    10.7     4.5    2883
3 2010-03-01 10089. 309212     5.7    20.4   15202
4 1969-06-01  601. 202507    11.1     4.4    2816
5 1968-09-01  568. 201095    10.6     4.6    2686
6 1978-09-01 1453. 223053    10.6     5.6    6125
7 1975-10-01 1061. 216587    13.4     8.6    7897
8 1980-11-01 1827. 228612    11.6     7.7    8023
9 1967-08-01  510. 198911    12.6     4.7    2945
10 2012-09-01 11062. 314647.    8.2    18.8   12115
# ... with 564 more rows

```

I set the seed to 455 so you get the same results. We can find the duplicate rows with the function `get_dupes` from the `janitor` package:

```

sample_with_replacement %>%
  janitor::get_dupes()

# A tibble: 363 x 7
  date      pce    pop psavert uempmed unemploy dupe_count
  <date>    <dbl>   <dbl>   <dbl>    <dbl>    <dbl>      <int>
1 1967-08-01  510. 198911    12.6     4.7    2945        2
2 1967-08-01  510. 198911    12.6     4.7    2945        2
3 1968-02-01  534. 199920    12.3     4.5    3001        3
4 1968-02-01  534. 199920    12.3     4.5    3001        3
5 1968-02-01  534. 199920    12.3     4.5    3001        3
6 1968-07-01  563. 200706    10.7     4.5    2883        3
7 1968-07-01  563. 200706    10.7     4.5    2883        3
8 1968-07-01  563. 200706    10.7     4.5    2883        3
9 1968-09-01  568. 201095    10.6     4.6    2686        2
10 1968-09-01 568. 201095    10.6     4.6    2686       2
# ... with 353 more rows

```

As you can see, the first line appears twice in the data frame. Now, why would we do this? This functionality allows us to create bootstraps from our data frame. Bootstrapping is a technique where a set of samples of the same size are drawn from a single original sample. For example, if you have a vector `c(1, 4, 5, 6)`, you can create the following bootstraps from this vector: `c(1, 4, 4, 6)`, `c(1, 1, 1, 1)` or `c(5, 5, 1, 6)`. Some values appear

more than once because bootstrapping allows each value to be pulled multiple times from the original data set. Once you have your bootstraps, you can calculate metrics from them. For example, the mean value of each bootstrap. The underlying logic of this technique is that since the sample itself is from a population, the bootstraps act as proxies for other samples from that population.

Now that we have created one bootstrap from our sample, we can create many. In the following code I have used `map` to create 2000 bootstraps from my original sample.

```
bootstraps <- map(1:2000, ~slice_sample(economics, prop = 1, replace = TRUE))

bootstraps %>% head(n = 2)

[[1]]
# A tibble: 574 x 6
  date      pce    pop psavert uempmed unemploy
  <date>    <dbl>  <dbl>    <dbl>    <dbl>    <dbl>
1 1992-04-01 4132. 255992     9.9     8.5    9415
2 1993-09-01 4512. 260867     6.9     8.3    8714
3 1999-05-01 6226. 278717     4.9     6.5    5796
4 1989-07-01 3586. 247342     8.2     5.6    6495
5 1986-05-01 2858. 240271     9.3     6.8    8439
6 2006-11-01 9380. 300094     3.9     8.3    6872
7 2000-05-01 6708. 281877     4.9     5.8    5758
8 1996-12-01 5379. 271125     6.4     7.8    7253
9 1992-10-01 4285. 257861     8       9      9398
10 1970-11-01 657. 206024    13.6     5.6    4898
# ... with 564 more rows

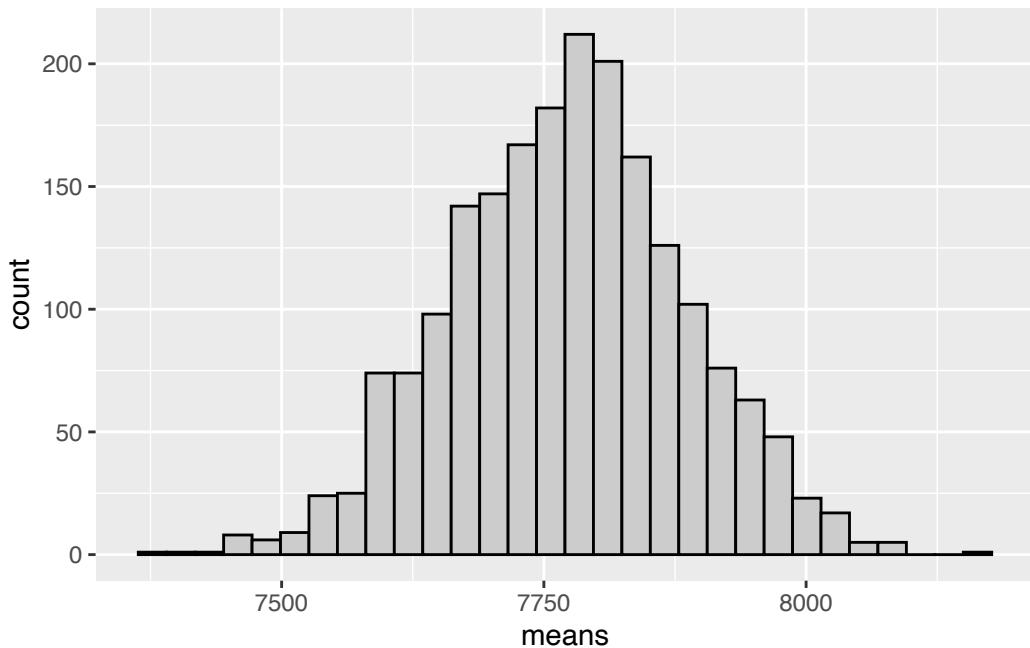
[[2]]
# A tibble: 574 x 6
  date      pce    pop psavert uempmed unemploy
  <date>    <dbl>  <dbl>    <dbl>    <dbl>    <dbl>
1 1979-11-01 1657. 226027     9.7     5.3    6238
2 1992-07-01 4205. 256894     9.6     8.6    9850
3 1975-03-01 991. 215198    12.7     7.2    7978
4 1970-07-01 648. 205052    13.5     5.1    4175
5 2013-11-01 11488. 317228     6.2    17.1   10787
6 1999-12-01 6539. 280716     4.4     5.8    5653
7 1978-10-01 1467. 223271    10.7     5.9    5947
8 1980-04-01 1695. 227061    11.3     5.8    7358
9 1995-12-01 5098. 267943     6.1     8.3    7423
```

```
10 1993-11-01 4554. 261425      6.3      8.3      8542
# ... with 564 more rows
```

`map` returns a list of data frames. Once we have the bootstraps, we can calculate any metric from them. Usually one calculates confidence intervals, standard deviations, but also measures of center like the mean from the bootstraps. Let's do the latter:

```
means <- map_dbl(bootstraps, ~ mean(.unemploy))

ggplot(NULL, aes(x = means)) +
  geom_histogram(fill = "grey80", color = "black")
```



As you can see, the distribution of the mean is normally distributed. Most of the mean values are around 7750, which is pretty close to the mean value of our sample:

```
economics$unemploy %>% mean
```

```
[1] 7771.31
```

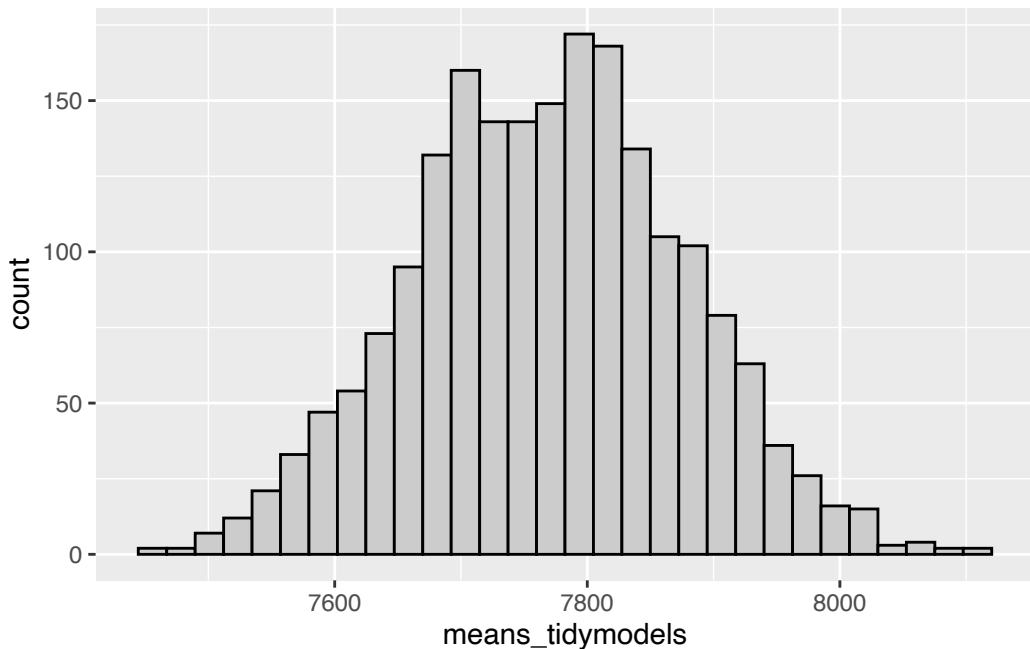
We can compare this result with the `bootstraps` function from the `tidymodels` package, which is more rigorous than our approach (It is not too important that you understand the code

here. Basically, we use the `bootstraps` function to create a similar distribution of the mean values).

```
bootstraps_tidymodels <- rsample::bootstraps(economics, times = 2000)

means_tidymodels <- map_dbl(bootstraps_tidymodels$splits,
                           ~ mean(rsample::analysis(.)$unemploy))

ggplot(NULL, aes(x = means_tidymodels)) +
  geom_histogram(fill = "grey80", color = "black")
```



This distribution is very similar to our distribution we created with `slice_sample`.

i Summary

Here's what you can take away from this tutorial.

- The slice functions slice rows based on their indices. Positive indices are kept, negative indices are removed.
- `group_by` and `slice_max / slice_min` is a powerful combination to reduce the size of your data frame by finding the rows within groups with the highest or lowest values.

- `slice_sample` can be used to create bootstraps from your data frame

13 How to do rowwise calculations

What will this tutorial cover?

In this tutorial you will learn how to compute on a data frame per row. You will be able to calculate summary statistics one row at a time with `rowwise` and some other vectorized functions that are faster than `rowwise`.

Who do I have to thank?

I have to thank [Jeffrey Girard](#) for his excellent tutorial on row-wise means in dplyr. He provided some excellent examples and ideas that I built this tutorial on. Also, I want to thank the guys on the tidyverse team who wrote [a really helpful vignette](#) on row-wise operations in dplyr.

Some of the simplest calculations in R can be confusing at times. Take, for example, this: You want to calculate the mean for each row of your data frame. Suppose this data frame represents the test scores of four students at three measurement times: first, second, and third.

```
(dframe <- tibble(  
  name = c("Jakob", "Eliud", "Jamal", "Emily"),  
  first = c(1, 4, 7, 10),  
  second = c(2, 5, 8, 11),  
  third = c(3, 6, 9, 12)  
)  
  
# A tibble: 4 x 4  
#>   name  first second third  
#>   <chr> <dbl>  <dbl> <dbl>  
#> 1 Jakob     1      2      3  
#> 2 Eliud     4      5      6  
#> 3 Jamal     7      8      9  
#> 4 Emily    10     11     12
```

Calculating the mean for each person from these three measurement points should be straightforward:

```

dframe %>%
  mutate(
    mean_performance = mean(c(first, second, third))
  )

# A tibble: 4 x 5
  name   first  second  third mean_performance
  <chr> <dbl>   <dbl>   <dbl>             <dbl>
1 Jakob     1      2      3            6.5
2 Eliud     4      5      6            6.5
3 Jamal     7      8      9            6.5
4 Emily    10     11     12           6.5

```

Apparently not. 6.5 is certainly not the average value of one of these rows. What's going on here? It turns out that `mean` calculates the average of all 9 values, not three for each row. Similar to this:

```

mean(c(c(1, 4, 7, 10),
       c(2, 5, 8, 11),
       c(3, 6, 9, 12)))

```

```
[1] 6.5
```

We would say that `mean` is not a vectorized function because it does not perform its calculations vector by vector. Instead, it throws each vector into a box and calculates the overall average of all its values.

Sooner or later, most of us will stumble upon this problem. And this applies not only to `mean`, but also to `sum`, `min`, `max`, and `median`:

```

dframe %>%
  mutate(
    mean_performance = mean(c(first, second, third)),
    sum_performance = sum(c(first, second, third)),
    min_performance = min(c(first, second, third)),
    max_performance = max(c(first, second, third)),
    median_performance = median(c(first, second, third))
  )

```

```

# A tibble: 4 x 9
  name   first  second  third mean_performance sum_perfor~1 min_p~2 max_p~3 media~4
  <chr> <dbl>  <dbl>  <dbl>             <dbl>           <dbl>  <dbl>  <dbl>  <dbl>
1 Jakob     1      2      3          6.5            78     1     12    6.5
2 Eliud     4      5      6          6.5            78     1     12    6.5
3 Jamal     7      8      9          6.5            78     1     12    6.5
4 Emily    10     11     12          6.5            78     1     12    6.5
# ... with abbreviated variable names 1: sum_performance, 2: min_performance,
#   3: max_performance, 4: median_performance

```

So far, so good. What confuses many is the fact that some row-wise calculations yield unexpected results, while others do not. These three use cases, for example, work flawlessly.

```

dframe %>%
  mutate(
    combined = paste(first, second, third),
    sum       = first + second + third,
    mean      = (first + second + third) / 3
  )

# A tibble: 4 x 7
  name   first  second  third combined    sum  mean
  <chr> <dbl>  <dbl>  <dbl> <chr>    <dbl> <dbl>
1 Jakob     1      2      3 1 2 3       6     2
2 Eliud     4      5      6 4 5 6       15    5
3 Jamal     7      8      9 7 8 9       24    8
4 Emily    10     11     12 10 11 12     33   11

```

So the problem occurs with functions that are not vectorized (e.g. `mean`) and with base R functions that compute summary statistics. There are several solutions to this problem. The first is `rowwise`:

13.1 Introducing `rowwise`

`rowwise` was introduced [in dplyr in 2020 with version 1.1.0](#). Essentially, the function ensures that operations are performed row by row. It works similar to `group_by`. It does not change how the data frame looks, but how calculations are performed with the data frame. Let's apply the function to our toy data frame and see what results we get:

```

dframe %>%
  rowwise()

# A tibble: 4 x 4
# Rowwise:
#> # ... with 4 rows and 4 variables:
#> #   name <chr>, first <dbl>, second <dbl>, third <dbl>
#> #   # 1 Jakob     1        2        3
#> # 2 Eliud     4        5        6
#> # 3 Jamal     7        8        9
#> # 4 Emily    10       11       12

```

As you can see, nothing changes. The output just tells you that further calculations will be performed row by row.

Now, if we calculate the mean of the individual's performance, we get the correct results:

```

dframe %>%
  rowwise() %>%
  mutate(
    mean_performance = mean(c(first, second, third))
  )

# A tibble: 4 x 5
# Rowwise:
#> # ... with 4 rows and 5 variables:
#> #   name <chr>, first <dbl>, second <dbl>, third <dbl>, mean_performance <dbl>
#> #   # 1 Jakob     1        2        3             2
#> # 2 Eliud     4        5        6             5
#> # 3 Jamal     7        8        9             8
#> # 4 Emily    10       11       12            11

```

To illustrate that `rowwise` is just a special case of `group_by`, we use the `group_by` function to obtain the same results:

```

dframe %>%
  group_by(name) %>%
  mutate(
    mean = mean(c(first, second, third))
  )

```

```
# A tibble: 4 x 5
# Groups:   name [4]
  name  first second third  mean
  <chr> <dbl>  <dbl> <dbl> <dbl>
1 Jakob     1      2      3      2
2 Eliud     4      5      6      5
3 Jamal     7      8      9      8
4 Emily    10     11     12     11
```

The same logic applies to all other base R functions that compute summary statistics:

```
dframe %>%
  rowwise() %>%
  mutate(
    mean_performance = mean(c(first, second, third)),
    sum_performance = sum(c(first, second, third)),
    min_performance = min(c(first, second, third)),
    max_performance = max(c(first, second, third)),
    median_performance = median(c(first, second, third))
  )

# A tibble: 4 x 9
# Rowwise:
  name  first second third mean_performance sum_perfor~1 min_p~2 max_p~3 media~4
  <chr> <dbl>  <dbl> <dbl>           <dbl>          <dbl>    <dbl>    <dbl>    <dbl>
1 Jakob     1      2      3                 2             6      1      3      2
2 Eliud     4      5      6                 5            15      4      6      5
3 Jamal     7      8      9                 8            24      7      9      8
4 Emily    10     11     12                11            33     10     12     11
# ... with abbreviated variable names 1: sum_performance, 2: min_performance,
#   3: max_performance, 4: median_performance
```

13.2 How to use tidyselect functions with rowwise

It turns out that it is not possible to add a tidyselect function within `mean` or any of the other functions:

```
dframe %>%
  rowwise() %>%
  mutate(
```

```

    mean_performance = mean(where(is.numeric))
  )

Error in `mutate()`:
! Problem while computing `mean_performance = mean(where(is.numeric))` .
  The error occurred in row 1.
Caused by error in `where()`:
! could not find function "where"
Run `rlang::last_error()` to see where the error occurred.

```

In these cases `c_across` comes to your rescue. It was developed especially for `rowwise` and can be considered a wrapper around `c()`. Let's try it first without a tidyselect function:

```

dframe %>%
  rowwise() %>%
  mutate(
    mean_performance = mean(c_across(c(first, second, third)))
  )

# A tibble: 4 x 5
# Rowwise:
  name   first  second  third mean_performance
  <chr> <dbl>   <dbl>   <dbl>             <dbl>
1 Jakob     1      2      3                 2
2 Eliud     4      5      6                 5
3 Jamal     7      8      9                 8
4 Emily    10     11     12                11

```

The expected results. However, instead of `c()` you can use any tidyselect function of your choice:

```

dframe %>%
  rowwise() %>%
  mutate(
    mean_performance = mean(c_across(where(is.numeric)))
  )

# A tibble: 4 x 5
# Rowwise:
  name   first  second  third mean_performance
  <chr> <dbl>   <dbl>   <dbl>             <dbl>
1 Jakob     1      2      3                 2
2 Eliud     4      5      6                 5
3 Jamal     7      8      9                 8
4 Emily    10     11     12                11

```

	<chr>	<dbl>	<dbl>	<dbl>	<dbl>
1	Jakob	1	2	3	2
2	Eliud	4	5	6	5
3	Jamal	7	8	9	8
4	Emily	10	11	12	11

This is especially important if you need to perform calculations on many columns. For example, the `billboard` data frame:

```
billboard %>%
  rowwise() %>%
  transmute(
    artist,
    track,
    sum = sum(c_across(contains("wk"))), na.rm = TRUE
  )

# A tibble: 317 x 3
# Rowwise:
#   artist      track        sum
#   <chr>       <chr>     <dbl>
# 1 2 Pac      Baby Don't Cry (Keep... 598
# 2 2Ge+her    The Hardest Part Of ... 270
# 3 3 Doors Down Kryptonite          1403
# 4 3 Doors Down Loser                1342
# 5 504 Boyz   Wobble Wobble        1012
# 6 98^0       Give Me Just One Nig... 753
# 7 A*Teens    Dancing Queen         485
# 8 Aaliyah    I Don't Wanna        1041
# 9 Aaliyah    Try Again            533
# 10 Adams, Yolanda Open My Heart  1355
# ... with 307 more rows
```

13.3 Don't forget to `ungroup()`

Not using `rowwise` can sometimes lead to problems, but using `rowwise` can also lead to problems. The most common mistake is forgetting to `ungroup` the data frame. I told you that `rowwise` does nothing but group the data so that the calculations are performed row by row. Similar to `group_by`, `rowwise` can also be ungrouped with `ungroup()`. If you don't do that, problems like this can occur:

```

dframe %>%
  rowwise() %>%
  mutate(
    mean = mean(c(first, second, third))
  ) %>%
  summarise(
    mean_across_students = mean(mean)
  )

# A tibble: 4 x 1
  mean_across_students
  <dbl>
1 2
2 5
3 8
4 11

```

Instead of calculating the mean for each person, the mean values are repeated for each student. The reason for this is that `summarise` still performs its computations row by row. To correct this logical error, we must `ungroup` the data frame:

```

dframe %>%
  ungroup() %>%
  mutate(
    mean = mean(c(first, second, third))
  ) %>%
  ungroup() %>%
  summarise(
    mean_across_students = mean(mean)
  )

# A tibble: 1 x 1
  mean_across_students
  <dbl>
1 6.5

```

So remember that `rowwise` should never join a party without `ungroup`.

13.4 Calculating proportions with `rowwise`

A nice use case of `rowwise` is to convert your values into proportions (I found the example in the [official vignette](#)). To calculate proportions, you must first calculate the sum of all values. Since the data is in a wide format, this can be done with `rowwise`:

```
(sums_per_row <- dframe %>%
  rowwise() %>%
  mutate(sum_per_row = sum(first, second, third)) %>%
  ungroup())

# A tibble: 4 x 5
  name   first  second  third sum_per_row
  <chr> <dbl>   <dbl>   <dbl>      <dbl>
1 Jakob    1       2       3        6
2 Eliud    4       5       6       15
3 Jamal    7       8       9       24
4 Emily   10      11      12      33
```

With the column `sum_per_row`, we can use `across` and convert all numeric columns to proportions:

```
sums_per_row %>%
  transmute(
    name,
    across(
      .cols = where(is.numeric),
      .fns  = ~ . / sum_per_row
    )
  )

# A tibble: 4 x 5
  name   first  second  third sum_per_row
  <chr> <dbl>   <dbl>   <dbl>      <dbl>
1 Jakob  0.167  0.333  0.5        1
2 Eliud  0.267  0.333  0.4        1
3 Jamal  0.292  0.333  0.375     1
4 Emily  0.303  0.333  0.364     1
```

13.5 If you care about performance, choose alternativ approaches

For small data sets `rowwise` is sufficient. However, if your data frame is very large, `rowwise` will show performance problems compared to alternative approaches. It is simply not very fast. Let's have a look:

To measure the performance, we will use the `bench` package:

```
library(bench)
```

Let's see how long it takes to calculate the sums of our data frame with four rows and three columns:

```
bench::mark(
  dframe %>%
    rowwise() %>%
    mutate(
      mean_performance = min(c(first, second, third))
    )
)$total_time
```

```
[1] 464ms
```

Your results will vary. On my computer it took about 450 milliseconds. What I haven't told you yet is that you can use other functions to calculate summary statistics on a row-by-row basis without using `rowwise`. `pmin` is one of those functions. Let's see how fast the code executes with `pmin`:

```
bench::mark(
  dframe %>%
    mutate(
      mean_performance = pmin(first, second, third)
    )
)$total_time
```

```
[1] 466ms
```

It is slightly faster than with `rowwise`. However, the performance advantage comes when the data frame is very large. For example, the diamond dataset has 53,940 rows (not super large, of course, but an approximation). Let's compare the efficiency of the two approaches using this data frame:

```

bench::mark(
  diamonds %>%
    rowwise() %>%
    mutate(
      mean_performance = min(c(x, y, z))
    ) %>%
    ungroup(),
  diamonds %>%
    mutate(
      mean_performance = pmin(x, y, z)
    )
) $total_time

```

Warning: Some expressions had a GC in every iteration; so filtering is disabled.

[1] 526ms 501ms

A difference of more than 50 milliseconds on my computer. Next, we simply duplicate the diamond data set 10 times and compare the results again:

```

duplicated_diamonds <- bind_rows(
  diamonds, diamonds, diamonds, diamonds, diamonds,
  diamonds, diamonds, diamonds, diamonds, diamonds
)

bench::mark(
  duplicated_diamonds %>%
    rowwise() %>%
    mutate(
      mean_performance = min(c(x, y, z))
    ) %>%
    ungroup(),
  duplicated_diamonds %>%
    mutate(
      mean_performance = pmin(x, y, z)
    )
) $total_time

```

Warning: Some expressions had a GC in every iteration; so filtering is disabled.

```
[1] 1.52s 500.73ms
```

For half a million rows, `pmap` is three times faster than working with `rowwise`. So if you are working with really large data frames, keep in mind that `rowwise` is not the most efficient method. Consider the alternatives. Let's take a closer look at these alternatives at the end of this tutorial.

13.6 A short deep-dive into `pmax` and `pmin`

You have already seen `pmin`. Compared to `min`, `pmin` works because it is a vectorized function. It calculates the minimum value for one or more vectors. In our case, these vectors are rows. If you have `pmin`, you must also have `pmax`. With `pmax` you can calculate the maximum value for all columns on a row basis:

```
dframe %>%
  mutate(
    max = pmax(first, second, third),
    min = pmin(first, second, third)
  )

# A tibble: 4 x 6
  name   first  second  third   max   min
  <chr> <dbl>   <dbl>   <dbl> <dbl> <dbl>
1 Jakob     1       2       3      3      1
2 Eliud     4       5       6      6      4
3 Jamal     7       8       9      9      7
4 Emily    10      11      12     12     10
```

Unfortunately there is no function `pmean`, `pmedian` or `psum`. However, we can calculate row-based sums with `rowSums`.

13.7 A deep-dive into `rowSums` and `rowMeans`

`rowSums` does what it says. It calculates the sums for rows. The function works with a matrix or a data frame. Providing a vector of values will not work, so we have to use this matrix trick to make it work:

```

dframe %>%
  mutate(
    sum = rowSums(matrix(c(first, second, third), ncol = 3))
  )

# A tibble: 4 x 5
  name   first  second  third   sum
  <chr> <dbl>   <dbl>   <dbl> <dbl>
1 Jakob     1       2       3      6
2 Eliud     4       5       6     15
3 Jamal     7       8       9     24
4 Emily    10      11      12     33

```

Similarly, we can use a subset of the data frame and put it into `rowSums`:

```
rowSums(dframe %>% select(-name))
```

```
[1] 6 15 24 33
```

A trick to not use a matrix is to use `select` with `mutate` and the piped data frame `..`:

```

dframe %>%
  mutate(
    sum1 = rowSums(select(., first, second, third)),
    sum2 = rowSums(across(first:third)),
    sum3 = rowSums(select(., matches("first|second|third"))),
  )

# A tibble: 4 x 7
  name   first  second  third  sum1  sum2  sum3
  <chr> <dbl>   <dbl>   <dbl> <dbl> <dbl> <dbl>
1 Jakob     1       2       3      6      6      6
2 Eliud     4       5       6     15     15     15
3 Jamal     7       8       9     24     24     24
4 Emily    10      11      12     33     33     33

```

Since `rowSums` can take a data frame, we can simply use a subset of our data frame as in `select(., first, second, third)` or `select(., matches("first|second|third"))`.

Similarly, we can calculate the mean of each row with `rowMeans`.

```

dframe %>%
  mutate(
    sum1 = rowMeans(matrix(c(first, second, third), ncol = 3)),
    sum2 = rowMeans(across(first:third)),
    sum3 = rowMeans(select(., first, second, third)),
    sum4 = rowMeans(select(., matches("first|second|third"))))
  )

# A tibble: 4 x 8
  name   first  second  third  sum1  sum2  sum3  sum4
  <chr> <dbl>   <dbl>   <dbl> <dbl> <dbl> <dbl> <dbl>
1 Jakob     1       2       3      2      2      2      2
2 Eliud     4       5       6      5      5      5      5
3 Jamal     7       8       9      8      8      8      8
4 Emily    10      11      12     11     11     11     11

```

Summary

Here's what you can take away from this tutorial.

- Be careful to calculate summary statistics row-wise using base R functions. Since these functions are not vectorized, they will calculate the summary statistics with all values from your data frame.
- `rowwise` ensures that computations are done one row at a time.
- Never forget to `ungroup rowwise`. If you don't your calculations might be wrong.
- If you care about performance, use `pmin`, `pmap`, `rowSums` or `rowMeans` instead of `rowwise`.

Part V

Improve grouping data

14 How to run many models with the new dplyr grouping functions

What will this tutorial cover?

In this tutorial, you will learn six ways to run many models simultaneously using a set of Tidyverse functions. We will create 142 linear regression models and figure out how to extract the model parameters and test statistics from them. Also, you will get to know some new grouping function introduced to dplyr in 2019.

Who do I have to thank?

I would like to thank [Mara Averick](#), [Chris Etienne](#), and [Indrajeet Patil](#) () for their insightful tweets on this topic and helping me get started.

[Hadley Wickham](#) showed us in 2016 that you can run many models at once with a few Tidyverse functions (see also [this chapter from the R for Data Science book](#)). Using many models can be a powerful technique to gain insights from your data. A classic example is the [Gapminder dataset](#). Suppose you want to find out whether life expectancy has shown a linear trend over the past 50 years in countries around the world.

First, we need access to the data. Fortunately, the [gapminder package](#) contains a data frame with the life expectancy of each country on each continent from 1952 to 2007:

```
library(gapminder)  
gapminder
```

```
# A tibble: 1,704 x 6  
  country   continent year lifeExp      pop gdpPercap  
  <fct>     <fct>    <int>   <dbl>    <int>     <dbl>  
1 Afghanistan Asia     1952    28.8    8425333    779.  
2 Afghanistan Asia     1957    30.3    9240934    821.  
3 Afghanistan Asia     1962    32.0    10267083   853.  
4 Afghanistan Asia     1967    34.0    11537966   836.  
5 Afghanistan Asia     1972    36.1    13079460   740.
```

```

6 Afghanistan Asia      1977    38.4 14880372    786.
7 Afghanistan Asia      1982    39.9 12881816    978.
8 Afghanistan Asia      1987    40.8 13867957    852.
9 Afghanistan Asia      1992    41.7 16317921    649.
10 Afghanistan Asia     1997    41.8 22227415    635.
# ... with 1,694 more rows

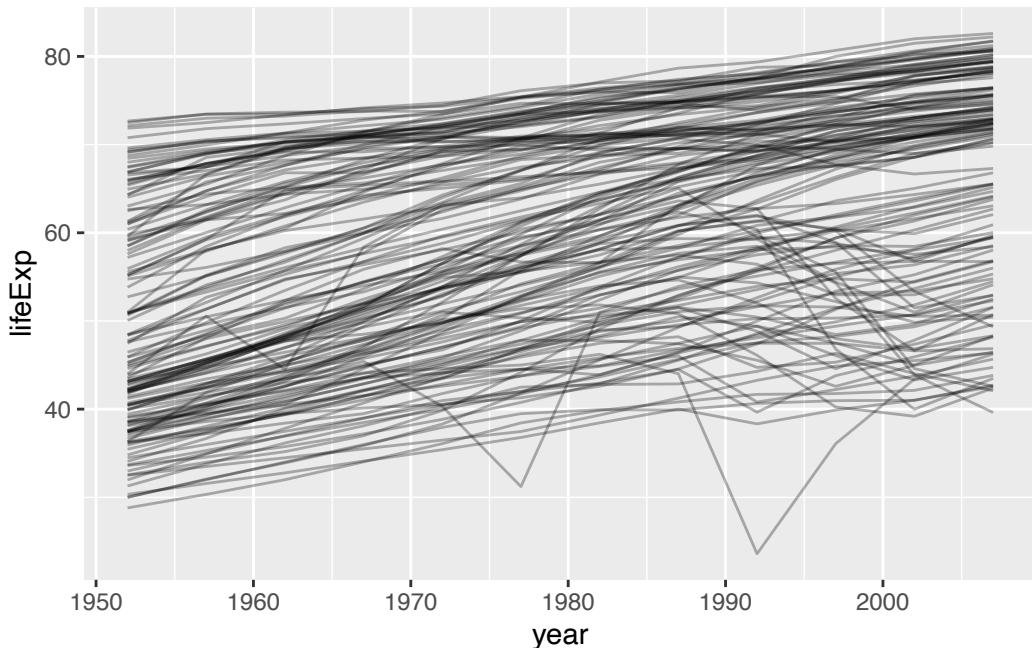
```

We can trace the development of life expectancy in these countries using a line chart:

```

ggplot(gapminder,
       aes(x = year, y = lifeExp, group = country)) +
  geom_line(alpha = .3)

```

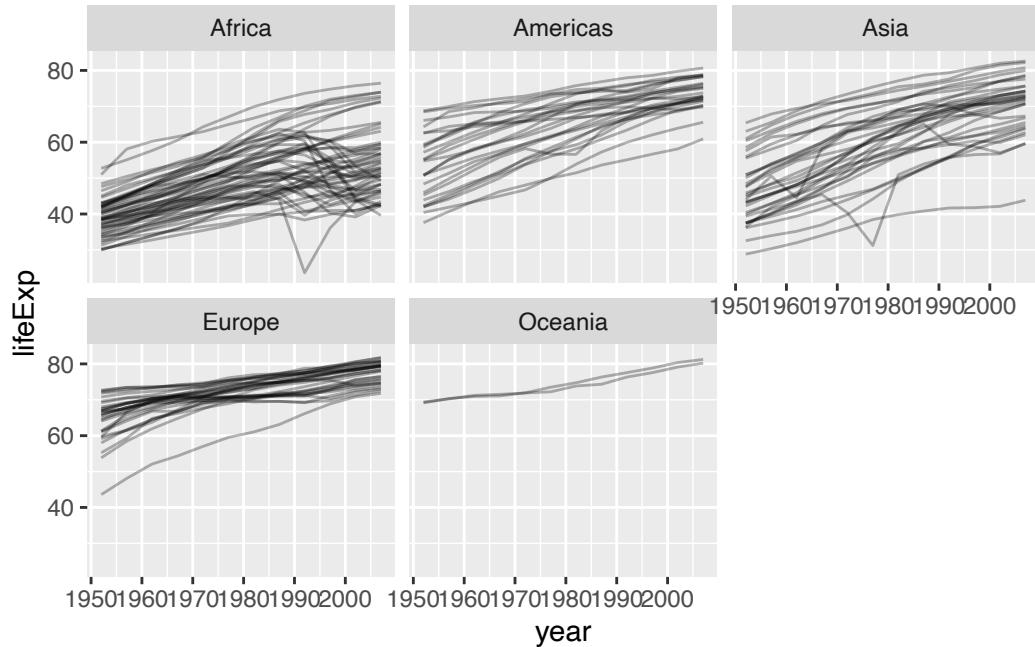


What we see is that on average life expectancy has risen from 1952 to 2007. Some countries have experienced a drastic decline in life expectancy. We can get a better picture if we split the plot by continent.

```

ggplot(gapminder,
       aes(x = year, y = lifeExp, group = country)) +
  geom_line(alpha = .3) +
  facet_wrap(vars(continent))

```



We can see that there is one major decline in Africa and two major declines in Asia.

14.1 Building a single linear model

To find out how well life expectancy has followed a linear trend over the past 50 years we build a linear regression model with year as the independent variable and life expectancy as the dependent variable :

```
model <- lm(lifeExp ~ year, data = gapminder)
```

We tell the function `lm` that `year` is our independent variable and `lifeExp` is our dependent variable.

Once we have created the model, we can retrieve the results parameters and test statistics with the `summary` function:

```
summary(model)
```

```
Call:  
lm(formula = lifeExp ~ year, data = gapminder)
```

```

Residuals:
    Min      1Q  Median      3Q     Max 
-39.949 -9.651   1.697  10.335  22.158 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept) -585.65219   32.31396 -18.12 <2e-16 ***
year          0.32590    0.01632   19.96 <2e-16 ***  
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 11.63 on 1702 degrees of freedom
Multiple R-squared:  0.1898,    Adjusted R-squared:  0.1893 
F-statistic: 398.6 on 1 and 1702 DF,  p-value: < 2.2e-16

```

We see that our regression coefficient for the independent variable year is positive (year = 0.32590), which means that life expectancy has increased over the years.

To perform further analysis with the parameters of our model, we can pipe the model into the `tidy` function from the `broom` package:

```

library(broom)

model %>%
  tidy()

# A tibble: 2 x 5
  term      estimate std.error statistic p.value
  <chr>      <dbl>     <dbl>     <dbl>    <dbl>
1 (Intercept) -586.       32.3      -18.1 2.90e-67
2 year         0.326     0.0163     20.0 7.55e-80

```

Similarly, we can get the test statistics of our model with `glance`:

```

model %>%
  broom::glance()

# A tibble: 1 x 12
r.squared adj.r.sq~1 sigma stati~2 p.value     df logLik     AIC     BIC devia~3
<dbl>        <dbl> <dbl>    <dbl>    <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>    <dbl>

```

```

1      0.190      0.189  11.6     399. 7.55e-80      1 -6598. 13202. 13218. 230229.
# ... with 2 more variables: df.residual <int>, nobs <int>, and abbreviated
#   variable names 1: adj.r.squared, 2: statistic, 3: deviance

```

That's all well and good, but how would we apply the same model to each country?

14.2 The split > apply > combine technique

The solution to this problem is the split > apply > combine technique. You have already come across this idea twice:

The combination of `group_by` and `summarise` is a method of split > apply > combine. For example, we could split the data by continent and year, calculate the mean for each group (apply), and combine the results in a data frame:

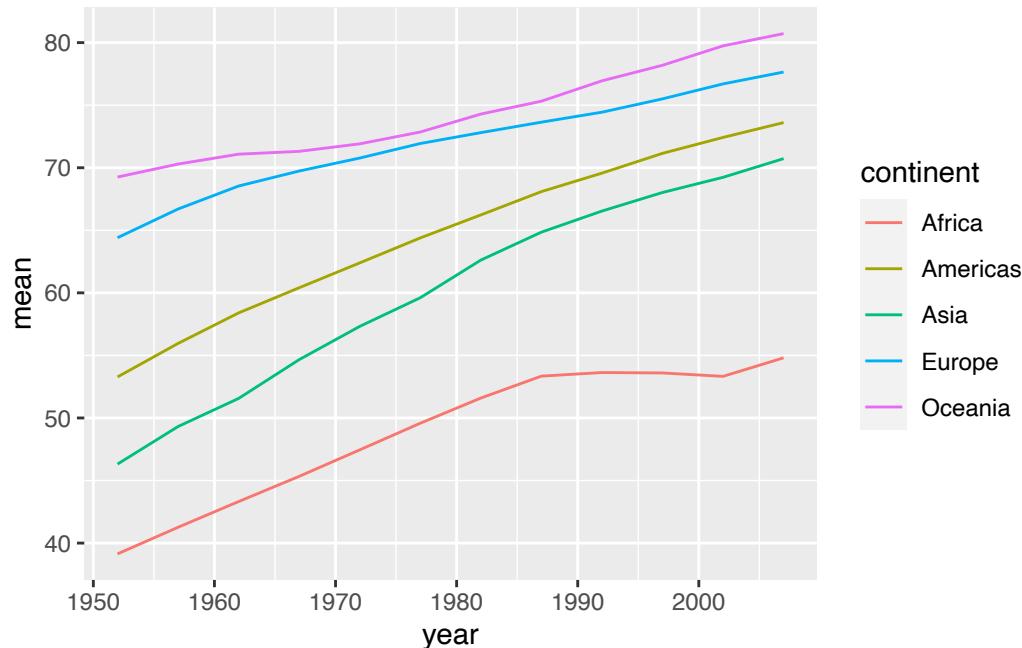
```
(group_by_summarise_example <- gapminder %>%
  group_by(continent, year) %>%
  summarise(mean = mean(lifeExp, na.rm = TRUE)) %>%
  ungroup())
```

``summarise()` has grouped output by 'continent'. You can override using the `groups` argument.`

```
# A tibble: 60 x 3
  continent year  mean
  <fct>     <int> <dbl>
1 Africa     1952  39.1
2 Africa     1957  41.3
3 Africa     1962  43.3
4 Africa     1967  45.3
5 Africa     1972  47.5
6 Africa     1977  49.6
7 Africa     1982  51.6
8 Africa     1987  53.3
9 Africa     1992  53.6
10 Africa    1997  53.6
# ... with 50 more rows
```

With this data, we can plot the development of life expectancy on the five continents:

```
group_by_summarise_example %>%
  ggplot(aes(x = year, y = mean)) +
  geom_line(aes(color = continent))
```



Another method for slice > apply > combine is to use `group_by` with `slice_max`, and `ungroup`:

```
(group_by_with_slice_max <- gapminder %>%
  group_by(continent) %>%
  slice_max(lifeExp, n = 1) %>%
  ungroup())
```

```
# A tibble: 5 x 6
  country continent year lifeExp      pop gdpPerCap
  <fct>   <fct>    <int>   <dbl>     <int>     <dbl>
1 Reunion  Africa     2007    76.4    798094     7670.
2 Canada   Americas   2007    80.7   33390141    36319.
3 Japan    Asia       2007    82.6  127467972    31656.
4 Iceland  Europe    2007    81.8    301931    36181.
5 Australia Oceania   2007    81.2  20434176    34435.
```

In this example, for each continent, we found the year in which life expectancy was highest over the last 50 years. In Africa, for example, the highest life expectancy ever measured was in Reunion in 2007.

14.3 Split > apply > combine for running many models

Now that you have an idea of what the split > apply > combine does, let's use it to run many models. And let's run the same linear model we just created for each country. From the results of these models, we can get an idea of where life expectancy is not following a linear trend.

Here is an example of how that might work. I will explain the code in a second.

```
(test_statistics <- gapminder %>%
  split(.country) %>%
  map_dfr(\(x) lm(lifeExp ~ year, .x) %>% broom::glance()))

# A tibble: 142 x 12
  r.squared adj.r.squ~1 sigma stati~2 p.value      df logLik     AIC     BIC devia~3
    <dbl>        <dbl> <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>
1   0.948       0.942  1.22    181.  9.84e- 8     1 -18.3   42.7   44.1   15.0
2   0.911       0.902  1.98    102.  1.46e- 6     1 -24.1   54.3   55.8   39.3
3   0.985       0.984  1.32    662.  1.81e-10    1 -19.3   44.6   46.0   17.5
4   0.888       0.877  1.41    79.1  4.59e- 6     1 -20.0   46.1   47.5   19.8
5   0.996       0.995  0.292   2246. 4.22e-13    1 -1.17   8.35   9.80   0.854
6   0.980       0.978  0.621   481.  8.67e-10    1 -10.2   26.4   27.9   3.85
7   0.992       0.991  0.407   1261. 7.44e-12    1 -5.16   16.3   17.8   1.66
8   0.967       0.963  1.64    291.  1.02e- 8     1 -21.9   49.7   51.2   26.9
9   0.989       0.988  0.977   930.  3.37e-11    1 -15.7   37.3   38.8   9.54
10  0.995       0.994  0.293   1822. 1.20e-12    1 -1.20   8.40   9.85   0.858
# ... with 132 more rows, 2 more variables: df.residual <int>, nobs <int>, and
#   abbreviated variable names 1: adj.r.squared, 2: statistic, 3: deviance
```

First, we use the `split` function from base R to split the data frame into a list. Each list element contains a data frame of one country:

```
list_of_countries <- gapminder %>%
  split(.country)

list_of_countries %>% length
```

```
[1] 142
```

The list contains 142 elements, which corresponds to the number of countries in the data frame:

```
gapminder$country %>% unique() %>% length()
```

```
[1] 142
```

Next, we apply two functions to each element of the list. We loop over the data frame of each country with `map_dfr` because we know that the output of the two functions will be a data frame (hence `dfr`). First we run a linear model for each list element. Perhaps you have stumbled across this code:

```
\(.x) lm(lifeExp ~ year, .x)
```

`\(.x)` is a shorthand option for an anonymous function in R. It was introduced with R 4.1.0 ([Keith McNulty](#) wrote a nice blog post about this).

Here is an simple example:

```
(\(.x) paste(.x, "loves R"))("Christian")
```

```
[1] "Christian loves R"
```

In our case the anonymous function returns the model object. This is how it looks for the first country in our list:

```
list_of_countries[[1]] %>%
  {lm(lifeExp ~ year, .)}
```

Call:

```
lm(formula = lifeExp ~ year, data = .)
```

Coefficients:

(Intercept)	year
-507.5343	0.2753

With `glance` we extract the test statistics from the model object:

```

list_of_countries[[1]] %>%
  {lm(lifeExp ~ year, .)} %>%
  glance()

# A tibble: 1 x 12
r.squ~1 adj.r~2 sigma stati~3 p.value      df logLik    AIC    BIC devia~4 df.re~5
<dbl>   <dbl> <dbl>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>   <int>
1  0.948   0.942  1.22    181.  9.84e-8     1  -18.3  42.7  44.1   15.0     10
# ... with 1 more variable: nobs <int>, and abbreviated variable names
#   1: r.squared, 2: adj.r.squared, 3: statistic, 4: deviance, 5: df.residual

```

Since this gives us a data frame, `map_dfr` can combine the results into a single data frame:

```

test_statistics

# A tibble: 142 x 12
r.squared adj.r.squ~1 sigma stati~2 p.value      df logLik    AIC    BIC devia~3
<dbl>       <dbl> <dbl>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>   <dbl>
1  0.948     0.942  1.22    181.  9.84e-8     1  -18.3  42.7  44.1   15.0
2  0.911     0.902  1.98    102.  1.46e-6     1  -24.1  54.3  55.8   39.3
3  0.985     0.984  1.32    662.  1.81e-10    1  -19.3  44.6  46.0   17.5
4  0.888     0.877  1.41    79.1  4.59e-6     1  -20.0  46.1  47.5   19.8
5  0.996     0.995  0.292   2246. 4.22e-13    1  -1.17  8.35  9.80   0.854
6  0.980     0.978  0.621   481.  8.67e-10    1  -10.2  26.4  27.9   3.85
7  0.992     0.991  0.407   1261. 7.44e-12    1  -5.16  16.3  17.8   1.66
8  0.967     0.963  1.64    291.  1.02e-8     1  -21.9  49.7  51.2   26.9
9  0.989     0.988  0.977   930.  3.37e-11    1  -15.7  37.3  38.8   9.54
10 0.995     0.994  0.293   1822. 1.20e-12    1  -1.20  8.40  9.85   0.858
# ... with 132 more rows, 2 more variables: df.residual <int>, nobs <int>, and
#   abbreviated variable names 1: adj.r.squared, 2: statistic, 3: deviance

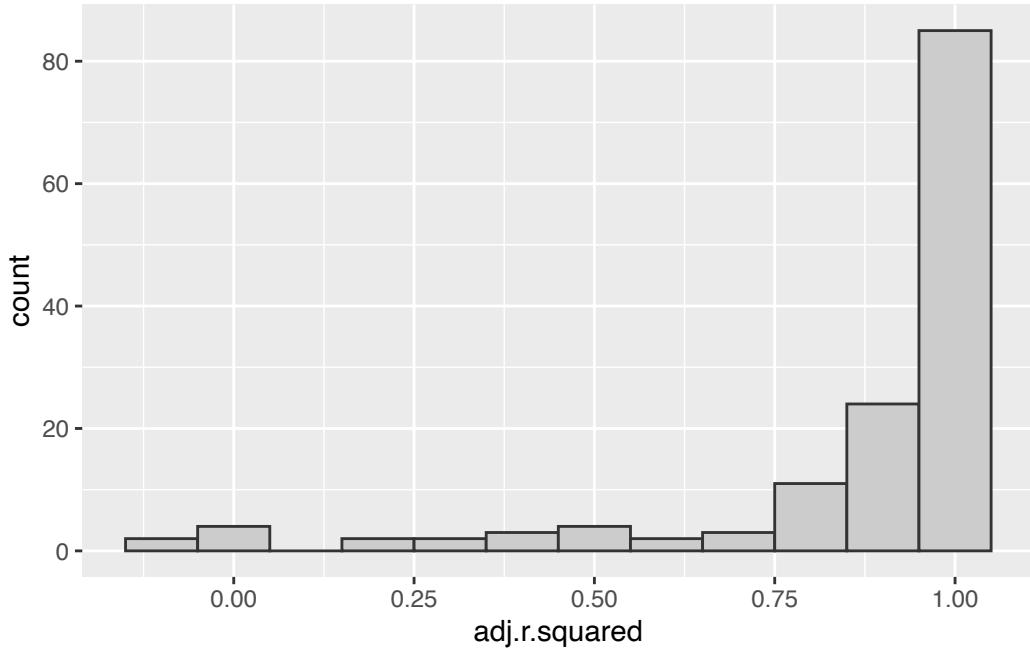
```

Now that we have run the same model for each country, we can see how well a linear model fits our data by looking at the adjusted R-squared:

```

test_statistics %>%
  ggplot(aes(x = adj.r.squared)) +
  geom_histogram(binwidth = .1, fill = "grey80", color = "gray20")

```



Looking at the histogram, most models appear to follow a linear trend (as many countries have an r-squared of $> .8$), but in a few countries, life expectancy clearly did not follow a linear trend.

Unfortunately, the data frame no longer contains the countries and continents, so we cannot extract the countries that appear to be outliers of our linear trend. Fortunately, there are many other ways to achieve the same results. I call it the Wild West of `split > apply > combine` for running many models.

14.4 The Wild West of `split > apply > combine` for running many models

It turns out that there is not just one way to run many models with the Tidyverse, but many. Here's an overview of the options I found. We'll go through them in more detail in a minute.

ID	Split type	split	apply	combine
1	Groups	<code>purrr::group_by</code>	<code>dplyr::group_map</code>	<code>purrr::map_dfr</code>
2	Groups	<code>purrr::group_by</code>	<code>dplyr::group_modify</code>	<code>dplyr::ungroup</code>
3	Lists	<code>base::split</code>	<code>purrr::map_dfr</code>	-
4	Lists	<code>base::split</code>	<code>purrr::map2_dfr</code>	-
5	Lists	<code>dplyr::group_split</code>	<code>purrr::map_dfr</code>	-

ID	Split type	split	apply	combine
6	Nested data	dplyr::group_nest	dplyr::mutate + purrr::map	tidy::unnest

First of all, we can differentiate between the three split types:

- *group*: When I talk about groups, I mean that the data is in the form of a [grouped tibble](#).
- *lists*: By lists I mean the native [list data type](#)
- *nested_data*: By nested data, I mean nested data frames in which [columns of a data frame contain lists or data frames](#).

Some methods combine the apply > combine step into one function (split > map_dfr and split -> map2_dfr). Therefore I added a hyphen (-) for the combine phase.

All methods have in common that a data frame comes in and a data frame goes out. What kind of data frame is returned depends on what we do in the apply phase. In our case, we chose to output the test statistics or parameters of our models. Similarly, we could also output the predictions of our models.

We will go through each of these examples next. Some of these examples use functions that are still in the experimental stage (for example, group_modify). So keep in mind that these functions may not be available forever. Either way, it's a good exercise to get familiar with the new grouping functions in dplyr.

14.5 group_by > group_map > map_dfr

Our first method is using grouped data. This is how it looks like:

```
gapminder %>%
  group_by(country) %>%
  group_map(
    .data =.,
    .f    = ~ lm(lifeExp ~ year, data = .) %>% glance()
  ) %>%
  map_dfr(~ .)

# A tibble: 142 x 12
#> # ... with 12 variables:
#> #   r.squared     adj.r.squared sigma    stati~2 p.value      df logLik     AIC     BIC devia~3
#> #   <dbl>        <dbl>     <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>
#> 1     0.948       0.942     1.22     181.  9.84e- 8      1 -18.3     42.7    44.1    15.0
```

```

2      0.911      0.902 1.98      102.  1.46e- 6      1 -24.1  54.3  55.8  39.3
3      0.985      0.984 1.32      662.  1.81e-10     1 -19.3  44.6  46.0  17.5
4      0.888      0.877 1.41      79.1 4.59e- 6      1 -20.0  46.1  47.5  19.8
5      0.996      0.995 0.292    2246.  4.22e-13     1 -1.17   8.35   9.80   0.854
6      0.980      0.978 0.621     481.  8.67e-10     1 -10.2   26.4   27.9   3.85
7      0.992      0.991 0.407    1261.  7.44e-12     1 -5.16   16.3   17.8   1.66
8      0.967      0.963 1.64      291.  1.02e- 8      1 -21.9   49.7   51.2   26.9
9      0.989      0.988 0.977     930.  3.37e-11     1 -15.7   37.3   38.8   9.54
10     0.995      0.994 0.293    1822.  1.20e-12     1 -1.20   8.40   9.85   0.858
# ... with 132 more rows, 2 more variables: df.residual <int>, nobs <int>, and
#   abbreviated variable names 1: adj.r.squared, 2: statistic, 3: deviance

```

As you can see, the resulting data frame does not contain a country or continent column. The first interesting bit in the code is the `group_map` function. `group_map` function takes a grouped tibble as input and outputs a list. We can see the list if we run the code without `map_dfr`:

```

gapminder %>%
  group_by(country) %>%
  group_map(
    .data = .,
    .f    = ~ lm(lifeExp ~ year, data = .)
  ) %>%
  head(n = 2)

```

[[1]]

```

Call:
lm(formula = lifeExp ~ year, data = .)

Coefficients:
(Intercept)      year
-507.5343       0.2753

```

[[2]]

```

Call:
lm(formula = lifeExp ~ year, data = .)

Coefficients:
(Intercept)      year
-594.0725       0.3347

```

Before this function, we could not simply iterate a function over groups in a grouped tibble. Instead, we split data frames into a list and applied purrr functions to the elements of that list (e.g., `map`).

If you like, `group_map` is the dplyr version of purrr's `map` functions.

Now that we have extracted the object of each model we can use `glance` to get the test statistics of these models:

```
gapminder %>%
  group_by(country) %>%
  group_map(
    .data =.,
    .f     = ~ lm(lifeExp ~ year, data = .) %>%
      glance()
  ) %>%
  head(n = 2)

[[1]]
# A tibble: 1 x 12
  r.squ~1 adj.r~2 sigma stati~3 p.value    df logLik   AIC   BIC devia~4 df.re~5
  <dbl>    <dbl> <dbl> <dbl>    <dbl> <dbl> <dbl> <dbl> <dbl>    <dbl>    <int>
1  0.948    0.942  1.22   181.  9.84e-8     1  -18.3  42.7  44.1   15.0     10
# ... with 1 more variable: nobs <int>, and abbreviated variable names
#   1: r.squared, 2: adj.r.squared, 3: statistic, 4: deviance, 5: df.residual

[[2]]
# A tibble: 1 x 12
  r.squ~1 adj.r~2 sigma stati~3 p.value    df logLik   AIC   BIC devia~4 df.re~5
  <dbl>    <dbl> <dbl> <dbl>    <dbl> <dbl> <dbl> <dbl> <dbl>    <dbl>    <int>
1  0.911    0.902  1.98   102.  1.46e-6     1  -24.1  54.3  55.8   39.3     10
# ... with 1 more variable: nobs <int>, and abbreviated variable names
#   1: r.squared, 2: adj.r.squared, 3: statistic, 4: deviance, 5: df.residual
```

Since this gives us a list of data frames, we need to combine them with `map_dfr(~ .)`.

```
gapminder %>%
  group_by(country) %>%
  group_map(
    .data =.,
    .f     = ~ lm(lifeExp ~ year, data = .) %>% glance()
  ) %>%
```

```

map_dfr(~ .)

# A tibble: 142 x 12
   r.squared adj.r.squ~1 sigma stati~2 p.value     df logLik    AIC    BIC devia~3
       <dbl>        <dbl> <dbl>      <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1     0.948        0.942 1.22     181.  9.84e- 8     1 -18.3  42.7  44.1  15.0
2     0.911        0.902 1.98     102.  1.46e- 6     1 -24.1  54.3  55.8  39.3
3     0.985        0.984 1.32     662.  1.81e-10    1 -19.3  44.6  46.0  17.5
4     0.888        0.877 1.41      79.1  4.59e- 6     1 -20.0  46.1  47.5  19.8
5     0.996        0.995 0.292    2246.  4.22e-13    1 -1.17  8.35  9.80  0.854
6     0.980        0.978 0.621    481.  8.67e-10    1 -10.2  26.4  27.9  3.85
7     0.992        0.991 0.407   1261.  7.44e-12    1 -5.16  16.3  17.8  1.66
8     0.967        0.963 1.64     291.  1.02e- 8     1 -21.9  49.7  51.2  26.9
9     0.989        0.988 0.977    930.  3.37e-11    1 -15.7  37.3  38.8  9.54
10    0.995        0.994 0.293   1822.  1.20e-12    1 -1.20  8.40  9.85  0.858
# ... with 132 more rows, 2 more variables: df.residual <int>, nobs <int>, and
#   abbreviated variable names 1: adj.r.squared, 2: statistic, 3: deviance

```

14.6 group_by > group_modify > ungroup

In our next example we use the new function `group_modify`. Compared to `group_map`, this function also accepts a grouped tibble, but outputs a grouped tibble instead of a list. With `group_modify` you have to make sure that a data frame is returned, otherwise the function will yield an error. This is what it looks like:

```

(method_two_results <- gapminder %>%
  group_by(country, continent) %>%
  group_modify(
    .data =.,
    .f     = ~ lm(lifeExp ~ year, data = .) %>% glance
  ) %>%
  ungroup()

# A tibble: 142 x 14
  country  conti~1 r.squ~2 adj.r~3 sigma stati~4 p.value     df logLik    AIC
  <fct>    <fct>    <dbl>    <dbl> <dbl>      <dbl> <dbl> <dbl> <dbl> <dbl>
1 Afghanistan Asia      0.948    0.942 1.22     181.  9.84e- 8     1 -18.3  42.7
2 Albania      Europe    0.911    0.902 1.98     102.  1.46e- 6     1 -24.1  54.3
3 Algeria      Africa    0.985    0.984 1.32     662.  1.81e-10    1 -19.3  44.6

```

```

4 Angola      Africa    0.888  0.877 1.41      79.1 4.59e- 6     1 -20.0  46.1
5 Argentina   Americ~  0.996  0.995 0.292    2246. 4.22e-13     1 -1.17  8.35
6 Australia   Oceania   0.980  0.978 0.621    481. 8.67e-10     1 -10.2  26.4
7 Austria     Europe    0.992  0.991 0.407    1261. 7.44e-12     1 -5.16  16.3
8 Bahrain     Asia      0.967  0.963 1.64      291. 1.02e- 8     1 -21.9  49.7
9 Bangladesh  Asia      0.989  0.988 0.977    930. 3.37e-11     1 -15.7  37.3
10 Belgium    Europe   0.995  0.994 0.293   1822. 1.20e-12     1 -1.20  8.40
# ... with 132 more rows, 4 more variables: BIC <dbl>, deviance <dbl>,
#   df.residual <int>, nobs <int>, and abbreviated variable names 1: continent,
#   2: r.squared, 3: adj.r.squared, 4: statistic

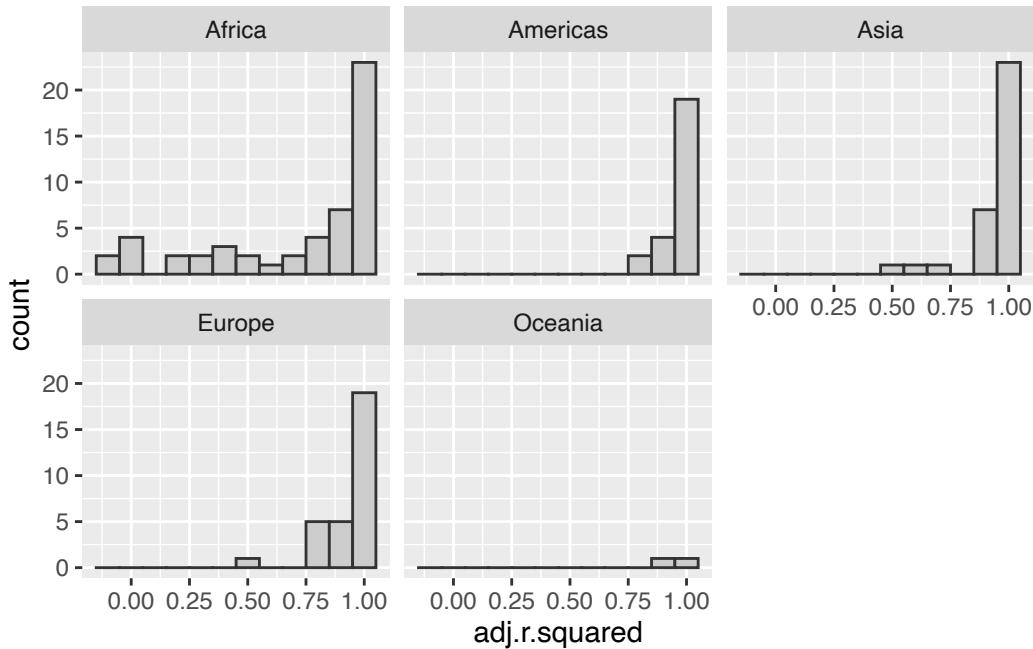
```

The great advantage of this method is that we can keep the country and continent as columns. With these results, we can compare, for example, the adjusted R-squared between the five continents:

```

method_two_results %>%
  ggplot(aes(x = adj.r.squared)) +
  geom_histogram(binwidth = .1, fill = "grey80", color = "gray20") +
  facet_wrap(vars(continent))

```



The results clearly show that our linear models perform least well in African countries (since these countries have small R-squared values).

A look at the data shows us that the countries with the worst fit are Rwanda, Botswana, Zimbabwe, Zambia and Swaziland:

```
method_two_results %>%
  slice_min(adj.r.squared, n = 5)

# A tibble: 5 x 14
  country continent r.squared adj.r.squared sigma statistic p.value    df logLik     AIC     BIC
  <fct>   <fct>        <dbl>      <dbl>    <dbl>      <dbl>    <dbl>    <dbl>    <dbl>    <dbl>
1 Rwanda  Africa       0.0172   -0.0811    6.56     0.175    0.685     1   -38.5    83.0    84.5
2 Botswa~ Africa       0.0340   -0.0626    6.11     0.352    0.566     1   -37.7    81.3    82.8
3 Zimbab~ Africa       0.0562   -0.0381    7.21     0.596    0.458     1   -39.6    85.3    86.7
4 Zambia  Africa       0.0598   -0.0342    4.53     0.636    0.444     1   -34.1    74.1    75.6
5 Swazil~ Africa       0.0682   -0.0250    6.64     0.732    0.412     1   -38.7    83.3    84.8
# ... with 3 more variables: deviance <dbl>, df.residual <int>, nobs <int>, and
#   abbreviated variable names 1: continent, 2: r.squared, 3: adj.r.squared,
#   4: statistic
```

The beauty of keeping the country and continent columns is that we can now compare the fit of our model to the actual life expectancies in these countries.

To get this data, we can use the `augment` function from the broom package. For each country, the function returns the fitted values of our dependent variable (in our case life expectancy), the residuals and the actual values of the dependent variable:

```
(augmented_data <- gapminder %>%
  group_by(country, continent) %>%
  group_modify(
    .data =.,
    .f    = ~ lm(lifeExp ~ year, data = .) %>% augment()
  ) %>%
  ungroup())

# A tibble: 1,704 x 10
  country continent lifeExp year .fitted .resid   .hat .sigma .cooksdi .std..2
  <fct>   <fct>     <dbl> <int>    <dbl>    <dbl>   <dbl>   <dbl>    <dbl>
1 Afghani~ Asia      28.8  1952     29.9 -1.11    0.295    1.21 2.43e-1 -1.08
2 Afghani~ Asia      30.3  1957     31.3 -0.952   0.225    1.24 1.13e-1 -0.884
3 Afghani~ Asia      32.0  1962     32.7 -0.664   0.169    1.27 3.60e-2 -0.595
4 Afghani~ Asia      34.0  1967     34.0 -0.0172  0.127    1.29 1.65e-5 -0.0151
5 Afghani~ Asia      36.1  1972     35.4  0.674   0.0991   1.27 1.85e-2  0.581
```

```

6 Afghanis~ Asia      38.4  1977     36.8  1.65   0.0851   1.15  9.23e-2  1.41
7 Afghanis~ Asia      39.9  1982     38.2  1.69   0.0851   1.15  9.67e-2  1.44
8 Afghanis~ Asia      40.8  1987     39.5  1.28   0.0991   1.21  6.67e-2  1.10
9 Afghanis~ Asia      41.7  1992     40.9  0.754  0.127    1.26  3.17e-2  0.660
10 Afghanis~ Asia     41.8  1997     42.3  -0.534 0.169    1.27  2.33e-2 -0.479
# ... with 1,694 more rows, and abbreviated variable names 1: continent,
#   2: .std.resid

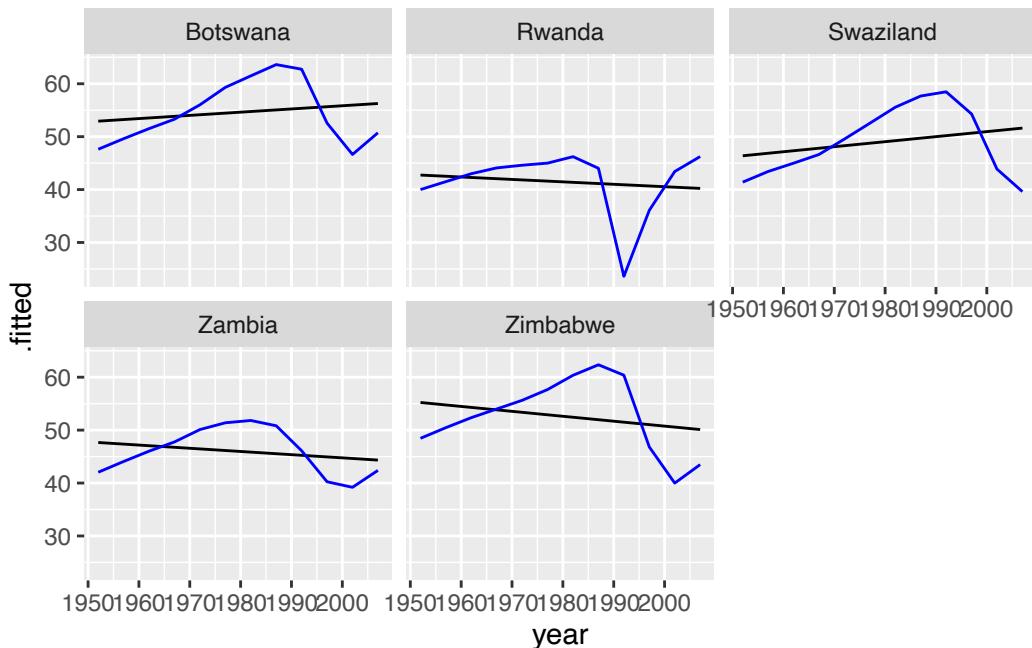
```

Now we are ready to compare our regression models for these five countries with the actual data:

```

augmented_data %>%
  filter(country %in% (
    method_two_results %>% slice_min(adj.r.squared, n = 5) %>%
    pull(country)
  )) %>%
  ggplot(aes(x = year, y = .fitted)) +
  geom_line() +
  geom_line(aes(y = .fitted + .resid), color = "blue") +
  facet_wrap(vars(country))

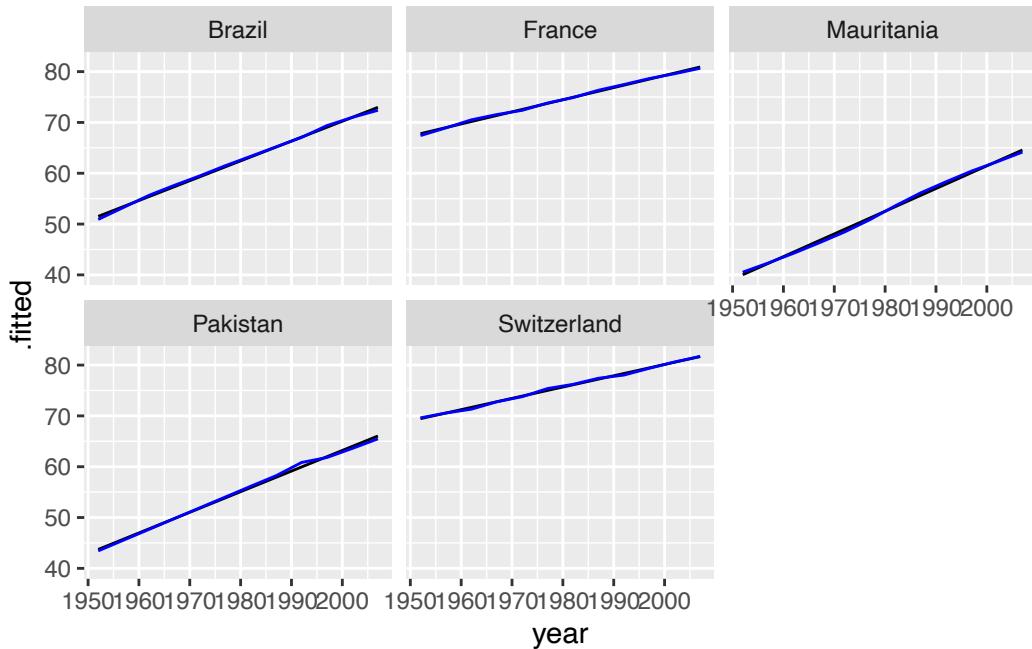
```



The blue line represents the actual data and the black line our fitted regression model.

Similarly, we can look at the countries with the best fit (`slice_max` instead of `slice_min`):

```
augmented_data %>%
  filter(country %in% (
    method_two_results %>% slice_max(adj.r.squared, n = 5) %>%
      pull(country)
  )) %>%
  ggplot(aes(x = year, y = .fitted)) +
  geom_line() +
  geom_line(aes(y = .fitted + .resid), color = "blue") +
  facet_wrap(vars(country))
```



14.7 `split > map2_dfr`

We have already covered on how to run many models using `split` and `map_dfr`. However, we have seen that this method does not preserve the country column. However, there is a trick to preserve it. And the trick is the function `map2_dfr`.

You may know that the function `names` returns the names of list objects:

```

my_list <- list(
  Afghanistan = c(1, 2, 3),
  Germany     = 3
)

```

```
my_list %>% names()
```

```
[1] "Afghanistan" "Germany"
```

You may also know that `map2_dfr` allows us to iterate over two arguments simultaneously. By combining both functions we can add a new column to the data frame returned by the `tidy`, `glance` or `augment` function, representing the country column:

```

gapminder %>%
  split(.country) %>%
  map2_dfr(
    .x = .,
    .y = names(.),
    .f = ~ lm(lifeExp ~ year, data = .x) %>%
      tidy() %>% mutate(country = .y)
  )

```

```

# A tibble: 284 x 6
  term      estimate std.error statistic p.value country
  <chr>      <dbl>     <dbl>     <dbl>    <dbl> <chr>
1 (Intercept) -508.     40.5      -12.5   1.93e- 7 Afghanistan
2 year         0.275     0.0205     13.5   9.84e- 8 Afghanistan
3 (Intercept) -594.     65.7      -9.05   3.94e- 6 Albania
4 year         0.335     0.0332     10.1   1.46e- 6 Albania
5 (Intercept) -1068.    43.8      -24.4   3.07e-10 Algeria
6 year         0.569     0.0221     25.7   1.81e-10 Algeria
7 (Intercept) -377.     46.6      -8.08   1.08e- 5 Angola
8 year         0.209     0.0235     8.90   4.59e- 6 Angola
9 (Intercept) -390.     9.68      -40.3   2.14e-12 Argentina
10 year        0.232     0.00489    47.4   4.22e-13 Argentina
# ... with 274 more rows

```

With this method we keep the country name but not the continent column since we have split the data frame into a one-dimensional list.

However, we can use another trick to keep both columns. We know that the data frame we split still contains the country and continent columns. We also know that both columns contain the same values per column. We can therefore add the country and continent to the data frame returned by the `tidy` function. Also we don't even need `map2_dfr` and can use `map_dfr` instead.

```
gapminder %>%
  split(.country) %>%
  map_dfr(
    .x = .,
    .f = ~ lm(lifeExp ~ year, data = .x) %>%
      tidy() %>%
      mutate(
        country = .x$country[1],
        continent = .x$continent[1])
  )

# A tibble: 284 x 7
  term      estimate std.error statistic p.value country   continent
  <chr>     <dbl>     <dbl>     <dbl>    <dbl> <fct>     <fct>
1 (Intercept) -508.     40.5      -12.5  1.93e- 7 Afghanistan Asia
2 year         0.275     0.0205     13.5  9.84e- 8 Afghanistan Asia
3 (Intercept) -594.     65.7      -9.05  3.94e- 6 Albania    Europe
4 year         0.335     0.0332     10.1  1.46e- 6 Albania    Europe
5 (Intercept) -1068.    43.8      -24.4  3.07e-10 Algeria   Africa
6 year         0.569     0.0221     25.7  1.81e-10 Algeria   Africa
7 (Intercept) -377.     46.6      -8.08  1.08e- 5 Angola    Africa
8 year         0.209     0.0235     8.90  4.59e- 6 Angola    Africa
9 (Intercept) -390.     9.68      -40.3  2.14e-12 Argentina Americas
10 year        0.232     0.00489    47.4  4.22e-13 Argentina Americas
# ... with 274 more rows
```

14.8 `group_split` > `map_dfr`

The combination of `group_split` and `map_dfr` is just an alternative version of `split` and `map_dfr`. `group_split` was introduced in dplyr for version 0.8.0 in 2019. It is still in an experimental state. The difference with `split` is that `group_split` does not name the elements of the returned list. Other than that, I don't see a good use case for using `group_split` over `split` for our use case.

Here is how it works:

```

gapminder %>%
  group_split(country) %>%
  map_dfr(
    .x = .,
    .f = ~ lm(lifeExp ~ year, data = .x) %>%
      tidy() %>%
      mutate(
        country = .x$country[1],
        continent = .x$continent[1])
  )

```

A tibble: 284 x 7

	term	estimate	std.error	statistic	p.value	country	continent
	<chr>	<dbl>	<dbl>	<dbl>	<dbl>	<fct>	<fct>
1	(Intercept)	-508.	40.5	-12.5	1.93e- 7	Afghanistan	Asia
2	year	0.275	0.0205	13.5	9.84e- 8	Afghanistan	Asia
3	(Intercept)	-594.	65.7	-9.05	3.94e- 6	Albania	Europe
4	year	0.335	0.0332	10.1	1.46e- 6	Albania	Europe
5	(Intercept)	-1068.	43.8	-24.4	3.07e-10	Algeria	Africa
6	year	0.569	0.0221	25.7	1.81e-10	Algeria	Africa
7	(Intercept)	-377.	46.6	-8.08	1.08e- 5	Angola	Africa
8	year	0.209	0.0235	8.90	4.59e- 6	Angola	Africa
9	(Intercept)	-390.	9.68	-40.3	2.14e-12	Argentina	Americas
10	year	0.232	0.00489	47.4	4.22e-13	Argentina	Americas
	# ... with 274 more rows						

14.9 `group_nest > mutate/map > unnest`

At last we have the combination of `group_nest`, `mutate` and `map` and `unnest`. I'll show you the code first and we'll go through it afterwards:

```

gapminder %>%
  group_nest(continent, country) %>%
  mutate(
    model = map(data, ~ lm(lifeExp ~ year, data = .)) %>%
      tidy()
  ) %>%
  select(-data) %>%
  unnest(model)

```

```
# A tibble: 284 x 7
  continent country     term    estimate std.error statistic p.value
  <fct>    <fct>    <chr>      <dbl>     <dbl>      <dbl>     <dbl>
1 Africa    Algeria (Intercept) -1068.     43.8     -24.4 3.07e-10
2 Africa    Algeria   year       0.569     0.0221    25.7 1.81e-10
3 Africa    Angola   (Intercept) -377.      46.6     -8.08 1.08e- 5
4 Africa    Angola    year       0.209     0.0235    8.90 4.59e- 6
5 Africa    Benin    (Intercept) -613.      38.9     -15.8 2.18e- 8
6 Africa    Benin     year       0.334     0.0196   17.0 1.04e- 8
7 Africa    Botswana (Intercept) -65.5     202.     -0.324 7.53e- 1
8 Africa    Botswana   year      0.0607    0.102     0.593 5.66e- 1
9 Africa    Burkina Faso (Intercept) -676.      67.8     -9.97 1.63e- 6
10 Africa   Burkina Faso  year      0.364     0.0342   10.6 9.05e- 7
# ... with 274 more rows
```

Let's break it down. The `group_nest` function was also introduced with the dplyr version 0.8.0 and works pretty similar to `nest`. Compared to `nest`, with `group_nest` you name the columns that should *not* be nested instead of the columns that should be nested.

```
gapminder %>%
  group_nest(continent, country)
```

```
# A tibble: 142 x 3
  continent country                               data
  <fct>    <fct>    <list<tibble[,4]>>
1 Africa    Algeria [12 x 4]
2 Africa    Angola [12 x 4]
3 Africa    Benin [12 x 4]
4 Africa    Botswana [12 x 4]
5 Africa    Burkina Faso [12 x 4]
6 Africa    Burundi [12 x 4]
7 Africa    Cameroon [12 x 4]
8 Africa    Central African Republic [12 x 4]
9 Africa    Chad [12 x 4]
10 Africa   Comoros [12 x 4]
# ... with 132 more rows
```

Next, we create a new column that contains our model results:

```
(model_results_nested <- gapminder %>%
  group_nest(continent, country) %>%
```

```

    mutate(
      model = map(data, ~ lm(lifeExp ~ year, data = .) %>%
        tidy())
    ))
```

A tibble: 142 x 4

	continent	country	data model
<fct>	<fct>		<list<tibble[,4]>> <list>
1	Africa	Algeria	[12 x 4] <tibble [2 x 5]>
2	Africa	Angola	[12 x 4] <tibble [2 x 5]>
3	Africa	Benin	[12 x 4] <tibble [2 x 5]>
4	Africa	Botswana	[12 x 4] <tibble [2 x 5]>
5	Africa	Burkina Faso	[12 x 4] <tibble [2 x 5]>
6	Africa	Burundi	[12 x 4] <tibble [2 x 5]>
7	Africa	Cameroon	[12 x 4] <tibble [2 x 5]>
8	Africa	Central African Republic	[12 x 4] <tibble [2 x 5]>
9	Africa	Chad	[12 x 4] <tibble [2 x 5]>
10	Africa	Comoros	[12 x 4] <tibble [2 x 5]>
# ... with 132 more rows			

Each value of the model column contains the results of the `tidy` function. For example, the model results for Algeria:

```
model_results_nested$model[[1]]
```

```
# A tibble: 2 x 5
  term       estimate std.error statistic p.value
  <chr>     <dbl>     <dbl>     <dbl>    <dbl>
1 (Intercept) -1068.      43.8     -24.4 3.07e-10
2 year         0.569     0.0221    25.7 1.81e-10
```

A nice feature of this method is that we can store the results of the `tidy` and `glance` functions in one data frame:

```
gapminder %>%
  group_nest(continent, country) %>%
  mutate(
    model           = map(data, ~ lm(lifeExp ~ year, data = .)),
    model_parameters = map(model, broom::tidy),
    model_test_statistics = map(model, broom::glance)
```

```
)
```

```
# A tibble: 142 x 6
  continent country
  <fct>    <fct>
  1 Africa   Algeria
  2 Africa   Angola
  3 Africa   Benin
  4 Africa   Botswana
  5 Africa   Burkina Faso
  6 Africa   Burundi
  7 Africa   Cameroon
  8 Africa   Central African Republic
  9 Africa   Chad
 10 Africa  Comoros
# ... with 132 more rows, and abbreviated variable names
#   1: model_parameters,
#   2: model_test_statistics
```

data model model_~1 model_~2
<list<tibble[,4]>> <lis> <list> <list>
[12 x 4] <lm> <tibble> <tibble>
[12 x 4] <lm> <tibble> <tibble>

Once we have this data frame, we can use `unnest` and `select` to unpack the results of our models:

```
gapminder %>%
  group_nest(continent, country) %>%
  mutate(
    model           = map(data, ~ lm(lifeExp ~ year, data = .)),
    model_parameters = map(model, broom::tidy),
    model_test_statistics = map(model, broom::glance)
  ) %>%
  select(-model, -model_parameters, -data) %>%
  unnest(model_test_statistics)
```

```
# A tibble: 142 x 14
  continent country  r.squ~1 adj.r~2 sigma stati~3 p.value      df logLik     AIC
  <fct>    <fct>    <dbl>    <dbl>  <dbl>    <dbl>    <dbl> <dbl>    <dbl> <dbl>
  1 Africa   Algeria  0.985    0.984  1.32  6.62e+2 1.81e-10    1 -19.3   44.6
  2 Africa   Angola   0.888    0.877  1.41  7.91e+1 4.59e- 6    1 -20.0   46.1
  3 Africa   Benin    0.967    0.963  1.17  2.89e+2 1.04e- 8    1 -17.9   41.7
  4 Africa   Botswana 0.0340  -0.0626 6.11  3.52e-1 5.66e- 1    1 -37.7   81.3
  5 Africa   Burkina ~ 0.919    0.911  2.05  1.13e+2 9.05e- 7    1 -24.5   55.1
  6 Africa   Burundi   0.766    0.743  1.61  3.27e+1 1.93e- 4    1 -21.7   49.3
  7 Africa   Cameroon  0.680    0.648  3.24  2.13e+1 9.63e- 4    1 -30.1   66.1
```

```

8 Africa    Central ~ 0.493   0.443   3.52   9.73e+0 1.09e- 2      1 -31.1   68.1
9 Africa    Chad       0.872   0.860   1.83   6.84e+1 8.82e- 6      1 -23.2   52.4
10 Africa   Comoros   0.997   0.997   0.479  3.17e+3 7.63e-14     1 -7.09   20.2
# ... with 132 more rows, 4 more variables: BIC <dbl>, deviance <dbl>,
#   df.residual <int>, nobs <int>, and abbreviated variable names 1: r.squared,
#   2: adj.r.squared, 3: statistic

```

14.10 Conclusion

We have seen that there are a few ways to run many models using a number of functions from the Tidyverse package. Some of these methods make it more difficult than others to extract the names of groups or splits. Time will tell which approach works best for most people. In any case, some methods look promising. I was particularly impressed with the combination of `group_by`, `group_modify` and `ungroup`. Keep in mind, however, that many of the new grouping functions introduced in dplyr in 2019 are still in the experimental stage. They may not be with us forever.

Summary

Here's what you can take away from this tutorial.

- There are three approaches to running many models. Those based on grouped tibbles, those based on lists and those based on nested data.
- The combination of `group_by`, `group_modify` and `ungroup` seems to be one of the most elegant ways to run many models.
- Many grouping functions that were introduced in dplyr in 2019 are still in an experimental stage and should be used with caution.

Part VI

Improve working with incomplete data

15 How to expand data frames and create complete combinations of values

What will this tutorial cover?

In this tutorial you will find out how to create complete sets of values using `complete`, `expand` and `crossing`. I will try to show you how these functions differ and for which use cases they are useful.

Who do I have to thank?

For this tutorial, I relied on [the official documentation](#). Kudos to the developers Hadley Wickham and Maximilian Girlich.

I've always found it difficult to distinguish the functions `complete`, `expand`, `nesting`, and `crossing` from another. In a sense, they do similar things. They find combinations of values in vectors or columns. I originally thought of writing a separate tutorial for each function, but digesting them all at once makes it easier to tell the difference between them. Let's take some time to look into these functions. And let's start with an overview of what they do:

Function	Explanation
<code>complete</code>	Turn implicit missing values into explicit values. The function completes combinations of values from columns that exist in a data frame and/or from vectors. <code>complete</code> is a shortcut version of <code>expand</code> .
<code>expand</code>	Creates a new tibble with all possible combinations of values from a data frame. The function is often used with joins.
<code>expand</code> with <code>nesting</code>	Create a new tibble with the unique combinations of column values that exist in a data frame.
<code>crossing</code>	Create a tibble with all combinations of values from vectors.

If you look at this table, you will notice a couple of things. First, `complete` makes an existing data frame longer by converting implicit values to existing values. This means that combinations of values that are not present in the data are created as new rows.

`expand`, in contrast, creates a new tibble. The tibble represents either all possible combinations of values or the unique combinations of values (with `nesting`). Suppose you have an incomplete tibble with months and years, in which the combination of the month “February” and the year 2013 is missing. You can use `expand` to create a complete set of years and months, including the combination of February and 2013. `crossing` works similarly to `expand`, except that it uses *vectors* to create combinations of values in a data frame. However, as we will see later, `expand` can also do this.

For this tutorial, we will use a made-up data set of running events. Suppose the following data frame shows the running races a runner has completed since 2010. The minutes show the time it took this person to complete the races. Let’s call her Anna.

```
running_races_anna<- tribble(
  ~year, ~race,           ~minutes,
  2010,   "half marathon", 110,
  2011,   "marathon",      230,
  2013,   "half marathon", 105,
  2016,   "10km",          50,
  2018,   "10km",          45,
  2018,   "half marathon", 100,
  2022,   "marathon",      210
)
```

You can see that some years are missing. Anna didn’t run a race in 2012. Also, she did not run a half marathon in 2016. In the next chapters, we will try to complete this data frame with the four functions. Let’s start with `complete`.

15.1 `complete`

Let’s assume Anna has only run 10Ks, half marathons, and marathons in recent years. Let’s further assume that she could have participated in every race each year. How many running races could she have participated in then?

A first approach could be to convert the implicit combinations into explicit combinations using `complete`. This essentially means nothing more than creating new rows representing the runs in which it did not participate. For these runs, `complete` sets the values of the `minutes` column to NA:

```
running_races_anna %>%
  complete(year, race)
```

```
# A tibble: 18 x 3
  year race      minutes
  <dbl> <chr>     <dbl>
1 2010 10km        NA
2 2010 half marathon 110
3 2010 marathon    NA
4 2011 10km        NA
5 2011 half marathon NA
6 2011 marathon    230
7 2013 10km        NA
8 2013 half marathon 105
9 2013 marathon    NA
10 2016 10km       50
11 2016 half marathon NA
12 2016 marathon   NA
13 2018 10km       45
14 2018 half marathon 100
15 2018 marathon   NA
16 2022 10km        NA
17 2022 half marathon NA
18 2022 marathon   210
```

Looking at the number of rows she could have participated in 18 competitions. But could she? You might see that we are missing some years. There is no data from 2012 or 2014. This is because `complete` only works with the values that are already in the data. Since she never participated in a race in 2012, we don't see these races.

However, we can add these values if we use vectors instead of columns. Suppose we want to ensure that the data frame includes all years between 2010 and 2022 and all three running events that are already present in the data:

```
running_races_anna %>%
  complete(year = 2010:2022, race)
```

```
# A tibble: 39 x 3
  year race      minutes
  <dbl> <chr>     <dbl>
1 2010 10km        NA
2 2010 half marathon 110
3 2010 marathon    NA
4 2011 10km        NA
5 2011 half marathon NA
```

```

6 2011 marathon      230
7 2012 10km          NA
8 2012 half marathon NA
9 2012 marathon      NA
10 2013 10km         NA
# ... with 29 more rows

```

All combinations of values that were already present in the data did not change. However, the code added rows that were not present. In other words, it added rows with years and races that were not present in the original data frame.

We could even go so far as to include new races to the data frame (i.e. ultra marathons):

```

running_races_anna %>%
  complete(year = 2010:2022, race = c(race, "ultra marathons"))

```

```

# A tibble: 52 x 3
  year   race       minutes
  <dbl> <chr>     <dbl>
1 2010 10km        NA
2 2010 half marathon 110
3 2010 marathon    NA
4 2010 ultra marathons NA
5 2011 10km        NA
6 2011 half marathon NA
7 2011 marathon    230
8 2011 ultra marathons NA
9 2012 10km        NA
10 2012 half marathon NA
# ... with 42 more rows

```

Look how we created a vector that includes the races already present in the data plus ultra marathons.

15.2 expand

The `expand` function does something very similar. However, instead of adding new rows with the complete set of values, a new data frame is created only for the columns you specify in the function (compared to `complete`, where we kept the `minutes` column).

First, let's create a simple example. Let's create a complete combination of years and races from the existing data frame:

```
running_races_anna %>%  
  expand(year, race)
```

```
# A tibble: 18 x 2  
  year   race  
  <dbl> <chr>  
1 2010  10km  
2 2010  half marathon  
3 2010  marathon  
4 2011  10km  
5 2011  half marathon  
6 2011  marathon  
7 2013  10km  
8 2013  half marathon  
9 2013  marathon  
10 2016 10km  
11 2016 half marathon  
12 2016 marathon  
13 2018 10km  
14 2018 half marathon  
15 2018 marathon  
16 2022 10km  
17 2022 half marathon  
18 2022 marathon
```

The result is a new data frame. You can see that the column `minutes` is missing. Similar to `complete` we can specify a vector instead of a column, for example to make sure that the data frame covers all years from 2010 to 2022:

```
running_races_anna %>%  
  expand(year = 2010:2022, race)
```

```
# A tibble: 39 x 2  
  year   race  
  <int> <chr>  
1 2010  10km  
2 2010  half marathon  
3 2010  marathon  
4 2011  10km  
5 2011  half marathon  
6 2011  marathon  
7 2013  10km  
8 2013  half marathon  
9 2013  marathon  
10 2016 10km  
11 2016 half marathon  
12 2016 marathon  
13 2018 10km  
14 2018 half marathon  
15 2018 marathon  
16 2020 10km  
17 2020 half marathon  
18 2020 marathon  
19 2020 marathon  
20 2021 10km  
21 2021 half marathon  
22 2021 marathon  
23 2022 10km  
24 2022 half marathon  
25 2022 marathon  
26 2022 marathon  
27 2022 marathon  
28 2022 marathon  
29 2022 marathon
```

```

4 2011 10km
5 2011 half marathon
6 2011 marathon
7 2012 10km
8 2012 half marathon
9 2012 marathon
10 2013 10km
# ... with 29 more rows

```

A neat trick to complete the years is the `full_seq` function:

```

running_races_anna %>%
  expand(year = full_seq(year, 1), race)

```

```

# A tibble: 39 x 2
  year   race
  <dbl> <chr>
1 2010 10km
2 2010 half marathon
3 2010 marathon
4 2011 10km
5 2011 half marathon
6 2011 marathon
7 2012 10km
8 2012 half marathon
9 2012 marathon
10 2013 10km
# ... with 29 more rows

```

In this case `full_seq` generated the complete set of years, starting with the lowest year in the data frame and ending with the highest year. The `1` indicates that the years should be incremented by 1 each time.

So we have a handle on all the combinations of years and races in our data frame. But we are missing the actual data, namely the minutes Anna took for these races. To add this data to the data frame, we combine `expand` with `full_join`:

```

(all_running_races_anna <- running_races_anna %>%
  expand(year = full_seq(year, 1), race) %>%
  full_join(running_races_anna, by = c("year", "race")))

```

```
# A tibble: 39 x 3
  year race      minutes
  <dbl> <chr>     <dbl>
1 2010 10km        NA
2 2010 half marathon 110
3 2010 marathon    NA
4 2011 10km        NA
5 2011 half marathon NA
6 2011 marathon    230
7 2012 10km        NA
8 2012 half marathon NA
9 2012 marathon    NA
10 2013 10km       NA
# ... with 29 more rows
```

This data frame includes all 39 races that Anna could have participated in between 2010 and 2022.

You may wonder why you should use `expand` instead of `complete` at all? The result is the same we got with `complete`. And the code it is more complicated.

If you take a look at the document, you will see that `complete` is actually a wrapper around `expand`. In other words, it is `expand` combined with `full_join` (see the [official code on GitHub](#)). Essentially, it is a shortcut for the more complicated code we just used. We will show this in the upcoming examples.

15.3 `expand/complete` with `group_by`

Now let's imagine Anna is running in a club with three other runners. Eva, John and Leonie.

```
running_races_club <- tribble(
  ~year, ~runner,   ~race,           ~minutes,
  2012, "Eva",     "half marathon", 109,
  2013, "Eva",     "marathon",      260,
  2022, "Eva",     "half marathon", 120,
  2018, "John",    "10km",          51,
  2019, "John",    "10km",          49,
  2020, "John",    "10km",          50,
  2019, "Leonie",  "half marathon", 45,
  2020, "Leonie",  "10km",          45,
  2021, "Leonie",  "half marathon", 102,
  2022, "Leonie",  "marathon",      220
```

```
)
```

Again, you want to find all races that each runner could have participated in since joining the club. If we used the same `expand` technique we just did, we will run into problems:

```
(all_running_races_club <- running_races_club %>%
  expand(year = full_seq(year, 1), race, runner))
```

```
# A tibble: 99 x 3
  year race      runner
  <dbl> <chr>    <chr>
1 2012 10km     Eva
2 2012 10km     John
3 2012 10km     Leonie
4 2012 half marathon Eva
5 2012 half marathon John
6 2012 half marathon Leonie
7 2012 marathon   Eva
8 2012 marathon   John
9 2012 marathon   Leonie
10 2013 10km    Eva
# ... with 89 more rows
```

Take John, for example:

```
all_running_races_club %>%
  filter(runner == "John")
```

```
# A tibble: 33 x 3
  year race      runner
  <dbl> <chr>    <chr>
1 2012 10km     John
2 2012 half marathon John
3 2012 marathon   John
4 2013 10km     John
5 2013 half marathon John
6 2013 marathon   John
7 2014 10km     John
8 2014 half marathon John
9 2014 marathon   John
10 2015 10km    John
```

```
# ... with 23 more rows
```

He joined the club in 2019. However, the data frame shows missed races from 2012. This is because the data frame contains the races of Eva, who joined in 2012.

We can fix this problem by grouping the data frame by runners.

```
(all_running_races_club_correct <- running_races_club %>%
  group_by(runner) %>%
  expand(year = full_seq(year, 1), race = c("10km", "half marathon",
                                             "marathon")) %>%
  ungroup())
```

```
# A tibble: 54 x 3
  runner   year race
  <chr>   <dbl> <chr>
1 Eva     2012 10km
2 Eva     2012 half marathon
3 Eva     2012 marathon
4 Eva     2013 10km
5 Eva     2013 half marathon
6 Eva     2013 marathon
7 Eva     2014 10km
8 Eva     2014 half marathon
9 Eva     2014 marathon
10 Eva    2015 10km
# ... with 44 more rows
```

With `group_by` we expand the rows only within the runners. If you now take a look at the data, you will notice that John has no races before 2018, which is exactly what we want.

```
all_running_races_club_correct %>%
  filter(runner == "John")
```

```
# A tibble: 9 x 3
  runner   year race
  <chr>   <dbl> <chr>
1 John     2018 10km
2 John     2018 half marathon
3 John     2018 marathon
4 John     2019 10km
```

```

5 John    2019 half marathon
6 John    2019 marathon
7 John    2020 10km
8 John    2020 half marathon
9 John    2020 marathon

```

Yet, we still need the actual data of the three runners. We use `left_join` to add the running times to the expanded data frame:

```
(complete_running_races_club <- all_running_races_club_correct %>%
  left_join(running_races_club, by = c("year", "runner", "race")))

# A tibble: 54 x 4
  runner  year race      minutes
  <chr>   <dbl> <chr>     <dbl>
1 Eva     2012 10km       NA
2 Eva     2012 half marathon 109
3 Eva     2012 marathon    NA
4 Eva     2013 10km       NA
5 Eva     2013 half marathon NA
6 Eva     2013 marathon    260
7 Eva     2014 10km       NA
8 Eva     2014 half marathon NA
9 Eva     2014 marathon    NA
10 Eva    2015 10km      NA
# ... with 44 more rows
```

Since we already know that `complete` is a shortcut for such an analysis, we can use it instead:

```
running_races_club %>%
  group_by(runner) %>%
  complete(year = full_seq(year, 1), race = c("10km", "half marathon",
                                              "marathon")) %>%
  ungroup()

# A tibble: 54 x 4
  runner  year race      minutes
  <chr>   <dbl> <chr>     <dbl>
1 Eva     2012 10km       NA
2 Eva     2012 half marathon 109
```

```

3 Eva      2012 marathon        NA
4 Eva      2013 10km           NA
5 Eva      2013 half marathon   NA
6 Eva      2013 marathon       260
7 Eva      2014 10km           NA
8 Eva      2014 half marathon  NA
9 Eva      2014 marathon       NA
10 Eva     2015 10km          NA
# ... with 44 more rows

```

With this data we can do some interesting analysis. We could visualize the percentage of competitions in which each runner actually participated.

First, we need to find out how many races each runner has completed. To do this, we count the number of races that a runner has or has not completed:

```
(count_races <- complete_running_races_club %>%
  count(runner, race, missed_races = is.na(minutes)))
```

```
# A tibble: 14 x 4
  runner race      missed_races     n
  <chr>  <chr>    <lgl>        <int>
1 Eva    10km     TRUE          11
2 Eva    half marathon FALSE         2
3 Eva    half marathon TRUE          9
4 Eva    marathon  FALSE         1
5 Eva    marathon  TRUE          10
6 John   10km     FALSE         3
7 John   half marathon TRUE          3
8 John   marathon  TRUE          3
9 Leonie 10km     FALSE         1
10 Leonie 10km    TRUE          3
11 Leonie half marathon FALSE         2
12 Leonie half marathon TRUE          2
13 Leonie marathon FALSE         1
14 Leonie marathon TRUE          3
```

We see that Eva has not completed a single 10-km run in the years she has been a member of the club, because there is a row missing where the `missed_races` column is set to `FALSE`.

Fortunately, we have learned that we can complete an existing data frame with `complete`. Let's do that:

```

count_races %>%
  complete(runner, race, missed_races, fill = list(n = 0))

# A tibble: 18 x 4
  runner race      missed_races     n
  <chr>  <chr>    <lgl>        <int>
1 Eva    10km     FALSE          0
2 Eva    10km     TRUE           11
3 Eva    half marathon FALSE         2
4 Eva    half marathon TRUE          9
5 Eva    marathon   FALSE         1
6 Eva    marathon   TRUE          10
7 John   10km     FALSE          3
8 John   10km     TRUE           0
9 John   half marathon FALSE         0
10 John  half marathon TRUE          3
11 John  marathon   FALSE         0
12 John  marathon   TRUE          3
13 Leonie 10km     FALSE         1
14 Leonie 10km     TRUE          3
15 Leonie half marathon FALSE         2
16 Leonie half marathon TRUE          2
17 Leonie marathon   FALSE         1
18 Leonie marathon   TRUE          3

```

The code has an interesting addition, the `fill` parameter. The parameter allows us turn NAs to actual values. Since we know that the missing rows represent the number of races that were or were not finish, we can be sure that they represent zero races. For Eva, for example, a row is missing for the 10km races in which she never participated.

Now that we have the complete count data of races per runner, we can calculate the percentage of races they participated in. To calculate the percentages, we must first put the data into a wide format and then create a column that represents the percentages:

```

count_races %>%
  complete(runner, race, missed_races, fill = list(n = 0)) %>%
  pivot_wider(names_from = missed_races, values_from = n) %>%
  mutate(
    percent_races = (`FALSE` / (`TRUE` + `FALSE`)) * 100
  )

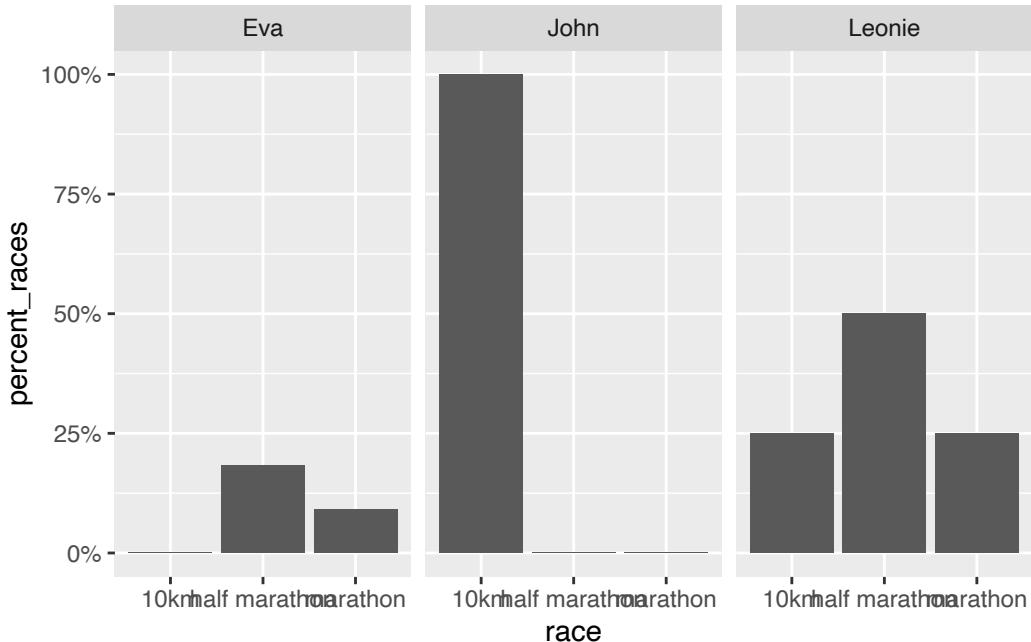
```

```
# A tibble: 9 x 5
  runner race      `FALSE` `TRUE` percent_races
  <chr>  <chr>     <int>   <int>      <dbl>
1 Eva    10km       0        11        0
2 Eva    half marathon 2        9        18.2
3 Eva    marathon    1        10        9.09
4 John   10km       3        0        100
5 John   half marathon 0        3        0
6 John   marathon    0        3        0
7 Leonie 10km       1        3        25
8 Leonie half marathon 2        2        50
9 Leonie marathon    1        3        25
```

Let's talk about Eva again. She participated in 0% of the 10k races and in 18.18% of the possible half marathons. Since she ran only 1 of 11 marathons, she participated in 9% of the marathons.

This is how it looks for all runners:

```
count_races %>%
  complete(runner, race, missed_races, fill = list(n = 0)) %>%
  pivot_wider(names_from = missed_races, values_from = n) %>%
  mutate(
    percent_races = (`FALSE` / (`TRUE` + `FALSE`)) * 100
  ) %>%
  ggplot(aes(x = race, y = percent_races)) +
  scale_y_continuous(labels = scales::label_percent(scale = 1)) +
  geom_col() +
  facet_wrap(vars(runner))
```



15.4 expand with nesting

So far, we have completed data frames for missing rows. Sometimes, however, we are interested in the unique combinations of values in a data frame. Suppose your running club has 540 members. You want to know in which competitions a runner has participated during her or his time in the club. This is basically the opposite of what we just did. Instead of finding all combinations of values we are looking for the unique combinations; in a given data frame!

To find these combinations we can combine `expand` with `nesting`:

```
running_races_club %>%
  expand(nesting(runner, race))
```

```
# A tibble: 6 x 2
  runner race
  <chr>  <chr>
1 Eva    half marathon
2 Eva    marathon
3 John   10km
4 Leonie 10km
5 Leonie half marathon
```

6 Leonie marathon

Once again, you can see that Eva has never run a 10K. John has never run a half marathon or marathon. But we have to infer that information from the data frame. The data shows what happened, not what didn't happen. To find out which runs the runners have never participated in, we can combine the code with `anti_join`:

```
full_combinations_runners <- expand(running_races_club,
  runner, race = c("10km", "half marathon","marathon"))

full_combinations_runners %>%
  anti_join(running_races_club, by = c("runner", "race"))

# A tibble: 3 x 2
  runner race
  <chr>  <chr>
1 Eva    10km
2 John   half marathon
3 John   marathon
```

The result of our analysis is now easier to process, as we no longer have to search for the known unknowns and get the desired results directly.

15.5 crossing

Let's talk about tennis. Suppose you want to create a data frame that shows all Grand Slams (Australian Open, French Open, Wimbledon, US Open) from 1905 to 2022 ([1905 was the first year all Grand Slams were held](#)). You don't have an existing data frame at hand, so you need to create one from scratch.

For these cases you need `crossing`. The difference from the other functions is that `crossing` does not need an existing data frame. We use vectors instead:

```
crossing(
  year = 1905:2022,
  major = c("Australian Open", "French Open",
           "Wimbledon", "US Open")
)
```

```
# A tibble: 472 x 2
  year major
  <int> <chr>
1 1905 Australien Open
2 1905 French Open
3 1905 US Open
4 1905 Wimbledon
5 1906 Australien Open
6 1906 French Open
7 1906 US Open
8 1906 Wimbledon
9 1907 Australien Open
10 1907 French Open
# ... with 462 more rows
```

This gives us a total of 472 Grand Slams.

Similarly, we could create a data frame representing the [World Marathon Majors](#), which started in 2006:

```
crossing(
  year = 2006:2022,
  races = c("Tokyo", "Boston", "Chicago",
            "London", "Berlin", "New York")
)
```



```
# A tibble: 102 x 2
  year races
  <int> <chr>
1 2006 Berlin
2 2006 Boston
3 2006 Chicago
4 2006 London
5 2006 New York
6 2006 Tokyo
7 2007 Berlin
8 2007 Boston
9 2007 Chicago
10 2007 London
# ... with 92 more rows
```

The data itself only gives us a complete set of combinations, by itself it is not very meaningful. `crossing` is usually a starting point for further analyses. Imagine if we had a data set with

all the world records set at these majors. We could join the world records to this data frame to determine the percentage of races in which a world record was set at the six majors.

i Summary

- `complete`, `expand` and `crossing` all create complete sets of combinations of values. `complete` and `expand` derive the complete set from values already present in a data frame or vectors, `crossing` from vectors only.
- `complete` is a wrapper around `expand`. It is basically `expand` in combination with `full_join`
- `complete` and `expand` can be used for grouped data frames to complete a set of combinations of values within groups only.
- `expand` and `crossing` create a new data frame, `complete` adds rows to an existing data frame
- `expand` in combination with `nesting` gives you the unique combinations of values in a data frame.

Part VII

**Improve converting data frames
between long and wide formats**

16 How to make a data frame longer

What will this tutorial cover?

In this tutorial you will learn how to make data frames longer. Most of the time you do this when your data frame is not tidy. We'll explain what an un-tidy data frame is and explore a few techniques to making data frames longer.

Who do I have to thank?

For this tutorial I have to thank Hadley Wickham for his article on Tidy Data ([Wickham, 2014](#)). Making data frames longer is closely related to tidy data. I used some ideas and examples from his article for this tutorial.

Many data sets are created or cleaned in spreadsheet programs. These programs are optimized for easy data entry and visual review. As a result, people tend to write un-tidy data.

Un-tidy data violates one of these three principles in one way or another (see [Wickham, 2014](#)):

- Each variable forms a column
- Each observation forms a row
- Each type of observation unit is a table

One could also say that in an un-tidy dataset, the physical layout is not linked to its semantics ([Neo, 2020](#)).

Suppose you receive the following data set from a colleague. Two groups of people “a” and “b” had to indicate for three weeks how often they ran in one week. The columns w-1 to w-3 represent the weeks 1 to 3:

```
(running_data <- tribble(  
  ~person, ~group, ~`w-1`, ~`w-2`, ~`w-3`,  
  "John",   "a",    4,      NA,      2,  
  "Marie",  "a",    2,      7,      3,  
  "Jane",   "b",    3,      8,      9,  
  "Peter",  "b",    1,      3,      3
```

```
)
```

```
# A tibble: 4 x 5
  person group `w-1` `w-2` `w-3`
  <chr>  <chr> <dbl> <dbl> <dbl>
1 John    a        4     NA     2
2 Marie   a        2      7     3
3 Jane    b        3      8     9
4 Peter   b        1      3     3
```

This data frame is un-tidy because not all columns represent variables. According to Wickham (2014), a variable contains “all values that measure the same underlying attribute (such as altitude, temperature, duration) across units” (p. 3). However, in our data frame, the columns `w-1`, `w-2`, and `w-3` represent values of a underlying variable `week` that is not represented as a column. A tidy representation of this data frame would look as follows:

```
running_data %>%
  pivot_longer(
    cols = `w-1`: `w-3`,
    names_to = "week",
    values_to = "value"
)

# A tibble: 12 x 4
  person group week  value
  <chr>  <chr> <chr> <dbl>
1 John    a     w-1     4
2 John    a     w-2     NA
3 John    a     w-3     2
4 Marie   a     w-1     2
5 Marie   a     w-2     7
6 Marie   a     w-3     3
7 Jane    b     w-1     3
8 Jane    b     w-2     8
9 Jane    b     w-3     9
10 Peter   b    w-1     1
11 Peter   b    w-2     3
12 Peter   b    w-3     3
```

You can already see that I used the `pivot_longer` function to create this data frame, which we will get to know in this tutorial. But before we dive deeper, let's get back to the main topic

of this tutorial: How to make a data frame longer. A data frame gets longer when we increase the number of its rows and decrease the number of its columns (see [Pivoting](#)). So when we tidy an un-tidy data set, we essentially make the data set longer.

In the following sections, we will go through some common use cases for cleaning up un-tidy data and use `pivot_longer` to make them longer. The use cases are mainly from [Wickham \(2014\)](#):

- Column headers are values of one variable, not variable names
- Multiple variables are stored in columns
- Multiple variables are stored in one column
- Variables are stored in both rows and columns.
- Variables are stored in both rows and columns

We will use some of the data sets from his article, but also others to increase the variability of the examples. Besides the use cases, we will also learn about some important parameters of the `pivot_longer` function.

16.1 Column headers are values of one variable, not variable names

We have already seen this problem in action in our running data set. The data set contains columns that represented values and not variable names:

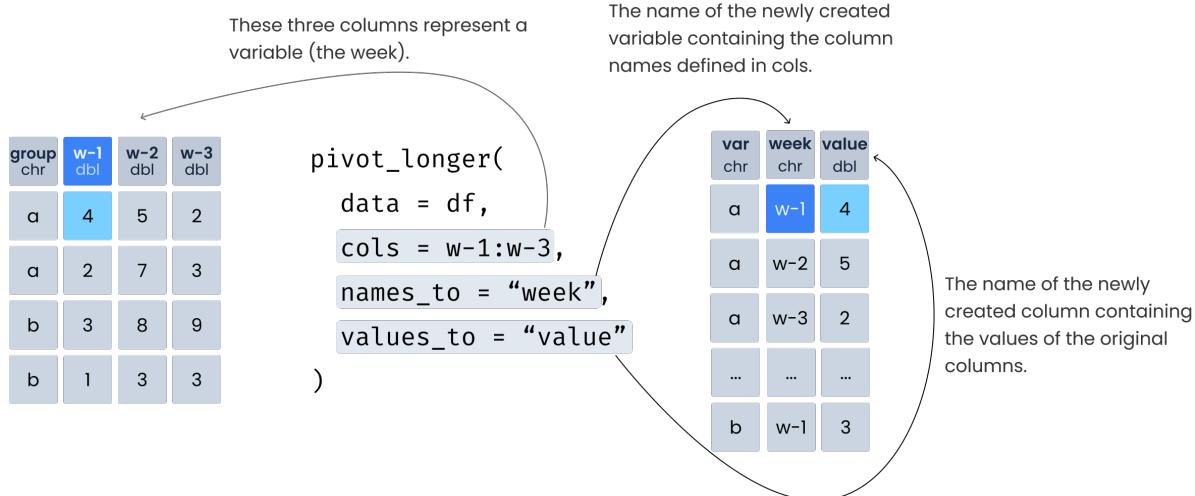
```
running_data %>% colnames  
  
[1] "person"  "group"   "w-1"      "w-2"      "w-3"
```

To make this data frame longer and tidy, we need to specify arguments for these four parameters in `pivot_longer`:

- `data`: The data frame to make longer
- `columns`: The columns that should be converted to a longer format
- `names_to`: The name of the new column that will contain the names of the columns
- `values_to`: The name of the new column that will contain the values inside the `cols` columns

Let's run the function and see what the results look like:

```
running_data %>%  
pivot_longer(  
  cols = `w-1`:`w-3`,  
  names_to = "week",
```



```

  values_to = "value"
)

```

```

# A tibble: 12 x 4
  person group week  value
  <chr>   <chr> <chr> <dbl>
1 John    a     w-1    4
2 John    a     w-2    NA
3 John    a     w-3    2
4 Marie   a     w-1    2
5 Marie   a     w-2    7
6 Marie   a     w-3    3
7 Jane    b     w-1    3
8 Jane    b     w-2    8
9 Jane    b     w-3    9
10 Peter   b    w-1    1
11 Peter   b    w-2    3
12 Peter   b    w-3    3

```

While our wider data frame had 20 values, our longer data frame has 48 values. We have more than doubled the number of values. The reason for this is that we duplicated the previous column names in our `week` column and also duplicated the values in the `person` and `group` columns.

We can further improve this code by removing the prefixes in the `week` column. `w-1` for example should be displayed as 1.

When you bring these columns to a longer format remove the “w-” prefix.

var chr	w-1 dbl	w-2 dbl	w-3 dbl
a	4	5	2
a	2	7	3
b	3	8	9
b	1	3	3

```
pivot_longer(  
  data = df,  
  cols = w-1:w-3,  
  names_to = "week",  
  names_prefix = "w-",  
  values_to = "value"  
)
```

var chr	week dbl	value dbl
a	1	4
a	2	5
a	3	2
...
b	1	3

```
running_data %>%  
  pivot_longer(  
    cols = `w-1`:`w-3`,  
    names_to = "week",  
    values_to = "value",  
    names_prefix = "w-"  
)
```

```
# A tibble: 12 x 4  
  person group week  value  
  <chr>   <chr> <chr> <dbl>  
1 John    a     1     4  
2 John    a     2     NA  
3 John    a     3     2  
4 Marie   a     1     2  
5 Marie   a     2     7  
6 Marie   a     3     3  
7 Jane    b     1     3  
8 Jane    b     2     8  
9 Jane    b     3     9  
10 Peter   b    1     1  
11 Peter   b    2     3  
12 Peter   b    3     3
```

You may also see that the variable `week` is a character and not a double. To convert the data type of the column `names_to` we can use the parameter `names_transform`:

```
running_data %>%
  pivot_longer(
    cols = `w-1`:`w-3`,
    names_to = "week",
    values_to = "value",
    names_prefix = "w-",
    names_transform = as.double
  )

# A tibble: 12 x 4
  person group  week value
  <chr>  <chr> <dbl> <dbl>
1 John    a      1     4
2 John    a      2     NA
3 John    a      3     2
4 Marie   a      1     2
5 Marie   a      2     7
6 Marie   a      3     3
7 Jane    b      1     3
8 Jane    b      2     8
9 Jane    b      3     9
10 Peter   b     1     1
11 Peter   b     2     3
12 Peter   b     3     3
```

Similarly, you could convert the data type of the `values_to` column with `values_transform` (a factor in this case):

```
running_data %>%
  pivot_longer(
    cols = `w-1`:`w-3`,
    names_to = "week",
    values_to = "value",
    names_prefix = "w-",
    values_transform = as.factor
  )

# A tibble: 12 x 4
```

```

  person group week  value
  <chr>  <chr> <chr> <fct>
1 John    a      1     4
2 John    a      2     <NA>
3 John    a      3     2
4 Marie   a      1     2
5 Marie   a      2     7
6 Marie   a      3     3
7 Jane    b      1     3
8 Jane    b      2     8
9 Jane    b      3     9
10 Peter  b      1     1
11 Peter  b      2     3
12 Peter  b      3     3

```

Here is another example where column headers represent values (from the `tidyverse` package):

```

relig_income

# A tibble: 18 x 11
# ... with 2 more variables: `>150k` <dbl>, `Don't know/refused` <dbl>, and
#   abbreviated variable names 1: `'$10-20k`, 2: `'$20-30k`, 3: `'$30-40k`,
  religion    `<$10k`  $10-2~1  $20-3~2  $30-4~3  $40-5~4  $50-7~5  $75-1~6  $100--7
  <chr>        <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>
1 Agnostic     27       34       60       81       76      137      122      109
2 Atheist      12       27       37       52       35      70       73       59
3 Buddhist     27       21       30       34       33      58       62       39
4 Catholic     418      617      732      670      638     1116     949      792
5 Don't know/r~ 15       14       15       11       10      35       21       17
6 Evangelical ~ 575      869      1064     982      881     1486     949      723
7 Hindu         1        9        7        9        11      34       47       48
8 Historically~ 228      244      236      238      197     223      131      81
9 Jehovah's Wi~ 20       27       24       24       21      30       15       11
10 Jewish        19      19       25       25       30      95       69       87
11 Mainline Prot 289      495      619      655      651     1107     939      753
12 Mormon        29       40       48       51       56      112      85       49
13 Muslim         6        7        9        10      9       23       16       8
14 Orthodox       13      17       23       32       32      47       38       42
15 Other Christ~  9        7        11       13       13      14       18       14
16 Other Faiths  20      33       40       46       49      63       46       40
17 Other World ~  5        2        3        4        2       7       3       4
18 Unaffiliated  217      299      374      365      341     528     407      321
# ... with 2 more variables: `>150k` <dbl>, `Don't know/refused` <dbl>, and
#   abbreviated variable names 1: `'$10-20k`, 2: `'$20-30k`, 3: `'$30-40k`,
```

```
# 4: `'$40-50k`, 5: `'$50-75k`, 6: `'$75-100k`, 7: `'$100-150k`
```

The columns <\$10k to Don't know/Refused are values of an underlying variable income. The values under these columns indicate the frequency with which individuals reported having a certain income. Let us tidy this data frame by making it longer:

```
relig_income %>%
  pivot_longer(
    cols = `<$10k`:`Don't know/refused`,
    names_to = "income",
    values_to = "freq"
  )

# A tibble: 180 x 3
  religion income           freq
  <chr>     <chr>        <dbl>
1 Agnostic <$10k            27
2 Agnostic $10-20k          34
3 Agnostic $20-30k          60
4 Agnostic $30-40k          81
5 Agnostic $40-50k          76
6 Agnostic $50-75k         137
7 Agnostic $75-100k         122
8 Agnostic $100-150k        109
9 Agnostic >150k            84
10 Agnostic Don't know/refused 96
# ... with 170 more rows
```

Again, our tidy data frame has more values ($180 * 3 = 540$) than our un-tidy data frame ($18 * 11 = 198$). Going back to our original statement that spreadsheet software is made for easy data entry, we can clearly see that it is easier to work with 198 values than 540.

Finally, another example. The data set **billboard** contains the top billboard rankings of the year 2000:

```
billboard

# A tibble: 317 x 79
  artist track date.ent~1   wk1   wk2   wk3   wk4   wk5   wk6   wk7   wk8   wk9
  <chr>  <chr> <date>      <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1 2 Pac  Baby~ 2000-02-26    87     82     72     77     87     94     99     NA     NA
```

```

2 2Ge+h~ The ~ 2000-09-02    91    87    92    NA    NA    NA    NA    NA    NA
3 3 Doo~ Kryp~ 2000-04-08    81    70    68    67    66    57    54    53    51
4 3 Doo~ Loser 2000-10-21    76    76    72    69    67    65    55    59    62
5 504 B~ Wobb~ 2000-04-15    57    34    25    17    17    31    36    49    53
6 98^0 Give~ 2000-08-19    51    39    34    26    26    19    2     2     3
7 A*Tee~ Danc~ 2000-07-08    97    97    96    95    100   NA    NA    NA    NA
8 Aaliy~ I Do~ 2000-01-29    84    62    51    41    38    35    35    38    38
9 Aaliy~ Try ~ 2000-03-18    59    53    38    28    21    18    16    14    12
10 Adams~ Open~ 2000-08-26   76    76    74    69    68    67    61    58    57
# ... with 307 more rows, 67 more variables: wk10 <dbl>, wk11 <dbl>,
#   wk12 <dbl>, wk13 <dbl>, wk14 <dbl>, wk15 <dbl>, wk16 <dbl>, wk17 <dbl>,
#   wk18 <dbl>, wk19 <dbl>, wk20 <dbl>, wk21 <dbl>, wk22 <dbl>, wk23 <dbl>,
#   wk24 <dbl>, wk25 <dbl>, wk26 <dbl>, wk27 <dbl>, wk28 <dbl>, wk29 <dbl>,
#   wk30 <dbl>, wk31 <dbl>, wk32 <dbl>, wk33 <dbl>, wk34 <dbl>, wk35 <dbl>,
#   wk36 <dbl>, wk37 <dbl>, wk38 <dbl>, wk39 <dbl>, wk40 <dbl>, wk41 <dbl>,
#   wk42 <dbl>, wk43 <dbl>, wk44 <dbl>, wk45 <dbl>, wk46 <dbl>, wk47 <dbl>, ...

```

Looking at the columns, we find a whopping 76 column names that are values of a variable week and not variables (wk1 to wk76):

```
billboard %>%
  colnames
```

```
[1] "artist"      "track"       "date.entered" "wk1"        "wk2"
[6] "wk3"         "wk4"         "wk5"          "wk6"        "wk7"
[11] "wk8"         "wk9"         "wk10"         "wk11"       "wk12"
[16] "wk13"        "wk14"        "wk15"         "wk16"       "wk17"
[21] "wk18"        "wk19"        "wk20"         "wk21"       "wk22"
[26] "wk23"        "wk24"        "wk25"         "wk26"       "wk27"
[31] "wk28"        "wk29"        "wk30"         "wk31"       "wk32"
[36] "wk33"        "wk34"        "wk35"         "wk36"       "wk37"
[41] "wk38"        "wk39"        "wk40"         "wk41"       "wk42"
[46] "wk43"        "wk44"        "wk45"         "wk46"       "wk47"
[51] "wk48"        "wk49"        "wk50"         "wk51"       "wk52"
[56] "wk53"        "wk54"        "wk55"         "wk56"       "wk57"
[61] "wk58"        "wk59"        "wk60"         "wk61"       "wk62"
[66] "wk63"        "wk64"        "wk65"         "wk66"       "wk67"
[71] "wk68"        "wk69"        "wk70"         "wk71"       "wk72"
[76] "wk73"        "wk74"        "wk75"         "wk76"
```

Even though this data frame is much wider than our previous examples, we can use the same function and parameters to make it longer:

```

billboard %>%
  pivot_longer(
    cols = contains("wk"),
    names_to = "week",
    values_to = "value",
    names_prefix = "^wk",
    names_transform = as.double
  )

# A tibble: 24,092 x 5
# ... with 24,082 more rows
  artist track           date.entered week value
  <chr>  <chr>          <date>       <dbl> <dbl>
1 2 Pac Baby Don't Cry (Keep... 2000-02-26     1     87
2 2 Pac Baby Don't Cry (Keep... 2000-02-26     2     82
3 2 Pac Baby Don't Cry (Keep... 2000-02-26     3     72
4 2 Pac Baby Don't Cry (Keep... 2000-02-26     4     77
5 2 Pac Baby Don't Cry (Keep... 2000-02-26     5     87
6 2 Pac Baby Don't Cry (Keep... 2000-02-26     6     94
7 2 Pac Baby Don't Cry (Keep... 2000-02-26     7     99
8 2 Pac Baby Don't Cry (Keep... 2000-02-26     8     NA
9 2 Pac Baby Don't Cry (Keep... 2000-02-26     9     NA
10 2 Pac Baby Don't Cry (Keep... 2000-02-26    10     NA
# ... with 24,082 more rows

```

16.2 Multiple variables are stored in columns

The previous use cases were quite clear, since we could assume that the columns used for `cols` represent all values of a single variable. However, this is not always the case. Let's take the data set `anscombe`:

```
anscombe
```

	x1	x2	x3	x4	y1	y2	y3	y4
1	10	10	10	8	8.04	9.14	7.46	6.58
2	8	8	8	8	6.95	8.14	6.77	5.76
3	13	13	13	8	7.58	8.74	12.74	7.71
4	9	9	9	8	8.81	8.77	7.11	8.84
5	11	11	11	8	8.33	9.26	7.81	8.47
6	14	14	14	8	9.96	8.10	8.84	7.04
7	6	6	6	8	7.24	6.13	6.08	5.25

```

8   4   4   4 19  4.26 3.10  5.39 12.50
9  12  12  12  8 10.84 9.13  8.15  5.56
10  7   7   7   8  4.82 7.26  6.42  7.91
11  5   5   5   8  5.68 4.74  5.73  6.89

```

Once we've made this data frame longer, we can see what it's all about. I can tell you this much: x represents values on the x-axis and y represents values on the y-axis. In other words, the column names represent two variables, x and y. Applying our `pivot_longer` logic to this example would not work because we would not be able to capture these two columns:

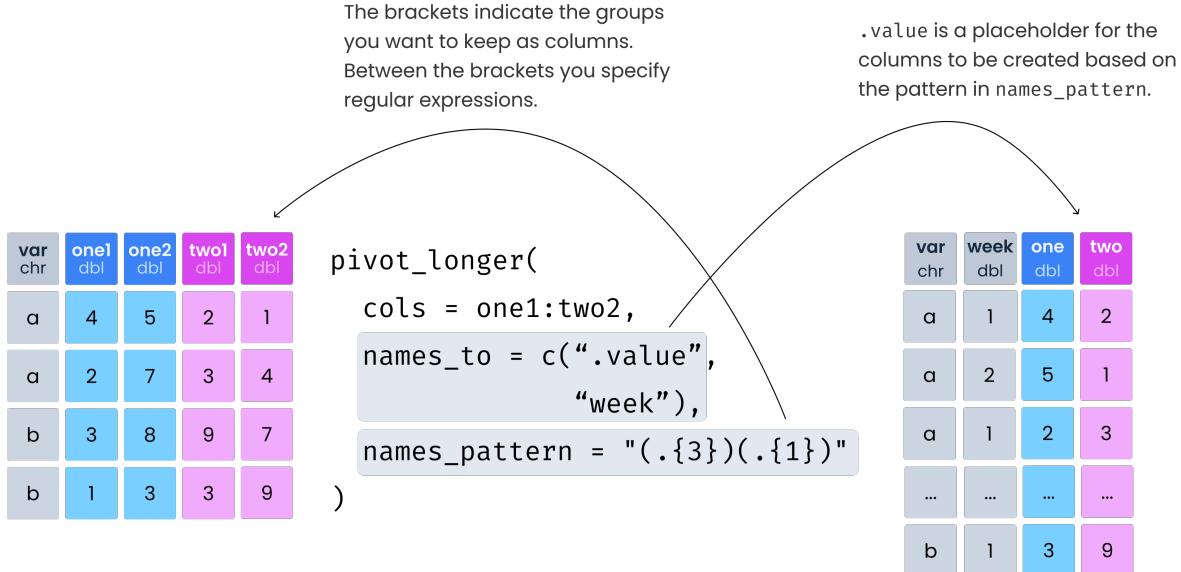
```

anscombe %>%
  pivot_longer(
    cols = x1:y4,
    names_to = "axis",
    values_to = "value"
  )

# A tibble: 88 x 2
  axis   value
  <chr> <dbl>
1 x1     10
2 x2     10
3 x3     10
4 x4      8
5 y1     8.04
6 y2     9.14
7 y3     7.46
8 y4     6.58
9 x1      8
10 x2     8
# ... with 78 more rows

```

To create two new variable columns in `names_to`, we need to use `.value` and the `names_pattern` parameter:



The most obscure element here is `.value`. To understand what `.value` does, let's first discuss `names_pattern`. `names_pattern` takes a regular expression. In the world of regular expressions, anything enclosed between parentheses is called a **group**. Groups allow us to capture parts of a string that belong together. In this example, the first group `(.{3})` contains the first three letters of a string. The second group `(.{1})` contains the fourth letter of this string. You can see that the length of the vector in `names_to` is equal to the number of groups defined in `names_pattern`. In other words, the elements in this vector represent the groups. If we look at the first group, we find two different elements: `one` and `two`. `.value` is a placeholder for these two elements. For each element, a new column is created with the text captured in the regular expression.

Now that we've seen how it works, let's tidy our data:

```

(anscombe_tidy <- anscombe %>%
  pivot_longer(
    cols = x1:y4,
    names_to = c(".value", "number"),
    names_pattern = "[xy](\\d+)",
  ))
  
```

```

# A tibble: 44 x 3
  number      x      y
  <chr>   <dbl> <dbl>
1 1          10  8.04
2 2          10  9.14
3 3          10  7.46
  
```

```

4 4      8  6.58
5 1      8  6.95
6 2      8  8.14
7 3      8  6.77
8 4      8  5.76
9 1      13 7.58
10 2     13 8.74
# ... with 34 more rows

```

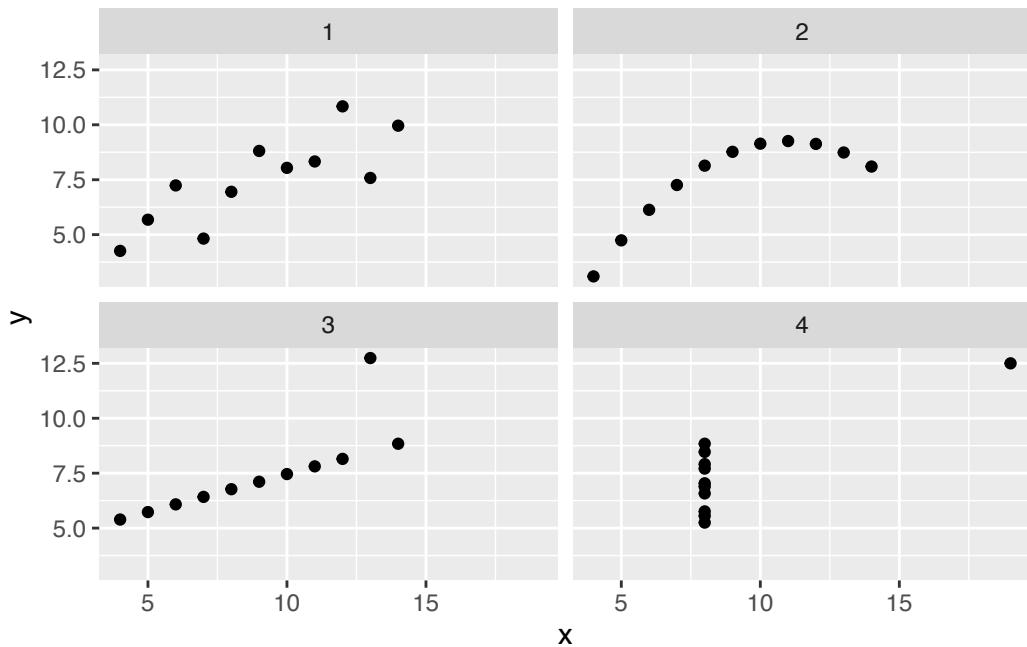
The regular expression needs some explanation. Again, the regex has two groups: "([xy])(\\d+)". The first group captures either the letter x or y: ([xy]). The second group captures one or more numbers: (\\d+).

Now that we have tidy data, we can visualize the idea behind the Anscombe dataset:

```

anscombe_tidy %>%
  ggplot(aes(x = x, y = y)) +
  geom_point() +
  facet_wrap(vars(number))

```



The data depicts four sets of data that have identical descriptive statistics (e.g., mean, standard deviation) but appear to be visually different from each other (see [Anscombe Quartet](#)).

16.3 Multiple variables are stored in one column

Let's look at another use case and example. The `who` dataset comes from the World Health Organization. It records confirmed tuberculosis cases broken down by country, year, and demographic group. The demographic groups are sex and age. Cases are broken down by four types: rel = relapse, sn = negative lung smear, sp = positive lung smear, ep = extrapulmonary.

```
who
```

```
# A tibble: 7,240 x 60
  country     iso2   iso3   year new_s~1 new_s~2 new_s~3 new_s~4 new_s~5 new_s~6
  <chr>      <chr>  <chr>  <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>
1 Afghanistan AF     AFG     1980      NA       NA       NA       NA       NA
2 Afghanistan AF     AFG     1981      NA       NA       NA       NA       NA
3 Afghanistan AF     AFG     1982      NA       NA       NA       NA       NA
4 Afghanistan AF     AFG     1983      NA       NA       NA       NA       NA
5 Afghanistan AF     AFG     1984      NA       NA       NA       NA       NA
6 Afghanistan AF     AFG     1985      NA       NA       NA       NA       NA
7 Afghanistan AF     AFG     1986      NA       NA       NA       NA       NA
8 Afghanistan AF     AFG     1987      NA       NA       NA       NA       NA
9 Afghanistan AF     AFG     1988      NA       NA       NA       NA       NA
10 Afghanistan AF    AFG     1989      NA       NA       NA       NA       NA
# ... with 7,230 more rows, 50 more variables: new_sp_m65 <dbl>,
#   new_sp_f014 <dbl>, new_sp_f1524 <dbl>, new_sp_f2534 <dbl>,
#   new_sp_f3544 <dbl>, new_sp_f4554 <dbl>, new_sp_f5564 <dbl>,
#   new_sp_f65 <dbl>, new_sn_m014 <dbl>, new_sn_m1524 <dbl>,
#   new_sn_m2534 <dbl>, new_sn_m3544 <dbl>, new_sn_m4554 <dbl>,
#   new_sn_m5564 <dbl>, new_sn_m65 <dbl>, new_sn_f014 <dbl>,
#   new_sn_f1524 <dbl>, new_sn_f2534 <dbl>, new_sn_f3544 <dbl>, ...
```

```
who %>% colnames
```

```
[1] "country"      "iso2"        "iso3"        "year"         "new_sp_m014"
[6] "new_sp_m1524" "new_sp_m2534" "new_sp_m3544" "new_sp_m4554" "new_sp_m5564"
[11] "new_sp_m65"   "new_sp_f014"   "new_sp_f1524" "new_sp_f2534" "new_sp_f3544"
[16] "new_sp_f4554" "new_sp_f5564" "new_sp_f65"   "new_sn_m014"  "new_sn_m1524"
[21] "new_sn_m2534" "new_sn_m3544" "new_sn_m4554" "new_sn_m5564" "new_sn_m65"
[26] "new_sn_f014"  "new_sn_f1524" "new_sn_f2534" "new_sn_f3544" "new_sn_f4554"
[31] "new_sn_f5564" "new_sn_f65"   "new_ep_m014"  "new_ep_m1524" "new_ep_m2534"
[36] "new_ep_m3544" "new_ep_m4554" "new_ep_m5564" "new_ep_m65"   "new_ep_f014"
```

```
[41] "new_ep_f1524" "new_ep_f2534" "new_ep_f3544" "new_ep_f4554" "new_ep_f5564"
[46] "new_ep_f65"    "newrel_m014"   "newrel_m1524"   "newrel_m2534"   "newrel_m3544"
[51] "newrel_m4554"  "newrel_m5564"  "newrel_m65"    "newrel_f014"   "newrel_f1524"
[56] "newrel_f2534"  "newrel_f3544"  "newrel_f4554"  "newrel_f5564"  "newrel_f65"
```

In this data frame, the underlying variables `type`, `cases`, `age` and `gender` are contained in the column headings. Let us take the column `new_sp_m1524`. `sp` stands for the type that has a positive lung smear. `m` stands for male and `1524` stands for the age group from 15 to 24 years.

Another problem with this data frame is that the column names are not well formatted. In some columns `new` is followed by an underscore `new_sp_m3544`, in others not, `newrel_m2534`.

Here you can see how this data frame can be tidied with `pivot_longer`. We will break it down further next.

```
(who_cleaned <- who %>%
  pivot_longer(
    cols = new_sp_m014:newrel_f65,
    names_pattern = "new_?([a-z]{2,3})_([a-z])(\\d+)",
    names_to = c("type", "sex", "age"),
    values_to = "cases"
  ) %>%
  mutate(
    age = case_when(
      str_length(age) == 2 ~ age,
      str_length(age) == 3 ~ str_replace(age, "(^.)", "\\\\1-"),
      str_length(age) == 4 ~ str_replace(age, "(^.{2})", "\\\\1-"),
      TRUE ~ age
    )
  )
)
```

```
# A tibble: 405,440 x 8
  country     iso2 iso3 year type sex   age cases
  <chr>       <chr> <chr> <dbl> <chr> <chr> <dbl>
1 Afghanistan AF   AFG   1980 sp    m    0-14   NA
2 Afghanistan AF   AFG   1980 sp    m    15-24   NA
3 Afghanistan AF   AFG   1980 sp    m    25-34   NA
4 Afghanistan AF   AFG   1980 sp    m    35-44   NA
5 Afghanistan AF   AFG   1980 sp    m    45-54   NA
6 Afghanistan AF   AFG   1980 sp    m    55-64   NA
7 Afghanistan AF   AFG   1980 sp    m    65      NA
```

```

8 Afghanistan AF      AFG      1980 sp      f      0-14      NA
9 Afghanistan AF      AFG      1980 sp      f      15-24      NA
10 Afghanistan AF     AFG      1980 sp      f      25-34      NA
# ... with 405,430 more rows

```

Let's start with `pivot_longer`. The main difference from our previous example is that the regular expression for `names_pattern` is more complex. The regular expression captures three groups. Each group is converted into a new column.

- The first group (`[a-z]{2,3}`) is converted into a column representing the type of case
- The second group (`[a-z]`) is converted to the gender column
- The third group (`\d+`) is translated into the column `age`

See also how we solved the problem with the underscore in the regular expression by making it optional `new_?`.

```
(who_tidy_first_step <- who %>%
  pivot_longer(
    cols = new_sp_m014:newrel_f65,
    names_pattern = "new_?( [a-z]{2,3})_( [a-z])(\\d+)",
    names_to = c("type", "sex", "age"),
    values_to = "cases"
  ))
```

```

# A tibble: 405,440 x 8
  country   iso2   iso3   year type   sex   age   cases
  <chr>     <chr>   <chr> <dbl> <chr> <chr> <dbl>
1 Afghanistan AF     AFG     1980 sp     m     014     NA
2 Afghanistan AF     AFG     1980 sp     m     1524    NA
3 Afghanistan AF     AFG     1980 sp     m     2534    NA
4 Afghanistan AF     AFG     1980 sp     m     3544    NA
5 Afghanistan AF     AFG     1980 sp     m     4554    NA
6 Afghanistan AF     AFG     1980 sp     m     5564    NA
7 Afghanistan AF     AFG     1980 sp     m     65      NA
8 Afghanistan AF     AFG     1980 sp     f     014      NA
9 Afghanistan AF     AFG     1980 sp     f     1524    NA
10 Afghanistan AF    AFG     1980 sp     f     2534    NA
# ... with 405,430 more rows

```

Next, we need to clean up the `age` column. It should be cleaned as follows:

- 014 -> 0-14

- 1524 -> 15-24
- 65 -> 65

We can do this conversion with `mutate` and `case_when`:

```
(who_tidy_second_step <- who_tidy_first_step %>%
  mutate(
    age = case_when(
      str_length(age) == 2 ~ age,
      str_length(age) == 3 ~ str_replace(age, "(^.)", "\\\1-"),
      str_length(age) == 4 ~ str_replace(age, "(.{2})", "\\\1-"),
      TRUE ~ age
    )
  ))
```

```
# A tibble: 405,440 x 8
  country     iso2   iso3   year type sex   age   cases
  <chr>      <chr>  <chr>  <dbl> <chr> <chr> <chr> <dbl>
1 Afghanistan AF     AFG     1980 sp     m    0-14    NA
2 Afghanistan AF     AFG     1980 sp     m    15-24    NA
3 Afghanistan AF     AFG     1980 sp     m    25-34    NA
4 Afghanistan AF     AFG     1980 sp     m    35-44    NA
5 Afghanistan AF     AFG     1980 sp     m    45-54    NA
6 Afghanistan AF     AFG     1980 sp     m    55-64    NA
7 Afghanistan AF     AFG     1980 sp     m    65      NA
8 Afghanistan AF     AFG     1980 sp     f    0-14    NA
9 Afghanistan AF     AFG     1980 sp     f    15-24    NA
10 Afghanistan AF    AFG     1980 sp    f    25-34   NA
# ... with 405,430 more rows
```

You can see that the data frame is quite large (405,440 rows and 8 columns). However, most of the values are `NA`. Fortunately, we can easily remove these `NAs` with the `values_drop_na` parameter by setting it to `TRUE`:

```
(who_cleaned_small <- who %>%
  pivot_longer(
    cols = new_sp_m014:newrel_f65,
    names_pattern = "new_?([a-z]{2,3})_( [a-z])(\\d+)",
    names_to = c("type", "sex", "age"),
    values_to = "cases",
    values_drop_na = TRUE
) %>%
```

```

mutate(
  age = case_when(
    str_length(age) == 2 ~ age,
    str_length(age) == 3 ~ str_replace(age, "(^.)", "\\\1-"),
    str_length(age) == 4 ~ str_replace(age, "(^.{2})", "\\\1-"),
    TRUE ~ age
  )
))

```

A tibble: 76,046 x 8

	country	iso2	iso3	year	type	sex	age	cases
	<chr>	<chr>	<chr>	<dbl>	<chr>	<chr>	<chr>	<dbl>
1	Afghanistan	AF	AFG	1997	sp	m	0-14	0
2	Afghanistan	AF	AFG	1997	sp	m	15-24	10
3	Afghanistan	AF	AFG	1997	sp	m	25-34	6
4	Afghanistan	AF	AFG	1997	sp	m	35-44	3
5	Afghanistan	AF	AFG	1997	sp	m	45-54	5
6	Afghanistan	AF	AFG	1997	sp	m	55-64	2
7	Afghanistan	AF	AFG	1997	sp	m	65	0
8	Afghanistan	AF	AFG	1997	sp	f	0-14	5
9	Afghanistan	AF	AFG	1997	sp	f	15-24	38
10	Afghanistan	AF	AFG	1997	sp	f	25-34	36
# ... with 76,036 more rows								

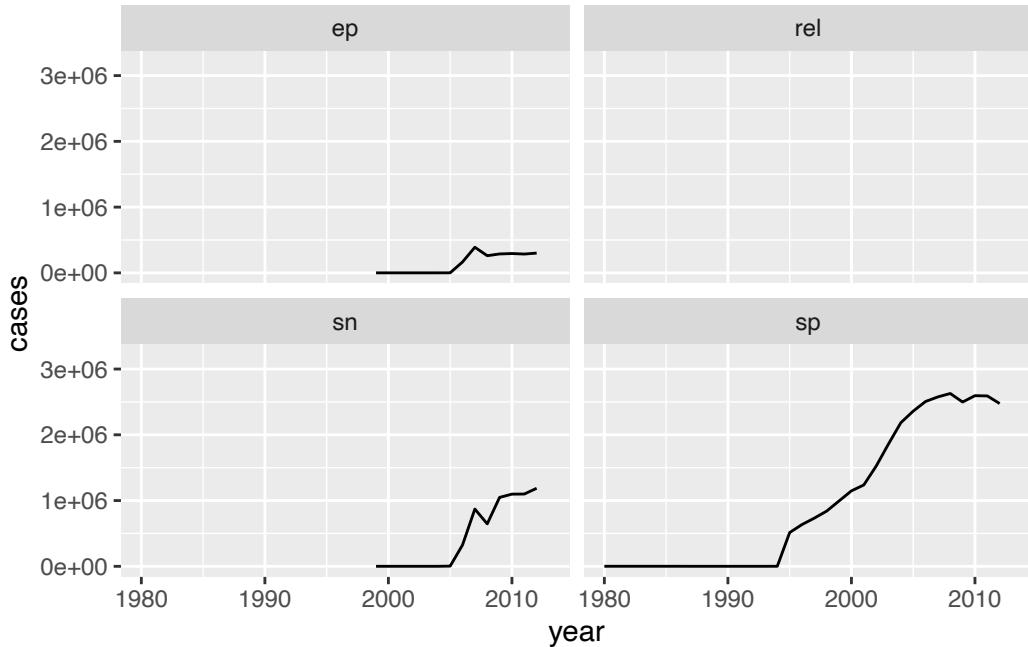
This data frame has only 76,046 lines. A reduction of 81%.

Now that we have this data frame, we can track cases over time:

```

who_cleaned_small %>%
  ggplot(aes(x = year, y = cases)) +
  stat_summary(
    fun = sum,
    geom = "line"
  ) +
  facet_wrap(vars(type))

```



16.4 Variables are stored in both rows and columns

In our last use case, an underlying variable is stored in both columns and rows. Consider this data frame:

```
weather_data <- tribble(
  ~id,      ~year,    ~month,   ~element,   ~d1,   ~d2,       ~d3,   ~d4,   ~d5,   ~d6,
  "MX17004", 2010,      1,   "tmax",    NA,    NA,      NA,    NA,    NA,    NA,
  "MX17004", 2010,      1,   "tmin",    NA,    NA,      NA,    NA,    NA,    NA,
  "MX17004", 2010,      2,   "tmax",    NA, 27.3, 24.1,    NA,    NA,    NA,
  "MX17004", 2010,      2,   "tmin",    NA, 14.4, 14.4,    NA,    NA,    NA,
  "MX17004", 2010,      3,   "tmax",    NA,    NA,      NA,    NA, 32.1,    NA,
  "MX17004", 2010,      3,   "tmin",    NA,    NA,      NA,    NA, 14.2,    NA,
  "MX17004", 2010,      4,   "tmax",    NA,    NA,      NA,    NA,    NA,    NA,
  "MX17004", 2010,      4,   "tmin",    NA,    NA,      NA,    NA,    NA,    NA,
  "MX17004", 2010,      5,   "tmax",    NA,    NA,      NA,    NA,    NA,    NA,
  "MX17004", 2010,      5,   "tmin",    NA,    NA,      NA,    NA,    NA,    NA,
) %>%
  mutate(across(d1:d6, as.numeric))
```

This data frame shows temperature data from a weather station in Mexico (see [Wickham, 2014, p. 10f](#)). Minimum and maximum temperatures were recorded daily. The columns d1 through

`d31` (some omitted here for readability) represent days, and the `year` and `month` columns represent the year and month, respectively.

What is striking about this data frame is that the underlying variable `date` is spread across rows and columns. Since the `date` variable should be in one column, this data is not tidy.

To solve this problem, we need to do two things. First, we need to make the data frame longer. Second, we need to create the `date` column. Here is how we could do this:

```
weather_data_cleaned <- weather_data %>%
  pivot_longer(
    cols = d1:d6,
    names_to = "day",
    names_prefix = "d",
    values_to = "value",
    values_drop_na = TRUE
  ) %>%
  unite(
    col = date,
    year, month, day,
    sep = "-"
  ) %>%
  mutate(
    date = as.Date(date, format = "%Y-%m-%d")
  ) %>% print()
```

```
# A tibble: 6 x 4
  id      date     element value
  <chr>   <date>   <chr>   <dbl>
1 MX17004 2010-02-02  ymax     27.3
2 MX17004 2010-02-03  ymax     24.1
3 MX17004 2010-02-02  ymin     14.4
4 MX17004 2010-02-03  ymin     14.4
5 MX17004 2010-03-05  ymax     32.1
6 MX17004 2010-03-05  ymin     14.2
```

The parameters in `pivot_longer` should already be familiar. We make the data frame longer with the columns `d1` to `d6`. We remove the prefix from these column values with `names_prefix` and we drop rows containing NAs.

The second step is to create the `date` column.

Summary

- Un-tidy data is usually the result of working with spreadsheet programs optimized for data entry
- To make messy data tidy, we usually have to lengthen it with `pivot_longer`.
- When we make data frames longer, we increase the number of rows and decrease the number of columns. We also increase the number of values in the data frame.
- There are four common use cases when making data frames longer: Column headers are values of one variable, not variable names, multiple variables are stored in columns, multiple variables are stored in one column, variables are stored in both rows and columns, variables are stored in both rows and column.
- If your column names contain more than one variable, you need to use the parameters `names_pattern` and `.value` in `names_to`.

17 How to make a data frame wider

What will this tutorial cover?

In this tutorial, you will learn how to make data frames wider. Because this technique reduces the number of values in a data frame, it is often useful when you need to make your data frames human readable. Other use cases are also discussed: For example, how to use the technique for feature engineering in machine learning and how to deal with certain challenges.

Who do I have to thank?

Before writing this tutorial, I asked the Twitter community about specific challenges they have with the `pivot_wider` function. I would like to thank everyone who provided answers. In particular, the following people who inspired me to write this tutorial:

[Cole](#), [Eric Stewart](#), [Guillaume Loignon](#), [Saurav Ghosh](#), [Marc-Aurèle Rivière](#), [neregauzak](#), [John Paul Helveston](#).

I also have to thank the developers of the tidyverse documentation. I took some of the ideas from their [official documentation of pivot_wider](#).

In the last tutorial, we said that many datasets are made longer in order to make them tidy. Tidy data is machine readable (thanks to [Cole](#) for this insight) in that it is optimized to be processed by data analytics tools. As a result, longer data frames have more values than shorter ones and they are easier to analyze with R.

Sometimes we want to do the opposite and make data frames wider. On Twitter, I asked the community to share their use cases. A good summary of most use cases is that wider data frames are more readable for humans. Either cognitively or for presentations. Here are the use cases we will cover in this tutorial based on the conversation on Twitter:

- How to use ‘pivot_wider’ (the simplest example)
- How to use `pivot_wider` to calculate ratios/percentages ([Julio](#) and [Eric Stewart](#))
- How to use `pivot_wider` to create tables of summary statistics (Proposal by [Guillaume Loignon](#)).
- How to make data frames wider for use in other software tools
- How to use `pivot_wider` to one-hot encode a factor ([Saurav Ghosh](#), [Marc-Aurèle Rivière](#))
- How to deal with multiple variable names stored in a column ([neregauzak](#))

- How to pivot_wider without an id column

In the following chapters, we will go through each of these use cases in detail. All of the cases have a few things in common. They extend the data frame by increasing the number of columns and decreasing the number of rows. Also, each use case can be implemented with the `pivot_wider` function.

17.1 How to use `pivot_wider` (the simplest example)

The `fish_encounters` data frame contains information about various stations that monitor and record the amount of fish passing through these stations downstream.

```
fish_encounters

# A tibble: 114 x 3
  fish   station  seen
  <fct> <fct>    <int>
1 4842  Release     1
2 4842  I80_1       1
3 4842  Lisbon      1
4 4842  Rstr        1
5 4842  Base_TD     1
6 4842  BCE         1
7 4842  BCW         1
8 4842  BCE2        1
9 4842  BCW2        1
10 4842 MAE         1
# ... with 104 more rows
```

The data frame has three columns. `fish` is an identifier for specific fish species. `station` is the name of the measuring station. `seen` indicates whether a fish was seen (1 if yes) or not seen (NA if not) at this station.

Suppose you would like to make this data frame wider because you would like to present the results in a human-readable table. To do this, you can use `pivot_wider` and provide arguments for its main parameters:

- `id_cols`: These columns are the identifiers for the observations. These column names remain unchanged in the data frame. Their values form the rows of the transformed data frame. By default, all columns except those specified in `names_from` and `values_from` become `id_cols`.

- **names_from**: These columns will be transformed into a wider format. Their values will be converted to columns. If you specify more than one column for **names_from**, the newly created column names will be a combination of the column values.
- **values_from**: The values of these columns will be used for the columns created with **names_from**.

Here is how the function looks like in action:

```
(fish_encounters_wide <- fish_encounters %>%
  pivot_wider(
    # this column will be kept in the new data frame
    id_cols = fish,
    # the values of this column will be the new column names
    names_from = station,
    # the values of these column will be used for the newly
    # created columns
    values_from = seen
  ))
```

A tibble: 19 x 12

	fish	Release	I80_1	Lisbon	Rstr	Base_TD	BCE	BCW	BCE2	BCW2	MAE	MAW
	<fct>	<int>	<int>	<int>	<int>	<int>	<int>	<int>	<int>	<int>	<int>	<int>
1	4842	1	1	1	1	1	1	1	1	1	1	1
2	4843	1	1	1	1	1	1	1	1	1	1	1
3	4844	1	1	1	1	1	1	1	1	1	1	1
4	4845	1	1	1	1	1	NA	NA	NA	NA	NA	NA
5	4847	1	1	1	NA	NA	NA	NA	NA	NA	NA	NA
6	4848	1	1	1	1	NA	NA	NA	NA	NA	NA	NA
7	4849	1	1	NA	NA	NA	NA	NA	NA	NA	NA	NA
8	4850	1	1	NA	1	1	1	1	NA	NA	NA	NA
9	4851	1	1	NA	NA	NA	NA	NA	NA	NA	NA	NA
10	4854	1	1	NA	NA	NA	NA	NA	NA	NA	NA	NA
11	4855	1	1	1	1	1	NA	NA	NA	NA	NA	NA
12	4857	1	1	1	1	1	1	1	1	1	NA	NA
13	4858	1	1	1	1	1	1	1	1	1	1	1
14	4859	1	1	1	1	1	NA	NA	NA	NA	NA	NA
15	4861	1	1	1	1	1	1	1	1	1	1	1
16	4862	1	1	1	1	1	1	1	1	1	NA	NA
17	4863	1	1	NA	NA	NA	NA	NA	NA	NA	NA	NA
18	4864	1	1	NA	NA	NA	NA	NA	NA	NA	NA	NA
19	4865	1	1	1	NA	NA	NA	NA	NA	NA	NA	NA

A small improvement to the data frame could be to prefix the new column names:

```

fish_encounters %>%
  pivot_wider(
    id_cols = fish,
    names_from = station,
    values_from = seen,
    # The prefix is added to each newly created column
    names_prefix = "station_"
  )

# A tibble: 19 x 12
# ... with 2 more variables: station_MAE <int>, station_MAW <int>, and
#   abbreviated variable names 1: station_Release, 2: station_I80_1,
#   3: station_Lisbon, 4: station_Rstr, 5: station_Base_TD, 6: station_BCE,
#   7: station_BCW, 8: station_BCE2, 9: station_BCW2

```

Another way to do this would be to use `names_glue`:

```

fish_encounters %>%
  pivot_wider(
    id_cols = fish,

```

```

    names_from = station,
    values_from = seen,
    # The prefix is added to each newly created column
    names_glue = "station_{station}"
  )

# A tibble: 19 x 12
  fish stati~1 stati~2 stati~3 stati~4 stati~5 stati~6 stati~7 stati~8 stati~9
  <fct> <int> <int> <int> <int> <int> <int> <int> <int> <int>
1 4842     1     1     1     1     1     1     1     1     1
2 4843     1     1     1     1     1     1     1     1     1
3 4844     1     1     1     1     1     1     1     1     1
4 4845     1     1     1     1     1     NA     NA     NA     NA
5 4847     1     1     1     NA     NA     NA     NA     NA     NA
6 4848     1     1     1     1     NA     NA     NA     NA     NA
7 4849     1     1     NA     NA     NA     NA     NA     NA     NA
8 4850     1     1     NA     1     1     1     1     NA     NA
9 4851     1     1     NA     NA     NA     NA     NA     NA     NA
10 4854    1     1     NA     NA     NA     NA     NA     NA     NA
11 4855    1     1     1     1     1     NA     NA     NA     NA
12 4857    1     1     1     1     1     1     1     1     1
13 4858    1     1     1     1     1     1     1     1     1
14 4859    1     1     1     1     1     NA     NA     NA     NA
15 4861    1     1     1     1     1     1     1     1     1
16 4862    1     1     1     1     1     1     1     1     1
17 4863    1     1     NA     NA     NA     NA     NA     NA     NA
18 4864    1     1     NA     NA     NA     NA     NA     NA     NA
19 4865    1     1     1     NA     NA     NA     NA     NA     NA
# ... with 2 more variables: station_MAE <int>, station_MAW <int>, and
#   abbreviated variable names 1: station_Release, 2: station_I80_1,
#   3: station_Lisbon, 4: station_Rstr, 5: station_Base_TD, 6: station_BCE,
#   7: station_BCW, 8: station_BCE2, 9: station_BCW2

```

`names_glue` takes a string with curly braces. Inside the curly braces you put the columns from `names_from`. The content inside the braces is replaced by the new column names.

17.2 How to use `pivot_wider` to calculate ratios/percentages

Suppose you have obtained the following data set on the rents and incomes of U.S. residents in U.S. states:

```
us_rent_income
```

```
# A tibble: 104 x 5
  GEOID NAME     variable estimate    moe
  <chr> <chr>     <chr>      <dbl> <dbl>
1 01   Alabama   income     24476    136
2 01   Alabama   rent       747      3
3 02   Alaska    income     32940    508
4 02   Alaska    rent       1200     13
5 04   Arizona   income     27517    148
6 04   Arizona   rent       972      4
7 05   Arkansas  income     23789    165
8 05   Arkansas  rent       709      5
9 06   California income    29454    109
10 06  California rent      1358     3
# ... with 94 more rows
```

The variable `variable` contains two values: `income` and `rent`. The actual estimated rent and the estimated income are stored in the variable `estimate`. The value for `income` indicates the mean annual income. The values for `rent` indicate the mean monthly income. `moe` indicates the margin of error for these values.

Clearly, this data frame is un-tidy as `income` and `rent` should be in two columns. Let's say you want to find out what percentage of their income people in different states have left for their rent. A suboptimal way would be a combination of `mutate` with `case_when` and `lead`:

```
us_rent_income %>%
  select(-moe) %>%
  mutate(
    # Standardize both values -> median yearly income/rent
    estimate = case_when(
      variable == "income" ~ estimate,
      variable == "rent"   ~ estimate * 12
    ),
    lead_estimate = lead(estimate),
    rent_percentage = (lead_estimate / estimate) * 100
  )

# A tibble: 104 x 6
  GEOID NAME     variable estimate lead_estimate rent_percentage
  <chr> <chr>     <chr>      <dbl>        <dbl>            <dbl>
1 01   Alabama   income     24476        24476          0
2 01   Alabama   rent       747         1494        19.8
3 02   Alaska    income     32940        32940          0
4 02   Alaska    rent       1200        14400        11.8
5 04   Arizona   income     27517        27517          0
6 04   Arizona   rent       972         1164        11.9
7 05   Arkansas  income     23789        23789          0
8 05   Arkansas  rent       709         850.9        11.9
9 06   California income    29454        29454          0
10 06  California rent      1358        16296        11.9
```

```

1 01    Alabama    income      24476      8964      36.6
2 01    Alabama    rent        8964      32940      367.
3 02    Alaska     income      32940      14400      43.7
4 02    Alaska     rent        14400      27517      191.
5 04    Arizona    income      27517      11664      42.4
6 04    Arizona    rent        11664      23789      204.
7 05    Arkansas   income      23789      8508       35.8
8 05    Arkansas   rent        8508       29454      346.
9 06    California income      29454      16296      55.3
10 06   California rent        16296      32401      199.
# ... with 94 more rows

```

The percentages we were looking for can be found in the `rent_percentage` column. Their values tell us two things: What percentage of rent people keep relative to their previous income, and what percentage of their income was relative to their rent. Again, we created an un-tidy set. Another problem with this approach is that we make an assumption with `lead`. We assume that the values `income` and `rent` alternate in the `variable` column. We could prove this, but it requires an unnecessary amount of work.

A better option is to make the data frame wider and calculate the percentage from the wider data set:

```

us_rent_income %>%
  pivot_wider(
    id_cols = c(GEOID, NAME),
    names_from = "variable",
    values_from = "estimate"
  ) %>%
  mutate(
    rent = rent * 12,
    percentage_of_rent = (rent / income) * 100
  )

```

# A tibble: 52 x 5		income	rent	percentage_of_rent
GEOID	NAME	<dbl>	<dbl>	<dbl>
1 01	Alabama	24476	8964	36.6
2 02	Alaska	32940	14400	43.7
3 04	Arizona	27517	11664	42.4
4 05	Arkansas	23789	8508	35.8
5 06	California	29454	16296	55.3
6 08	Colorado	32401	13500	41.7

```

7 09    Connecticut      35326 13476      38.1
8 10    Delaware         31560 12912      40.9
9 11    District of Columbia 43198 17088      39.6
10 12   Florida          25952 12924      49.8
# ... with 42 more rows

```

This approach has three advantages. First, we obtain a tidy data set. Second, we do not depend on the assumption that the `income` and `rent` values alternate. Third, it is less cognitively demanding. Since each column contains a variable, we don't need to worry about what those values represent. We simply divide one value by the other and multiply by 100.

17.3 How to use `pivot_wider` to create tables of summary statistics

Summary statistics are usually presented in papers, posters, and presentations. Since there is a limited amount of space available in these formats, they are presented in wider tables. As you may have heard already, wider data frames have fewer values than longer ones. In the next example, we will reduce the number of values from 105 to 40 by making the data frame wider. In other words, we're making the summary statistics of the data more human readable.

Diamonds can be described by different characteristics. The `cut` of a diamond can have different qualities (Fair, Good, Very Good Premium, Ideal). The color of a diamond can be categorized by the letters D (best) to J (worst). A diamond with color “D” is completely colorless. A diamond with the color “J” has some color and would therefore be of lower quality.

Suppose you want to plot the average prices of diamonds with different cuts and colors. You calculate them with `group_by` and `summarise`:

```

(means_diamonds <- diamonds %>%
  group_by(cut, color) %>%
  summarise(
    mean = mean(price)
  ))

# A tibble: 35 x 3
# Groups:   cut [5]
  cut   color  mean
  <ord> <ord> <dbl>
1 Fair   D     4291.
2 Fair   E     3682.

```

```

3 Fair F      3827.
4 Fair G      4239.
5 Fair H      5136.
6 Fair I      4685.
7 Fair J      4976.
8 Good D     3405.
9 Good E     3424.
10 Good F     3496.
# ... with 25 more rows

```

As you can see, the data frame has $35 * 3 = 105$ values. Not only can we reduce the data frame to 40 values, but we can also make it more readable so that readers can find each value quickly. Let's transform the data frame with `pivot_wider`:

```

means_diamonds %>%
  pivot_wider(
    id_cols = cut,
    names_from = color,
    values_from = mean,
    names_prefix = "mean_"
  )

# A tibble: 5 x 8
# Groups:   cut [5]
  cut      mean_D mean_E mean_F mean_G mean_H mean_I mean_J
  <ord>    <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
1 Fair      4291.  3682.  3827.  4239.  5136.  4685.  4976.
2 Good      3405.  3424.  3496.  4123.  4276.  5079.  4574.
3 Very Good 3470.  3215.  3779.  3873.  4535.  5256.  5104.
4 Premium   3631.  3539.  4325.  4501.  5217.  5946.  6295.
5 Ideal     2629.  2598.  3375.  3721.  3889.  4452.  4918.

```

With this data frame, we can make comparisons more easily: How much more expensive are “ideal” diamonds compared to “fair” diamonds? What influence does color have on the price of diamonds?

There is another way to calculate the mean values of these variables. I would like to point out that I do not recommend this approach, but the example is helpful to explain another parameter of `pivot_wider`. Suppose we do not calculate the means from the beginning and instead `select` the relevant columns and convert this data frame to a wider format:

```

(diamonds_means_as_lists <- diamonds %>%
  select(cut, color, price) %>%
  pivot_wider(
    id_cols = cut,
    names_from = color,
    values_from = price
  ))

```

A tibble: 5 x 8

	cut	E	I	J	H	F	G	D
	<ord>	<list>	<list>	<list>	<list>	<list>	<list>	<list>
1	Ideal	<int [3,903]>	<int [2,093]>	<int [896]>	<int>	<int>	<int>	<int>
2	Premium	<int [2,337]>	<int [1,428]>	<int [808]>	<int>	<int>	<int>	<int>
3	Good	<int [933]>	<int [522]>	<int [307]>	<int>	<int>	<int>	<int>
4	Very Good	<int [2,400]>	<int [1,204]>	<int [678]>	<int>	<int>	<int>	<int>
5	Fair	<int [224]>	<int [175]>	<int [119]>	<int>	<int>	<int>	<int>

What we get are columns that contain lists as values. Why? Because the rows were not uniquely identifiable. A row is uniquely identifiable if for each row there is only one value per column. In our case, all value combinations of `cut` and `color` appear more than once (e.g. “Fair” + “D” appears 163 times):

```

diamonds %>%
  select(cut, color, price) %>%
  count(cut, color)

```

A tibble: 35 x 3

	cut	color	n
	<ord>	<ord>	<int>
1	Fair	D	163
2	Fair	E	224
3	Fair	F	312
4	Fair	G	314
5	Fair	H	303
6	Fair	I	175
7	Fair	J	119
8	Good	D	662
9	Good	E	933
10	Good	F	909
	# ... with 25 more rows		

Enter `values_fn`. The parameter takes a function and applies this function to each list cell. For example, the first cell of column E contains integers.

```
diamonds_means_as_lists$E[[1]] %>% head  
[1] 326 554 2757 2761 2761 2762
```

From each of these lists we can calculate its mean:

```
diamonds %>%  
  select(cut, color, price) %>%  
  pivot_wider(  
    id_cols = cut,  
    names_from = color,  
    values_from = price,  
    values_fn = mean  
)  
  
# A tibble: 5 x 8  
  cut       E     I     J     H     F     G     D  
  <ord>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>  
1 Ideal    2598. 4452. 4918. 3889. 3375. 3721. 2629.  
2 Premium  3539. 5946. 6295. 5217. 4325. 4501. 3631.  
3 Good     3424. 5079. 4574. 4276. 3496. 4123. 3405.  
4 Very Good 3215. 5256. 5104. 4535. 3779. 3873. 3470.  
5 Fair     3682. 4685. 4976. 5136. 3827. 4239. 4291.
```

These are exactly the same values we calculated with `group_by`, `summarize` and `pivot_wider`. The only difference between the two data frames is that the order of the columns and the order of the values of `cut` are different.

Again, I do not recommend this approach. With `values_fn`, `pivot_longer` does two things at once. It makes the data frame wider and calculates summary statistics for the values. Separating the two steps makes the code easier to read and more comprehensible.

17.4 How to make data frames wider for use in other software tools

Suppose you conducted an experiment to test whether caffeine has an effect on the 100-meter time of runners. Two groups ran 100 meters twice with a break of 30 minutes. On the second run, the treatment group received a caffeine boost 10 minutes before the run, while the control

group didn't. The runners' cadence was also measured, i.e., the number of steps they take in one minute. Each runner is uniquely identifiable by an `id`:

```
(runners_data <- tibble(
  id = as.numeric(gl(6, 2)),
  group = c(rep("treatment", 6), rep("control", 6)),
  measurement = c(rep(c("pre", "post"), 6)),
  speed = rnorm(12, mean = 12, sd = 0.5),
  cadence = rnorm(12, mean = 160, 3)
))

# A tibble: 12 x 5
  id group measurement speed cadence
  <dbl> <chr>     <chr>     <dbl>    <dbl>
1 1 treatment pre      12.6    160.
2 1 treatment post    12.4    164.
3 2 treatment pre      11.9    158.
4 2 treatment post     10.2    163.
5 3 treatment pre      12.4    159.
6 3 treatment post     11.6    160.
7 4 control  pre      12.0    160.
8 4 control  post     11.3    157.
9 5 control  pre      11.7    164.
10 5 control  post     12.4    161.
11 6 control  pre      12.6    162.
12 6 control  post     12.4    162.
```

Let's say you need to analyze the data using GUI-based software. With such software and such experimental conditions, it is not uncommon that you need to make the data frame wider to compare the pre and post values of a variable.

If you pass only the `speed` column to `values_from`, you would lose all information about the cadence of the runners:

```
runners_data %>%
  pivot_wider(
  id_cols = id,
  names_from = measurement,
  values_from = speed
)

# A tibble: 6 x 3
```

```

      id  pre  post
<dbl> <dbl> <dbl>
1     1 12.6 12.4
2     2 11.9 10.2
3     3 12.4 11.6
4     4 12.0 11.3
5     5 11.7 12.4
6     6 12.6 12.4

```

To keep all the information from the data frame, we need to pass the columns `speed` and `cadence` to `values_from`:

```

runners_data %>%
  pivot_wider(
    id_cols = c(id, group),
    names_from = measurement,
    values_from = c(speed, cadence)
  )

# A tibble: 6 x 6
  id group      speed_pre speed_post cadence_pre cadence_post
<dbl> <chr>        <dbl>       <dbl>       <dbl>       <dbl>
1     1 treatment    12.6       12.4       160.       164.
2     2 treatment    11.9       10.2       158.       163.
3     3 treatment    12.4       11.6       159.       160.
4     4 control      12.0       11.3       160.       157.
5     5 control      11.7       12.4       164.       161.
6     6 control      12.6       12.4       162.       162.

```

The function created four new columns. Why four? First you count the number of unique values in the variable specified in `names_of`. Then you multiply this number by the number of columns specified in `values_from`: $2 * 2 = 4$.

We can aesthetically improve the names of these columns with `names_sep` and `names_glue`. Let's start with `names_sep`, since we haven't seen it yet. If you pass more than one column to `values_from`, the parameter specifies how you join the values. In our case with a dot `.`:

```

runners_data %>%
  pivot_wider(
    id_cols = c(id, group),
    names_from = measurement,
    values_from = c(speed, cadence),

```

```

    names_sep = "."
)

# A tibble: 6 x 6
  id group      speed.pre speed.post cadence.pre cadence.post
  <dbl> <chr>        <dbl>       <dbl>       <dbl>       <dbl>
1     1 treatment    12.6       12.4       160.       164.
2     2 treatment    11.9       10.2       158.       163.
3     3 treatment    12.4       11.6       159.       160.
4     4 control      12.0       11.3       160.       157.
5     5 control      11.7       12.4       164.       161.
6     6 control      12.6       12.4       162.       162.

```

You can also change the order of the values with `names_glue`:

```

runners_data %>%
  pivot_wider(
    id_cols = c(id, group),
    names_from = measurement,
    values_from = c(speed, cadence),
    names_glue = "{measurement}. {.value}"
)

# A tibble: 6 x 6
  id group      pre.speed post.speed pre.cadence post.cadence
  <dbl> <chr>        <dbl>       <dbl>       <dbl>       <dbl>
1     1 treatment    12.6       12.4       160.       164.
2     2 treatment    11.9       10.2       158.       163.
3     3 treatment    12.4       11.6       159.       160.
4     4 control      12.0       11.3       160.       157.
5     5 control      11.7       12.4       164.       161.
6     6 control      12.6       12.4       162.       162.

```

`.value` needs an explanation. We have seen this particular string in our `pivot_longer` tutorial. `.value` is a placeholder for the column names specified in `values_from`. Since we passed `speed` and `cadence` to the parameter, `.value` is replaced by these two values.

Not only GUI software like SPSS sometimes needs wider data to run statistical tests. Also R has some statistical functions that need wider data (see `t.test`). `pivot_wider` is therefore often needed in teams that use R in combination with GUI-based programs or for statistical analyses.

17.5 How to deal with multiple variable names stored in a column

Here is a distinctly un-tidy data frame (thanks to [neregauzak](#) for providing [this data set](#)).

```
(overnight_stays <- read_csv("data/etrm_03h_2.csv"))

# A tibble: 4 x 146
  variable      zona ~1 categ~2 da de~3 2011~~4 2011~~5 2011~~6 2011~~7 2011~~8
  <chr>        <chr>  <chr>    <chr>     <dbl>    <dbl>    <dbl>    <dbl>    <dbl>
1 Entradas     C.A. d~ Total   Total     1.20e5  1.40e5  1.78e5  2.18e5  2.08e5
2 Pernoctaciones C.A. d~ Total   Total     2.12e5  2.47e5  3.17e5  4.03e5  3.81e5
3 Grado de ocup~ C.A. d~ Total   Total     2.72e1  3.35e1  3.76e1  4.85e1  4.46e1
4 Grado de ocup~ C.A. d~ Total   Total     3.56e1  4.36e1  4.86e1  5.64e1  5.6 e1
# ... with 137 more variables: `2011-06` <dbl>, `2011-07` <dbl>,
#   `2011-08` <dbl>, `2011-09` <dbl>, `2011-10` <dbl>, `2011-11` <dbl>,
#   `2011-12` <dbl>, `2012-01` <dbl>, `2012-02` <dbl>, `2012-03` <dbl>,
#   `2012-04` <dbl>, `2012-05` <dbl>, `2012-06` <dbl>, `2012-07` <dbl>,
#   `2012-08` <dbl>, `2012-09` <dbl>, `2012-10` <dbl>, `2012-11` <dbl>,
#   `2012-12` <dbl>, `2013-01` <dbl>, `2013-02` <dbl>, `2013-03` <dbl>,
#   `2013-04` <dbl>, `2013-05` <dbl>, `2013-06` <dbl>, `2013-07` <dbl>, ...
```

The data frame contains data on entries, overnight stays and occupancy rates in hotel establishments in the Basque Country by geographic area, category (aggregated), day of the week and month.

You may have noticed the problem with the variable `variable`. The variable contains strings with actual variable names. These four values should be columns by themselves.

Also, the values of an underlying `date` column are spread across multiple columns:

```
overnight_stays %>%
  colnames %>%
  head(n = 20)

[1] "variable"      "zona geografica" "categoria"          "da de la semana"
[5] "2011-01"        "2011-02"           "2011-03"            "2011-04"
[9] "2011-05"        "2011-06"           "2011-07"            "2011-08"
[13] "2011-09"        "2011-10"           "2011-11"            "2011-12"
[17] "2012-01"        "2012-02"           "2012-03"            "2012-04"
```

To make this data frame tidy, we need to combine `pivot_longer` with `pivot_wider`. First we make the data frame longer by creating a `date` variable:

```
(overnight_stays_longer <- overnight_stays %>%
  pivot_longer(
    cols = matches("\\\\d{4}-\\\\d{2},"),
    names_to = "date",
    values_to = "value"
  )
)

# A tibble: 568 x 6
  variable `zona geografica` categoria `da de la semana` date     value
  <chr>      <chr>        <chr>        <chr>        <chr>     <dbl>
1 Entradas C.A. de Euskadi Total       Total       2011-01 120035
2 Entradas C.A. de Euskadi Total       Total       2011-02 140090
3 Entradas C.A. de Euskadi Total       Total       2011-03 177734
4 Entradas C.A. de Euskadi Total       Total       2011-04 218319
5 Entradas C.A. de Euskadi Total       Total       2011-05 207706
6 Entradas C.A. de Euskadi Total       Total       2011-06 225072
7 Entradas C.A. de Euskadi Total       Total       2011-07 273814
8 Entradas C.A. de Euskadi Total       Total       2011-08 277775
9 Entradas C.A. de Euskadi Total       Total       2011-09 239742
10 Entradas C.A. de Euskadi Total      Total       2011-10 217931
# ... with 558 more rows
```

Still, the four variable names stored in `variable` must be columns of their own. So let's make them wider with `pivot_wider`:

```
(overnight_stays_tidy <- overnight_stays_longer %>%
  pivot_wider(
    names_from = variable,
    values_from = value
  )
)

# A tibble: 142 x 8
  `zona geografica` categoria da de la ~1 date   Entra~2 Perno~3 Grado~4 Grado~5
  <chr>          <chr>    <chr>        <chr>     <dbl>    <dbl>    <dbl>    <dbl>
1 C.A. de Euskadi Total    Total       2011~ 120035  212303    27.2   35.6
2 C.A. de Euskadi Total    Total       2011~ 140090  246950    33.5   43.6
3 C.A. de Euskadi Total    Total       2011~ 177734  316541    37.6   48.6
4 C.A. de Euskadi Total    Total       2011~ 218319  403064    48.5   56.4
5 C.A. de Euskadi Total    Total       2011~ 207706  381320    44.6   56
6 C.A. de Euskadi Total    Total       2011~ 225072  416376    49.5   60.8
7 C.A. de Euskadi Total    Total       2011~ 273814  534680    60.5   68.3
```

```

8 C.A. de Euskadi    Total      Total      2011~  277775  607178   68.3   73.3
9 C.A. de Euskadi    Total      Total      2011~  239742  462017   54.7   65.6
10 C.A. de Euskadi   Total      Total      2011~  217931  410032   47.8   57.7
# ... with 132 more rows, and abbreviated variable names 1: `da de la semana` ,
#   2: Entradas, 3: Pernoctaciones, 4: `Grado de ocupacin por plazas` ,
#   5: `Grado de ocupacin por habitaciones`

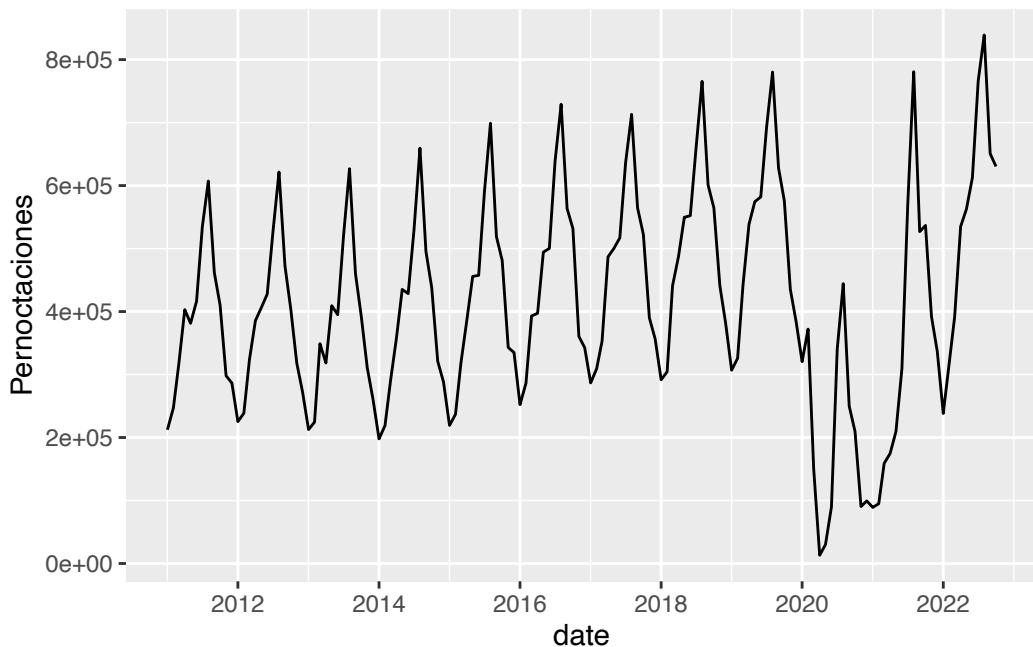
```

Now that the data frame is tidy, we can analyze the data properly. For example, we could plot the number of overnight stays over time:

```

overnight_stays_tidy %>%
  mutate(
    date = lubridate::ym(date)
  ) %>%
  ggplot(aes(x = date, y = Pernoctaciones)) +
  geom_line()

```



You can clearly see the onset of the Covid pandemic in 2020 and the seasonal trends within each year.

17.6 How to use pivot_wider to one-hot encode a factor

Marc-Aurèle Rivière provided me with [this use case](#). One-hot encoding is a machine learning technique in which categorial values are converted so they are readable by machine learning algorithms. With this technique, categorical values are converted into multiple numbers such that the length of the set is equal to the number of categorical values and the numbers contain only 0s and 1s. Each set contains the number 1 only once. This technique is important for some machine learning algorithms because they can only work with numeric columns. People with a statistical background know this technique by a slightly different name: Dummy variables.

Let's say you have a data set with the sugar values of four fruits: Pineapple, watermelon, bananas and grapes. With one-hot encoding, your fruits would be represented as follows:

	pineapple	watermelon	bananas	grapes
pineapple	1	0	0	0
watermelon	0	1	0	0
bananas	0	0	1	0
grapes	0	0	0	1

For example, pineapples would be represented by the set $\{1, 0, 0, 0\}$. Watermelons by the set $\{0, 1, 0, 0\}$.

Let's look at the technique using a simple example data frame:

```
(sugar_in_fruits_per_100g <- tribble(
  ~id, ~fruit, ~sugar_level,
  1, "pineapple", 10,
  2, "watermelon", 6,
  3, "banana", 12,
  4, "grape", 16
))

# A tibble: 4 x 3
  id fruit      sugar_level
  <dbl> <chr>        <dbl>
1     1 pineapple    10
2     2 watermelon    6
3     3 banana       12
4     4 grape        16
```

The first step is to put the data into a wider format. The strange thing about this step is that we use the same column for `names_from` and `names_value`:

```

sugar_in_fruits_per_100g %>%
  pivot_wider(
    names_from = fruit,
    values_from = fruit,
  )

# A tibble: 4 x 6
  id sugar_level pineapple watermelon banana grape
  <dbl>      <dbl> <chr>       <chr>      <chr>   <chr>
1     1          10 pineapple <NA>        <NA>    <NA>
2     2            6 <NA>       watermelon <NA>    <NA>
3     3           12 <NA>       <NA>       banana <NA>
4     4           16 <NA>       <NA>       <NA>    grape

```

Now that we have created a new column for each category or fruit, we need to convert the strings to the number 1. How can we do that? We know that the function `as.numeric` converts the value TRUE to the value 1:

```
as.numeric(TRUE)
```

```
[1] 1
```

So we have to convert the string to TRUE. We also know that any string is not a missing value:

```
is.na("pineapple")
```

```
[1] FALSE
```

If we toggle this boolean value, we get TRUE:

```
as.numeric(!is.na("pineapple"))
```

```
[1] 1
```

We can apply this transformation to any value of our newly created columns with `values_fn`:

```

sugar_in_fruits_per_100g %>%
  pivot_wider(
    names_from = fruit,
    values_from = fruit,
    values_fn = \x) as.numeric(!is.na(x))
)

# A tibble: 4 x 6
  id sugar_level pineapple watermelon banana grape
  <dbl>      <dbl>       <dbl>      <dbl>   <dbl>   <dbl>
1     1         10          1        NA      NA      NA
2     2          6         NA          1      NA      NA
3     3         12         NA          NA      1      NA
4     4         16         NA          NA      NA      1

```

Next we need to convert all NA to 0s. This can be done with `values_fill`. The parameter takes a value that will be used for each NA in the newly created columns. With `names_prefix` we can also add a prefix to the new columns:

```

sugar_in_fruits_per_100g %>%
  pivot_wider(
    names_from = fruit,
    values_from = fruit,
    values_fn = \x) as.numeric(!is.na(x)),
    values_fill = 0,
    names_prefix = "fruit_"
)

# A tibble: 4 x 6
  id sugar_level fruit_pineapple fruit_watermelon fruit_banana fruit_grape
  <dbl>      <dbl>           <dbl>           <dbl>       <dbl>       <dbl>
1     1         10              1              0          0          0
2     2          6              0              1          0          0
3     3         12              0              0          1          0
4     4         16              0              0          0          1

```

Voila! We have prepared the data frame for a machine learning algorithms using one-hot encoding.

17.7 How to use pivot_wider without an id column

This issue is discussed in the official [pivot_wider vignette](#). Suppose you are faced with the challenge of cleaning up this data frame. You should make the data frame wider in that it should include a `name`, `company` and `email` column.

```
(contacts <- tribble(
  ~field, ~value,
  "name", "Jiena McLellan",
  "company", "Toyota",
  "name", "John Smith",
  "company", "google",
  "email", "john@google.com",
  "name", "Huxley Ratcliffe"
))

# A tibble: 6 x 2
  field    value
  <chr>   <chr>
1 name     Jiena McLellan
2 company  Toyota
3 name     John Smith
4 company  google
5 email    john@google.com
6 name     Huxley Ratcliffe
```

It seems like this data frame is a simple example of `pivot_wider`. But without an id column, the three new columns contain lists of characters:

```
contacts %>%
  pivot_wider(
    id_cols = NULL,
    names_from = field,
    values_from = value
  )

# A tibble: 1 x 3
  name      company    email
  <list>    <list>    <list>
1 <chr [3]> <chr [2]> <chr [1]>
```

Why is that? The `id` columns in `id_cols` are used to identify each observation. For each observation the function creates a new row. Since we don't have id columns, only one row is created.

The solution is simple. Create an id column. Since not every person in the data frame has a name, company, and email, we cannot iterate from 1 to 3. Doing it manually is not an option either, as the data frame can easily grow by hundreds of lines. A nice trick is to use `cumsum`.

```
(contacts_with_id <- contacts %>%
  mutate(
    id = cumsum(field == "name")
  ))
```



```
# A tibble: 6 x 3
  field      value     id
  <chr>     <chr>   <int>
1 name      Jiena McLellan 1
2 company   Toyota        1
3 name      John Smith    2
4 company   google        2
5 email     john@google.com 2
6 name      Huxley Ratcliffe 3
```

In the context of `mutate` `cumsum` goes through a column and increments a number by one each time a new value is reached. This technique is robust when some characteristics of a person are not present (e.g. the email).

The rest is straightforward. We convert the data into a wider format:

```
contacts_with_id %>%
  pivot_wider(
    id_cols = id,
    names_from = field,
    values_from = value
  )
```



```
# A tibble: 3 x 4
  id name          company email
  <int> <chr>        <chr>   <chr>
1     1 Jiena McLellan Toyota  <NA>
2     2 John Smith    google   john@google.com
3     3 Huxley Ratcliffe <NA>    <NA>
```

Summary

- `pivot_wider` is often used to make data frames more readable (e.g. for posters or presentations).
- `pivot_wider` transforms data frames by reducing the number of rows and increasing the number of columns. It also reduces the total number of values within a data frame.
- `pivot_wider` can be used to implement one-hot encoding for a factor.

Part VIII

Improve your tidyverse fundamentals

18 How to make use of curly curly inside functions

What will this tutorial cover?

In this tutorial we will talk about the curly curly operator `{()}` of the `rlang` package. Curly curly allows you to refer to column names inside tidyverse functions. Basically, curly curly allows you to treat column names as if they were defined in the workspace. It's also easier to write code because you have to type less.

Who do I have to thank?

My thanks go to Lionel Henry, who wrote [an excellent blog post about the curly curly operator](#). I also thank [Bruno Rodrigues](#) who wrote another [nice blog post about curly curly](#).

18.1 What is curly curly and the `rlang` package?

As you dive deeper into Tidyverse, sooner or later you will hear about tidy evaluation, data masking and the curly curly operator `{()}`. All concepts have their basis in the `rlang` package of the Tidyverse package. `rlang` is a collection of frameworks and tools that help you program with R.

Most users of R that don't write packages will not need to invest much time in the `rlang` package. Sometimes, however, an error occurs that has its roots in the logic of the `rlang` package.

Here is the issue. Suppose you have written a function and want to pass a variable as an argument to the function body. The function should filter a data frame for a variable that is greater than a specified value. If you run this function, you will get the following error:

```
filter_larger_than <- function(data, variable, value) {  
  data %>%  
    filter(variable > value)
```

```

}

filter_larger_than(mpg, displ, 4)

Error in `filter()`:
! Problem while computing `...1 = variable > value`.
Caused by error in `mask$eval_all_filter()`:
! object 'displ' not found
Backtrace:
1. global filter_larger_than(mpg, displ, 2)
4. dplyr:::filter.data.frame(., variable > value)
5. dplyr:::filter_rows(.data, ..., caller_env = caller_env())
6. dplyr:::filter_eval(dots, mask = mask, error_call = error_call)
8. mask$eval_all_filter(dots, env_filter)

```

The issue is that you forgot to enclose the variable in the **curly curly operator**. But before we learn the mechanics of solving the issue, it's useful to learn a little bit of terminology. Let's start with **tidy evaluation**.

18.2 What is tidy evaluation?

You may have noticed that for almost all Tidyverse functions, you can simply refer to the column names without specifying the data frames they come from. In this example, we can directly refer to `displ` without specifying its data frame (`mpg$displ`):

```

mpg %>%
  filter(displ > 6)

# A tibble: 5 x 11
  manufacturer model      displ   year   cyl trans drv   cty   hwy fl class
  <chr>        <chr>     <dbl> <int> <int> <chr> <chr> <int> <int> <chr> <chr>
1 chevrolet    corvette    6.2   2008     8 manu~ r       16    26 p   2sea~
2 chevrolet    corvette    6.2   2008     8 auto~ r       15    25 p   2sea~
3 chevrolet    corvette    7.0   2008     8 manu~ r       15    24 p   2sea~
4 chevrolet    k1500 taho~  6.5   1999     8 auto~ 4       14    17 d   suv 
5 jeep         grand cher~  6.1   2008     8 auto~ 4       11    14 p   suv 

```

The technique that makes this possible is called tidy evaluation. In other words, "It makes it possible to manipulate data frame columns as if they were defined in the workspace" (<https://www.tidyverse.org/blog/2019/06/rlang-0-4-0/>).

The **workspace** is also called the global environment. You can think of an environment as a named list (e.g. `x`, `y`, `my_function`) or an unordered bag of names. To see the names in your environment, you can use `ls`:

```
ls(all.names = TRUE)  
  
[1] ".First"  ".main"   "filter"
```

The environment you normally work in is called **global environment** or **workspace**. We can check if the environment is the global environment by running this function:

```
environment()  
  
<environment: R_GlobalEnv>
```

All objects in the workspace are accessible through your R scripts and the console. If we were to create a new data frame named `my_tibble`, we would see that the columns of the data frame are not part of the environment, but the data frame is:

```
my_tibble <- tibble(  
  id = 1, 2, 3,  
  column_one = c(1, 3, 4)  
)  
  
ls()  
  
[1] "filter"     "my_tibble"
```

With tidy evaluation R acts as if the columns of data frames were part of the environment. In other words, they are accessible via the Tidyverse functions.

A closer look at the environment shows us that the columns of the data frame `my_tibble` do not belong to it. Nevertheless, through tidy evaluation we can access them in functions like `filter` and `select`.

There are two types of tidy evaluations: *data masking* and *tidy selection*. Data masking makes it possible to use columns as if they are variables in the environment. Tidy selection makes it possible to refer to columns inside tidy-select arguments (such as `starts_with`).

18.3 What is data masking?

Data masking allows you to use column names directly in `arrange`, `count`, `filter` and many other Tidyverse functions. The term itself tells you what it does: It masks the data. The [Webster Dictionary](#) defines masking as “to conceal (something) from view”. In our case, this “something” are the data frames.

The benefit of data masking is that you need to type less. Without data masking you would need to refer to column names with the name of the data frame and `mpg$displ`. As a result, code will become harder to read. Here is an example in Base R:

```
mpg[mpg$manufacturer == "audi" & mpg$displ > 3, ]
```

```
# A tibble: 5 x 11
  manufacturer model      displ  year   cyl trans drv   cty   hwy fl class
  <chr>        <chr>     <dbl> <int> <int> <chr> <chr> <int> <int> <chr> <chr>
1 audi         a4          3.1  2008     6 auto(~ f       18    27 p   comp~
2 audi         a4 quattro  3.1  2008     6 auto(~ 4      17    25 p   comp~
3 audi         a4 quattro  3.1  2008     6 manua~ 4      15    25 p   comp~
4 audi         a6 quattro  3.1  2008     6 auto(~ 4      17    25 p   mids~
5 audi         a6 quattro  4.2  2008     8 auto(~ 4      16    23 p   mids~
```

Within Tidyverse you can make the masked columns explicit by using the `.data` pronoun (you can find out more about `.data` in the [official documentation](#)).

```
mpg %>%
  filter(.data$displ > 6)
```

```
# A tibble: 5 x 11
  manufacturer model      displ  year   cyl trans drv   cty   hwy fl class
  <chr>        <chr>     <dbl> <int> <int> <chr> <chr> <int> <int> <chr> <chr>
1 chevrolet   corvette   6.2   2008     8 manu~ r       16    26 p   2sea~
2 chevrolet   corvette   6.2   2008     8 auto~ r      15    25 p   2sea~
3 chevrolet   corvette   7     2008     8 manu~ r      15    24 p   2sea~
4 chevrolet   k1500 taho~  6.5   1999     8 auto~ 4      14    17 d    suv
5 jeep        grand cher~  6.1   2008     8 auto~ 4      11    14 p   suv
```

18.4 What is curly curly `{}`?

Curly curly is a new operator that was introduced to [rlang 0.4.0](#). In short, curly curly makes data masking work inside functions.

You may have tried to create a function that uses Tidyverse functions in the function body. Suppose you want to write a function that filters a data frame based on a minimum value of a particular column. As we have seen in the beginning of this tutorial, this approach doesn't work:

```
filter_larger_than <- function(data, variable, value) {  
  data %>%  
    filter(variable > value)  
}  
  
filter_larger_than(mpg, displ, 4)  
  
Error in `filter()`:  
! Problem while computing `..1 = variable > value`.  
Caused by error in `mask$eval_all_filter()`:  
! object 'displ' not found  
Backtrace:  
1. global filter_larger_than(mpg, displ, 2)  
4. dplyr:::filter.data.frame(., variable > value)  
5. dplyr:::filter_rows(.data, ..., caller_env = caller_env())  
6. dplyr:::filter_eval(dots, mask = mask, error_call = error_call)  
8. mask$eval_all_filter(dots, env_filter)
```

Instead, you must enclose the column names with the curly curly operator:

```
filter_larger_than <- function(data, variable, value) {  
  data %>%  
    filter({{variable}} > value)  
}  
  
filter_larger_than(mpg, displ, 4)  
  
# A tibble: 71 x 11  
  manufacturer model      displ  year   cyl trans drv   cty   hwy fl class  
  <chr>        <chr>     <dbl> <int> <int> <chr> <chr> <int> <int> <chr> <chr>  
1 audi         a6 quattro  4.2   2008     8 auto~ 4       16    23 p   mids~  
2 chevrolet    c1500 sub~  5.3   2008     8 auto~ r       14    20 r   suv  
3 chevrolet    c1500 sub~  5.3   2008     8 auto~ r       11    15 e   suv  
4 chevrolet    c1500 sub~  5.3   2008     8 auto~ r       14    20 r   suv  
5 chevrolet    c1500 sub~  5.7   1999     8 auto~ r       13    17 r   suv  
6 chevrolet    c1500 sub~  6     2008     8 auto~ r       12    17 r   suv
```

```

7 chevrolet corvette 5.7 1999 8 manu~ r 16 26 p 2sea~
8 chevrolet corvette 5.7 1999 8 auto~ r 15 23 p 2sea~
9 chevrolet corvette 6.2 2008 8 manu~ r 16 26 p 2sea~
10 chevrolet corvette 6.2 2008 8 auto~ r 15 25 p 2sea~
# ... with 61 more rows

```

This is the trick. The mechanics are very simple. Let's see it in action with another example.

Tidy evaluation or data masking also works for data visualizations written in ggplot2. If you want to create a function that returns a ggplot object, you can refer to a column specified in the arguments by enclosing it with the curly curly operator.

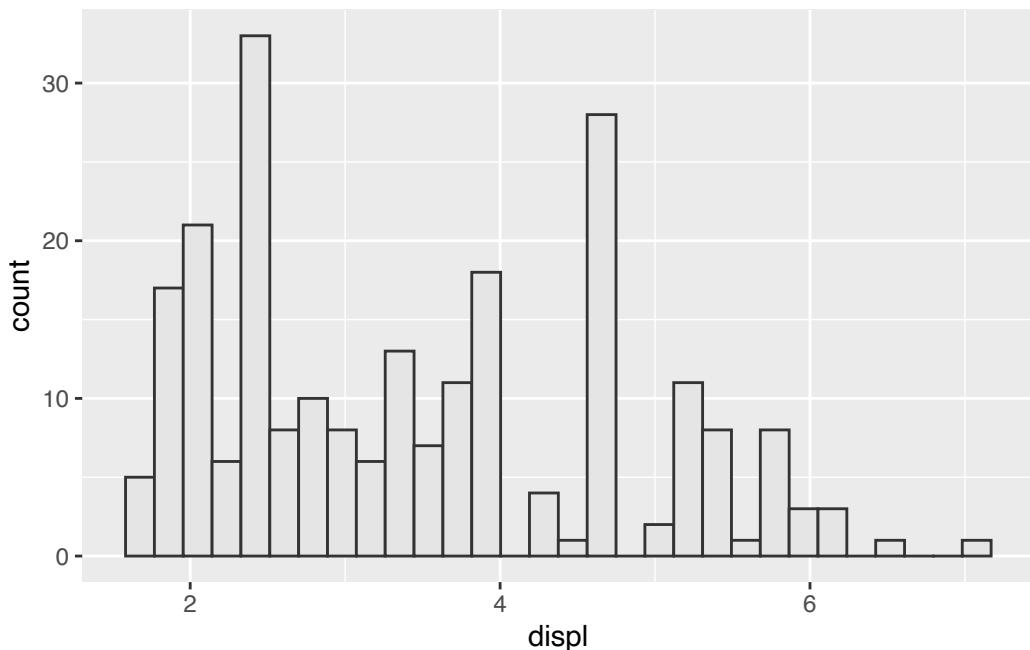
This function creates, for example, a histogram for an arbitrary data frame and a column of the data frame:

```

custom_histogram <- function(data, variable) {
  data %>%
    ggplot(aes(x = {{variable}})) +
    geom_histogram(fill = "grey90",
                  color = "grey20")
}

custom_histogram(mpg, displ)

```



18.5 How to pass multiple arguments to a function with the dot-dot-dot argument

A reasonable question to ask at this point would be whether multiple arguments passed to a function via the dot-dot-dot syntax must also be enclosed by a curly curly operator. The answer is no. Let's explore a few examples.

For those unfamiliar with dot-dot-dot arguments, let's go through a simple example. The `custom_select` function takes two arguments: a data frame `data` and an arbitrary number of unknown arguments In this case is a placeholder for a list of column names:

```
custom_select <- function(data, ...) {  
  data %>%  
  select(...)  
}  
  
custom_select(mpg, displ, manufacturer)  
  
# A tibble: 234 x 2  
  displ manufacturer  
  <dbl> <chr>  
1   1.8 audi  
2   1.8 audi  
3   2   audi  
4   2   audi  
5   2.8 audi  
6   2.8 audi  
7   3.1 audi  
8   1.8 audi  
9   1.8 audi  
10  2   audi  
# ... with 224 more rows
```

It may happen that you have to enclose a column with the curly curly operator and use the ... arguments, like in this example. The function `get_summary_statistics` groups the data frame by a specific column and then calculates the mean and standard deviation for any number of columns in that data frame. See how we do not use a curly curly operator for the ... arguments.

```
get_summary_statistics <- function(data, column, ...) {  
  data %>%
```

```

group_by({{column}}) %>%
summarise(
  across(
    .cols = c(...),
    .fns  = list(mean = ~ mean(., na.rm = TRUE),
                 sd   = ~ sd(., na.rm = TRUE)))
  )
)
}

get_summary_statistics(mpg, manufacturer, displ, cyl)

```

```

# A tibble: 15 x 5
  manufacturer displ_mean displ_sd cyl_mean cyl_sd
  <chr>          <dbl>     <dbl>     <dbl>     <dbl>
1 audi            2.54      0.673     5.22     1.22
2 chevrolet       5.06      1.37      7.26     1.37
3 dodge           4.38      0.868     7.08     1.12
4 ford            4.54      0.541      7.2      1
5 honda           1.71      0.145      4        0
6 hyundai         2.43      0.365     4.86     1.03
7 jeep             4.58      1.02      7.25     1.04
8 land rover      4.3       0.258      8        0
9 lincoln          5.4       0          8        0
10 mercury         4.4       0.490      7        1.15
11 nissan          3.27      0.864     5.54     1.20
12 pontiac         3.96      0.808     6.4      0.894
13 subaru          2.46      0.109      4        0
14 toyota           2.95      0.931     5.12     1.32
15 volkswagen      2.26      0.443     4.59     0.844

```

Here is another example using `slice_max`. Again, curly curly is only used for the specific column `column` that is masked:

```

slice_max_by <- function(data, ..., column, n) {
  data %>%
    group_by(...) %>%
    slice_max({{column}}, n = n) %>%
    ungroup()
}

```

```

slice_max_by(diamonds, cut, color, column = price, n = 1)

# A tibble: 37 x 10
  carat cut   color clarity depth table price     x     y     z
  <dbl> <ord> <ord> <ord>    <dbl> <dbl> <int> <dbl> <dbl> <dbl>
1  2.02 Fair    D     SI1      65     55 16386  7.94  7.84  5.13
2  1.5   Fair    E     VS1     65.4    57 15584  7.14  7.07  4.65
3  1.93 Fair    F     VS1     58.9    62 17995  8.17  7.97  4.75
4  2.01 Fair    G     SI1     70.6    64 18574  7.43  6.64  4.69
5  2.02 Fair    H     VS2     64.5    57 18565  8     7.95  5.14
6  3.01 Fair    I     SI2     65.8    56 18242  8.99  8.94  5.9
7  3.01 Fair    I     SI2     65.8    56 18242  8.99  8.94  5.9
8  4.5   Fair    J     I1      65.8    58 18531  10.2   10.2  6.72
9  2.04 Good   D     SI1     61.9    60 18468  8.15  8.11  5.03
10 2.02 Good   E     SI2     58.8    61 18236  8.21  8.25  4.84
# ... with 27 more rows

```

In the previous examples ... was a placeholder for column names. Similarly, you can specify whole arguments for:

```

grouped_filter <- function(data, grouping_column, ...) {
  data %>%
    group_by({{grouping_column}}) %>%
    filter(...) %>%
    ungroup()
}

grouped_filter(mpg, manufacturer, displ > 5, year == 2008)

# A tibble: 20 x 11
  manufacturer model      displ  year   cyl trans drv   cty   hwy fl class
  <chr>        <chr>    <dbl> <int> <int> <chr> <chr> <int> <int> <chr> <chr>
1 chevrolet    c1500 sub~    5.3  2008     8 auto~ r       14    20 r   suv 
2 chevrolet    c1500 sub~    5.3  2008     8 auto~ r       11    15 e   suv 
3 chevrolet    c1500 sub~    5.3  2008     8 auto~ r       14    20 r   suv 
4 chevrolet    c1500 sub~    6    2008     8 auto~ r       12    17 r   suv 
5 chevrolet    corvette    6.2  2008     8 manu~ r       16    26 p   2sea~
6 chevrolet    corvette    6.2  2008     8 auto~ r       15    25 p   2sea~
7 chevrolet    corvette    7   2008     8 manu~ r       15    24 p   2sea~
8 chevrolet    k1500 tah~    5.3  2008     8 auto~ 4       14    19 r   suv 

```

9	chevrolet	k1500 tah~	5.3	2008	8	auto~	4	11	14	e	suv
10	dodge	durango 4~	5.7	2008	8	auto~	4	13	18	r	suv
11	dodge	ram 1500 ~	5.7	2008	8	auto~	4	13	17	r	pick~
12	ford	expeditio~	5.4	2008	8	auto~	r	12	18	r	suv
13	ford	f150 pick~	5.4	2008	8	auto~	4	13	17	r	pick~
14	ford	mustang	5.4	2008	8	manu~	r	14	20	p	subc~
15	jeep	grand che~	5.7	2008	8	auto~	4	13	18	r	suv
16	jeep	grand che~	6.1	2008	8	auto~	4	11	14	p	suv
17	lincoln	navigator~	5.4	2008	8	auto~	r	12	18	r	suv
18	nissan	pathfinde~	5.6	2008	8	auto~	4	12	18	p	suv
19	pontiac	grand prix	5.3	2008	8	auto~	f	16	25	p	mids~
20	toyota	land crui~	5.7	2008	8	auto~	4	13	18	r	suv

This concludes our tutorial on the curly curly operator. The concept is not difficult to grasp, but essential if you want to write custom functions and pass columns to the function body.

Summary

- Tidyverse uses tidy evaluation and data masking so that you can use data frame columns as if they are defined in the workspace.
- When passing columns to functions and using the columns inside Tidyverse functions, enclose them with the curly curly operator `{()}`.

Part IX

Improve your purrr skills

19 How to use the `map` function family effectively

What will this tutorial cover?

This tutorial marks the beginning of a series on the `purrr` package, where we will delve deeper into the `map` family of functions. These functions provide an alternative to lists and allow you to iterate over atomic vectors, lists, and data frames. By the end of this tutorial, you will have experience with the most common use cases of these functions and will be equipped to start using them.

Who do I have to thank?

I started this chapter by putting out a [tweet asking the community for any issues they've had while working with `map` and its functions](#). Their feedback was crucial in writing this tutorial. A big shoutout to everyone who helped shape this chapter with their insights: [Isabella R. Ghement](#), [Richard Glolz](#), [Antoine Bichat](#), [Brenton Wiernik](#), [Matthew Kay](#), [Statistik Dresden](#), [Alexander Wuttke](#), [R & Vegan](#), [Juan LB](#), [bing_bong_telecom](#), [Kevin Korenblat](#), [Daniel Thiele](#), [Alex Pax](#), [Karsten Sieber](#), [Marius Grabow](#), [John T. Stone III](#), [Paul Bochtler](#)

19.1 Preparation

Before beginning this tutorial, make sure you have `purrr` 1.0.0 and R 4.1.0 installed. `purrr` 1.0.0 was released in [Dezember 2022](#) and introduced several breaking changes to the package, while R 4.1.0, released in [May, 2021](#), introduced the native pipe operator `|>`. This tutorial and the following will use the native pipe instead of the [magrittr](#) pipe.

19.2 A list, vector and data frame primer

Mastering `purrr` and the `map` family of functions can be challenging without a solid understanding of lists, vectors, and data frames. The `map` functions mostly take these data structures

as input. If you're not familiar with them, it's gonna be tough to follow along. I'll be talking about them a lot in this tutorial, so it's a good idea to brush up on them before diving in. Also, it can be a lot to take in all at once - data structures and the `map` functions. So, let's spend some time reviewing these data structures first.

19.2.1 An overview of data structures in R

R has five basic data structures: Atomic vectors, matrices, arrays, lists, and data frames (see <http://adv-r.had.co.nz/Data-structures.html>). They're different from each other in two ways: whether they hold the same data type or not, and whether they're one or two-dimensional. So, here's a quick rundown of these data structures:

	Homogeneous	Heterogeneous
1 dimension	Atomic vector	List
2 dimensions	Matrix	Data frame
n dimensions	Array	

Figure 19.1: R data structures

I won't be going too in-depth on matrices and arrays in this tutorial since they're not as important for purrr as the other data structures.

19.2.2 Atomic vectors

Atomic vectors are just one type of vector, with lists being the other. We'll talk more about lists in a bit. Both atomic vectors and lists have one thing in common: they're one-dimensional. Think of it like this: if we compare dimensions to directions, an element in a one-dimensional data structure can only move forward and backward (or up and down). But an element in a two-dimensional data structure can move in any direction. Here is a visual overview of the four types of atomic vectors (based on the book [R for Data Science](#)):

We have four types of atomic vectors: logical (`TRUE` or `FALSE`), character (text), integer (whole numbers like 1), and double (numbers with decimals like 1.31). Integers and doubles are also called numeric atomic vectors. You've all created them before. Let me show you how to create one of each type:

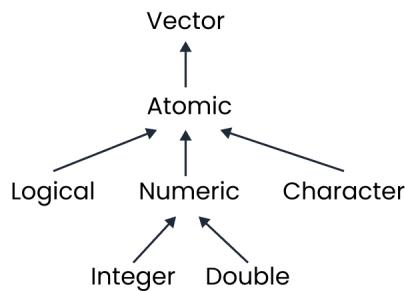


Figure 19.2: Atomic vectors

```
c(TRUE, FALSE) # logical
c("a", "bunch", "of strings") # character
c(4, 5, 9) # integer
c(5.1, 8.2) # double
```

It becomes interesting when we attempt to create non-homogeneous (or heterogeneous) vectors. By default, R will try to convert all elements of the vector to the same data type. Here is an example where we mix a double and a character:

```
c(2.1, "8")
```

```
[1] "2.1" "8"
```

The output is a character vector. In R or programming terminology, we would say the double was coerced to a character (for more information, refer to [this explanation](#) from R in a nutshell). To verify this is a character vector, we pipe it into `typeof`:

```
c(2.1, "8") |> typeof()
```

```
[1] "character"
```

Coercion occurs when we combine vector elements of different types. In this example, we have combined a logical, a double, and a character:

```
c(TRUE, 8.1, "house")
```

```
[1] "TRUE"  "8.1"   "house"
```

Again, we get a character vector. What we see here are coercion rules. These rules determine in what order data types should be coerced. According to Joseph Adler in his book [R in a nutshell](#) the coercion rules in R go as follows:

logical < integer < numeric < complex < character < list

When combining multiple data types in a vector, the data type on the left of the coercion rules will be converted to the data type on the right. For example, in the previous examples, the double and the logical was coerced to a character. This is why, when we combine a logical and a double, the logical will be converted to a double.

```
c(TRUE, 5.1, FALSE)
```

```
[1] 1.0 5.1 0.0
```

As you can see in this example, when coerced to a double, TRUE is represented as 1 and FALSE as 0. That's all for now about atomic vectors. Next, let's discuss another one-dimensional data structure: lists.

19.2.3 Lists

What differentiates lists from atomic vectors are two things. First, lists are heterogeneous, which means its elements can be of different data types. Second, lists are recursive, which means that list elements can contain other objects. Here is an example of a list that shows both of these properties:

```
(my_list <- list(
  a = 3,
  b = "Some text",
  c = list(
    x = c(4, 5, 6)
  ),
  c(TRUE, FALSE)
))
```

```
$a
[1] 3
```

```
$b  
[1] "Some text"
```

```
$c  
$c$x  
[1] 4 5 6
```

```
[[4]]  
[1] TRUE FALSE
```

In this example, the list holds three elements (**a**, **b**, and **c**). Each element is of a different data type: **a** is a numeric vector, **b** is a character vector, and **c** is a list. This is an example of a heterogeneous data structure. Recursion is also demonstrated in this example. The third element **c** contains a list, which itself contains other elements. This is not possible with atomic vectors. Recursive data structures can be extended to any depth desired:

```
list(  
  c = list(  
    x = list(  
      z = list(  
        t = c(1, 2, 3)  
      )  
    )  
  )  
)
```

```
$c  
$c$x  
$c$x$z  
$c$x$z$t  
[1] 1 2 3
```

To better read a deep data structure like this, we can use the **str()** function which provides us with information about the data types of the elements:

```
str(my_list)
```

```
List of 4  
$ a: num 3  
$ b: chr "Some text"
```

```
$ c>List of 1
..$ x: num [1:3] 4 5 6
$ : logi [1:2] TRUE FALSE
```

An even more readable representation of this list is a visual organizer that uses color and shape to indicate the data types of the elements:

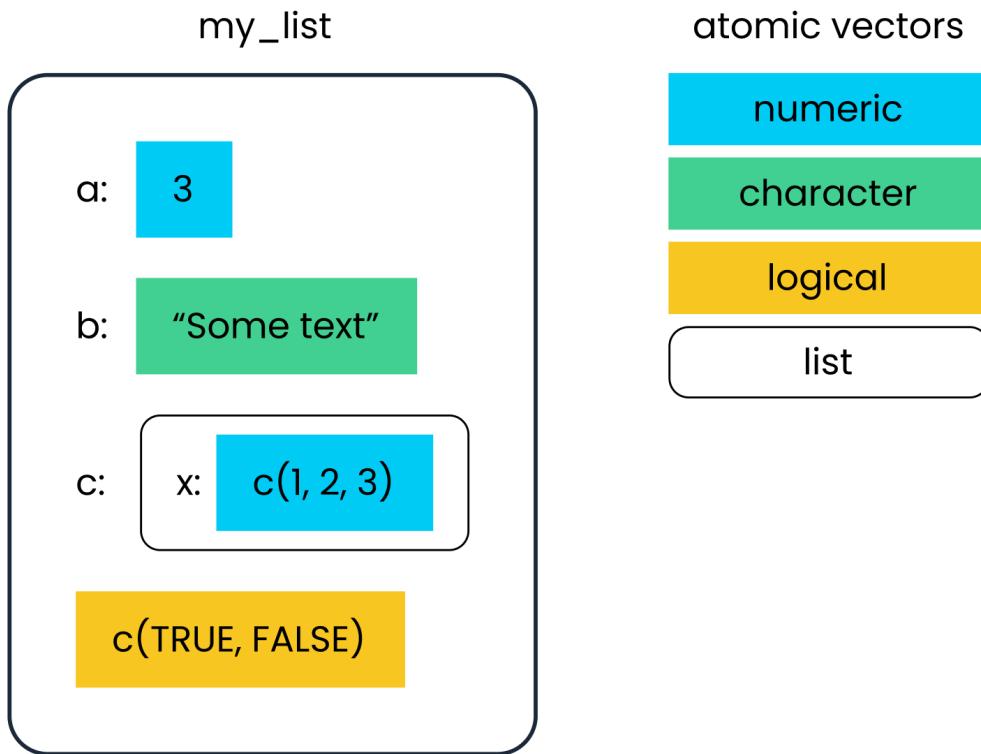


Figure 19.3: Visual depiction of a list

Direct your attention to the list elements. They are represented as white rectangles with black rounded corners. For example, the element labeled **c** is a list. Elements **a** and **x** (within **c**) contain numeric vectors and element **b** holds a character vector.

You may also notice that the last element of the list, the logical vector, does not have a name. This is valid in lists. This leads us to the next topic about lists: subsetting. Subsetting refers to the act of retrieving an element from a list. The topic may be confusing for those who are new to working with lists, so we will look at five ways of doing it first and then discuss them in more detail.

```
my_list[1]
my_list[[1]]
my_list$a
my_list[["a"]]
my_list["a"]
```

Here is one rule to remember. Whenever you subset a list element using one square bracket [, the result will be a list. For example, if I were to subset the first list element **a** using `my_list[1]` or `my_list["a"]`, the output will be a list. Coming back to our visual organizer, the returned list contains a numeric vector:

`my_list[1]`

`my_list["a"]`



We can confirm this by checking the object type:

```
my_list[1] |> typeof()
```

```
[1] "list"
```

```
my_list["a"] |> typeof()
```

```
[1] "list"
```

Here's another rule to remember: When we subset a list element using double square brackets [[, we retrieve the element itself, in our case a double vector:

```
my_list[[1]] |> typeof()
```

```
[1] "double"
```

```
my_list[["a"]] |> typeof()
```

```
[1] "double"
```

Similarly, if a list item has a name, we can directly subset it by using its name, which is equivalent to using [:

```
my_list$a
```

```
[1] 3
```

What I didn't tell you yet is how indexing works in R for lists and vectors. Indexing is a method of arranging the elements in a list or vector and assigning them a numerical value. For example:

```
my_list[[4]]
```

```
[1] TRUE FALSE
```

What we are saying here is this: "Give me the fourth element of that list". In R, lists always start indexing at 1, so `my_list[[1]]` would return the first element of the list. Indexing is crucial for accessing elements that do not have a name. In R, attempting to index an element that is outside the range of a list will result in an error. For instance, in the given example, the list does not have a tenth element.

```
my_list[[10]]
```

```
Error in my_list[[10]] : subscript out of bounds
```

The developers of purrr have addressed this issue by creating the `pluck` function, which functions similarly to using double square brackets `[[` for lists. A useful feature of this function is that it always returns a value, even if the element doesn't exist. For example, if `my_list` only contains four elements, attempting to pick the tenth element would not result in an error. Instead `pluck` returns `NULL` for the element that doesn't exist. Here is an example:

```
pluck(  
  .x = my_list,  
  10  
)
```

```
NULL
```

The first argument `.x` takes a list. All subsequent arguments take indices, which can be either integers or the name of an element. In the following example, I extract the first element, labeled as `x`, of the list `c` within `my_list` (have another look at `my_list` if this is not clear):

```
pluck(my_list, "c", 1)
```

```
[1] 4 5 6
```

This function call is similar to:

```
my_list[["c"]][[1]]
```

```
[1] 4 5 6
```

19.2.4 Data frames

Data frames are a specialized type of list that can hold heterogeneous elements. However, there are two key differences between lists and data frames. Firstly, the elements or columns of a data frame must have equal length. This is not a requirement for list elements. Secondly, data frames are two-dimensional, consisting of rows and columns. They are essentially lists of same-length vectors, as described the book [Advanced R](#). Like lists, data frames can be subsetted by referencing the names of its elements.

```
(my_data_frame <- data.frame(  
  a = c(1, 3, 4),  
  b = c("A", "few", "words"))
```

```
)
```

```
a      b  
1 1      A  
2 3    few  
3 4 words
```

```
my_data_frame$a
```

```
[1] 1 3 4
```

Even the subsetting rules for data frames are similar to those for lists. Using a single set of square brackets will return a list, while double square brackets will return the specific element or column:

```
my_data_frame[1] |> typeof()
```

```
[1] "list"
```

Data frames are composed of columns that are atomic vectors, meaning they consist of a single data type. If one column of a data frame is a list, R will throw an error:

```
data.frame(  
  a = c(1, 3, 4),  
  b = c("A", "few", "words"),  
  c = list(a = 3, b = TRUE, c = c(3, 7))  
)
```

```
Error in data.frame(a = c(1, 3, 4), b = c("A", "few", "words"), c = list(a = 3, :  
arguments imply differing number of rows: 3, 2
```

R will throw an error because the elements in the list are not of the same length as the other columns in the data frame. Even if the column is modified so that the elements have the same length, the output may not be as expected:

```
data.frame(  
  a = c(1, 3, 4),
```

```

b = c("A", "few", "words"),
c = list(a = 3, b = TRUE, c = 7)
)

a      b c.a  c.b c.c
1 1     A   3 TRUE    7
2 3     few  3 TRUE    7
3 4 words 3 TRUE    7

```

Interestingly, the column `c` was spread across three columns `c.a`, `c.b`, and `c.c`. The code worked, but the list was flattened into these three columns. Despite being widely used in R for over 20 years, data frames lack some properties that make data analysis easier. To address this, the creators of the Tidverse package introduced tibbles. You can think of tibbles as better looking and better behaving data frames.

If the same data frame was created as a tibble, it would be called a **nested tibble**. These nested tibbles will be important later in the tutorial when we will introduce the map functions. The next example shows a nested tibble, created by replacing `data.frame` with `tibble` in the previous code.

```

(my_tibble <- tibble(
  a = c(1, 3, 4),
  b = c("A", "few", "words"),
  c = list(a = 3, b = TRUE, c = c(3, 7))
))

# A tibble: 3 x 3
  a     b     c
  <dbl> <chr> <named list>
1     1 A     <dbl [1]>
2     3 few   <lgl [1]>
3     4 words <dbl [2]>

```

The result is a column `c` that holds a list. This column behaves similarly to a list, with the same subsetting and indexing methods we have covered previously.

```

# Returns list element as list of length 1
my_tibble$c[1]
my_tibble$c["a"]

# Returns list element

```

```
my_tibble$c[[1]]  
my_tibble$c$a  
pluck(my_tibble$c, 1)
```

With this, you should now have the necessary vocabulary and concepts to get started with the map family of functions. It is important to ensure that you have fully grasped the concepts discussed up to this point, as the rest of the tutorial will build upon these foundations.

19.3 For loops in R

At the start of the tutorial, I mentioned that the map family of functions iterate over elements of certain data types. For most learners, the first technique for iterating over elements is the for loop. This is because for loops are explicit, showing each computational step clearly, and are a fundamental concept that all programmers should understand. To provide a clear connection between for loops and the map functions, I will briefly introduce for loops at this point in the tutorial.

As an example, suppose you want to use a for loop to calculate the square of each element in a numeric vector:

```
.input <- c(2, 4, 6)  
output <- vector(mode = "numeric", length = length(.input))  
  
for (i in seq_along(.input)) {  
  output[[i]] <- .input[[i]]^2  
}  
  
output
```

```
[1] 4 16 36
```

.input is an atomic vector of type numeric or integer. output is an empty atomic vector of type numeric with the same length as .input. The length parameter is used to allocate a specific amount of space for that vector. Initially, output is empty. The for loop uses seq_along(), which creates a sequence of values from 1 to the number of elements in .input. For example, if input has three elements, the sequence would be 1, 2, 3.

```
seq_along(.input)
```

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

Within the for loop, `i` corresponds to the current element of the sequence created by `seq_along()`.

```
for (i in seq_along(.input)) {  
  print(i)  
}
```

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

Within the body of the for loop, we calculate the square of each vector element and store the result at the same index in the output vector (`output[[i]] <- .input[[i]]^2`).

For loops and the `map` functions have some similarities and can be easily translated between, as demonstrated in the [loopurrr package](#) created by Tim Tiefenbach. They share the following characteristics:

- Both for loops and map functions take input in the form of a list, vector, data frame, or tibble.
- They both iterate over each element of these data structures and apply a computation to that element.
- They both create or return a new object.

It's important to note that these similarities only apply to the specific for loop we just created. Now that we have covered for loops, we are ready to start working with map functions.

19.4 Introduction of the map family of functions

`map` is a family of functions, consisting of at least five different functions. Similar to for loops, the map functions iterate over elements, apply a function to each element, and return the results in a data structure of your choice. The following is a visual overview of the `map` functions:

Two arguments are essential for the map functions:

- `.x`: This is either a list or an atomic vector. Data frames can also be used as input, in which case `map` will iterate over each column of the data frame.
- `.f`: A function provided to `.f` will be applied to each element of `.x`. It is considered good practice to write this function in the form of an anonymous R function (`\(x) <FUNCTION BODY>`) instead of using the tilde `~` which is considered bad practice.

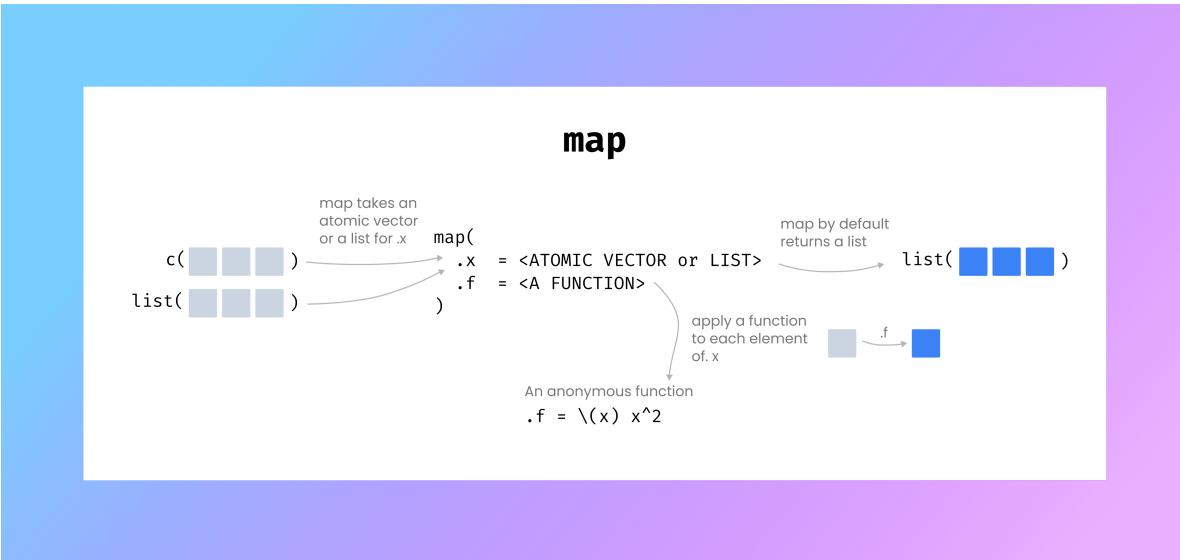


Figure 19.4: map function overview

It's important to note that `map` will always return an object that is the same length as the input. With this general understanding, let's see an example of `map`. Similar to the for loop previously, we will square each element in a numeric vector.

```
(first_map_output <- c(1, 2, 3) |> map(.f = \((x) x^2))
```

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

```
[[2]]
[1] 4
```

```
[[3]]
[1] 9
```

In this example, the `.x` argument is implicitly passed to `map`, which is why it is not visible in the code. The output has the same length as the input (3 items) and it is returned as a list:

```
first_map_output |> typeof()
```

```
[1] "list"
```

What is great about the `map` functions is that the returned object does not have to be a list. The family includes different functions that specify the type of data type for the returned object by using suffixes. Here are the functions:

- `map_chr`: Returns a character vector
- `map_int`: Returns an integer vector
- `map_dbl`: Returns a double vector
- `map_lgl`: Returns a logical vector

These are the same data types that were initially introduced as atomic vectors.

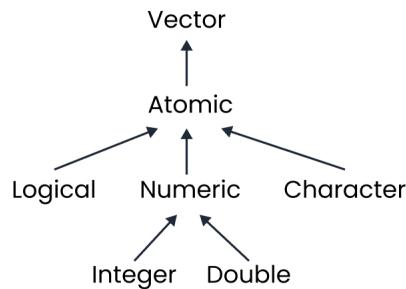


Figure 19.5: Atomic vectors

In our output `first_map_output`, we know that all elements in the list are numerics, specifically doubles.

```
first_map_output[[1]] |> typeof()
```

```
[1] "double"
```

Since the list elements are all of the same type, we can create a homogeneous atomic double vector instead of a list. To accomplish this, we can use the `map_dbl` function.

```
(map_output_double <- c(1, 2, 3) |> map_dbl(.f = \((x) x^2))
```

```
[1] 1 4 9
```

Indeed, the output is a vector of doubles:

```
map_output_double |> typeof()
```

```
[1] "double"
```

In contrast to `map`, the other functions in the `map` family create homogeneous atomic vectors. However, this can cause issues in practice. For example, if we want the `map_lgl` function to return a logical vector, and the input vector only contains the numbers 0 and 1:

```
c(0, 1, 1) |> map_lgl(.f = \((x) x^2)
```

```
[1] FALSE TRUE TRUE
```

In this case, it works as expected. As previously discussed, coercion rules convert the numbers 0 and 1 to `TRUE` and `FALSE`, and vice versa. However, `map_lgl()` doesn't allow for the coercion of numbers other than 0 and 1 to logicals.

```
c(0, 1, 2) |> map_lgl(.f = \((x) x^2)
```

```
Error in `map_lgl()`:  
In index: 3.  
Caused by error:  
! Can't coerce from a double vector to a logical vector.  
Run `rlang::last_error()` to see where the error occurred.
```

An example of another error that is thrown because the coercion rules were not followed is this:

```
c(1, 2, 3) |> map_dbl(.f = \((x) list(x))
```

```
Error in `map_dbl()`:  
In index: 1.  
Caused by error:  
! Can't coerce from a list to a double vector.  
Backtrace:  
1. purrr::map_dbl(c(1, 2, 3), .f = function(x) list(x))  
2. purrr:::map_("double", .x, .f, ..., .progress = .progress)  
3. purrr:::with_indexed_errors(...)  
4. base::withCallingHandlers(...)
```

This code doesn't work because `map dbl` has no idea how to convert a list to a double (that's exactly what the error says if you look closely).

So, the key takeaway from these examples is to always be aware of what kind of output you're expecting when using the `map` functions. You will usually receive an error when you break the coercion rules of atomic vectors and lists. Which brings us to the next topic.

19.5 Using map with a list as input

In the examples shown before, the input was an atomic vector and the output was either an atomic vector or a list. However, this isn't always the case. Lists are another common input for the `map` functions. Let me show you an example:

```
list(x = 1, y = 2, z = 3) |> map_dbl(.f = \(x) x)
```

```
x y z
1 2 3
```

What's interesting about this example is that we didn't have to subset the list element! In other words, `x` in the anonymous function `.f` represents the list element. Subsetting becomes necessary only when the input list contains list elements, as shown in this example:

```
list(
  x = list(1),
  y = list(4),
  z = list(4)
) |>
  map_dbl(.f = \(x) x[[1]])
```

```
x y z
1 4 4
```

Another important point to remember is that when we iterate over a list, we iterate through its individual elements. This holds true even if the elements themselves are more complex data structures like tibbles:

```
list(
  tibble1 = tibble(id = c(1, 2, 3), value = c(4, 5, 6)),
  tibble1 = tibble(id = c(1, 2, 3), value = c(8, 10, 33))
```

```
) |> map(  
  .f = \((x) x$value  
)
```

```
$tibble1  
[1] 4 5 6
```

```
$tibble1  
[1] 8 10 33
```

As the input list comprises of tibbles, `x` within the anonymous function refers to a tibble. Columns of tibbles can be accessed using the `$` operator, thus `x$value` returns the `value` column from each tibble within the list.

Working with lists and using the `map` function can become more challenging when each element in the list contains a different data structure. In this example, the first element is a tibble and the second an atomic vector of type numeric:

```
list(  
  tibble1 = tibble(id = c(1, 2, 3), value = c(4, 5, 6)),  
  vector1 = c(3, 4, 5)  
) |> map(  
  .f = \((x) {  
    if (is_tibble(x)) (  
      return(x$value)  
    )  
  
    x  
  }  
)
```

```
$tibble1  
[1] 4 5 6
```

```
$vector1  
[1] 3 4 5
```

The `map` function returns the `value` column from the tibble and `vector1` as well. To extract what is needed, I have used an if statement in the anonymous function's body.

Similarly, if statements can be used to avoid throwing errors. In this example, no error is thrown as the log of the character vector is not computed and instead the function returns an NA.

```
list(
  v1 = c(3, 4, 5),
  v2 = c(8, 22, 34),
  v3 = c("a", "b", "c")
) |> map(
  .f = \x) {
  if(is.numeric(x)) (
    return(log(x))
  ) else if (is.character(x)) (
    return(NA)
  )
}
```

```
$v1
[1] 1.098612 1.386294 1.609438
```

```
$v2
[1] 2.079442 3.091042 3.526361
```

```
$v3
[1] NA
```

19.6 Error handling with `safely` and `possibly`

Error handling can be done better, however. The `safely` and `possibly` functions both deal with errors but handle them in different ways:

- `safely` creates a version of `.f` that always runs successfully. It returns a list with two elements: the result and the error. If the function call works, it returns the value, the error will be `NULL`. If an error occurs, the result will be `NULL` and error contains the error object.
- `possibly` modifies `.f` so that it returns a default value whenever an error occurs.

19.6.1 safely

Here's an example of using `safely`. The list contains two numerics and a character. The `map` function should calculate the log of each list element. Since you can't compute logs for characters, this function call would result in an error. But, if we wrap `log` in `safely`, `map` returns a list:

```
safe_log <- safely(log)

(log_res <- list(2, 3, "A") |>
  map(.f = \((x) safe_log(x)))
```



```
[[1]]
[[1]]$result
[1] 0.6931472

[[1]]$error
NULL

[[2]]
[[2]]$result
[1] 1.098612

[[2]]$error
NULL

[[3]]
[[3]]$result
NULL

[[3]]$error
<simpleError in .Primitive("log")(x, base): non-numeric argument to mathematical function>
```

Each element in the returned list has two elements: `result` and `error`. From the output, we can see two things. The third element under `result` shows that `map` computed `NULL` for A. Also, the third element contains an error object under `$error`.

At first glance, it may seem like you could use another `map` function to return only the `result` elements from the list, but it's more complicated than it appears, because this doesn't work in this example.

```
log_res |> map_dbl(.f = \((x) x$result)
```

```
Error in `map_dbl()`:  
In index: 3.  
Caused by error:  
! Result must be length 1, not 0.  
Backtrace:  
1. purrr::map_dbl(log_res, .f = function(x) x$result)  
2. purrr:::map_("double", .x, .f, ..., .progress = .progress)  
3. purrr:::with_indexed_errors(...)  
4. base::withCallingHandlers(...)
```

Do you remember that we said that the length of the output of `map` will always be of the same length than the input? However, this is not the case here and that's why we get an error. The issue is that if you convert `NULL` to a number you get a numeric of length 0:

```
as.double(NULL) |> length()
```

```
[1] 0
```

To overcome this problem, we need to replace `NULL` with `NA`, since `NA` has a length of 1.

```
as.double(NA) |> length()
```

```
[1] 1
```

Luckily, `safely` has an argument called `otherwise` which allows you to assign a default value for elements that result in an error (the default value could also be a number, of course):

```
safe_log_na <- safely(log, otherwise = NA)

log_res_na <- list(2, 3, "A") |>
  map(.f = \((x) safe_log_na(x))

log_res_na |>
  map_dbl(.f = \((x) x$result)
```

```
[1] 0.6931472 1.0986123      NA
```

19.6.2 possibly

`possibly` takes a different approach than `safely`. Unlike `safely` it doesn't create the sub-elements `result` and `error`. Instead, it creates default values for elements that throw an error. In this example, whenever an error occurs, `.f` returns `NA`:

```
log Possibly <- possibly(log, otherwise = NA)

list(2, 3, "A") |>
  map_dbl(.f = \((x) log Possibly(x))
```

```
[1] 0.6931472 1.0986123      NA
```

19.7 map_vec

I've not been entirely honest with you. The `map` family of functions has another function called `map_vec`. It's a new function introduced in purrr 1.0.0. You might have noticed that I haven't discussed factors or dates so far. These data types are vectors, built on top of atomic vectors. `map_vec` is specifically designed for these types of vectors.

We can show that factors or dates are atomic vectors by checking their data type. Dates, for instance, are doubles in disguise:

```
as.Date('2022-01-31') |> typeof()
```

```
[1] "double"
```

And factors are integers:

```
as.factor(c(1, 2, 3)) |> typeof()
```

```
[1] "integer"
```

Now, suppose you have created a vector of dates:

```
c(as.Date('2022-01-31'), as.Date('2022-10-12'))
```

```
[1] "2022-01-31" "2022-10-12"
```

You would like to increment each date in the vector by one month. For example, November 12, 2022 should become December 12, 2022. The lubridate package offers the `%m+%` operator and the `period()` function which in combination enable to increase dates:

```
library(lubridate)

as.Date('2022-11-12') %m+% months(1)

[1] "2022-12-12"
```

If you were to use `map` to increment each date in the vector, you would get a list as output, but not a date vector:

```
c(as.Date('2022-01-31'), as.Date('2022-10-12')) |>
  map(.f = \((x) x %m+% months(1))
```

```
[[1]]
[1] "2022-02-28"

[[2]]
[1] "2022-11-12"
```

The function `map_vec` was created so that you can iterate over datetime, date, and factor vectors and get the same type of vector in return. So, if we use `map_vec` instead of `map` in our example, the output of `map_vec` would be a date vector:

```
c(as.Date('2022-01-31'), as.Date('2022-10-12')) |>
  map_vec(\(x) x %m+% months(1))
```

```
[1] "2022-02-28" "2022-11-12"
```

Another useful application of `map_vec` is generating a series of dates within a specified range. For example, it can be used to create a vector of the same date for the next 50 years:

```
1:50 |>
  map_vec(\(x) as.Date(ISOdate(x + 2023, 11, 12)))
```

```
[1] "2024-11-12" "2025-11-12" "2026-11-12" "2027-11-12" "2028-11-12"
[6] "2029-11-12" "2030-11-12" "2031-11-12" "2032-11-12" "2033-11-12"
[11] "2034-11-12" "2035-11-12" "2036-11-12" "2037-11-12" "2038-11-12"
[16] "2039-11-12" "2040-11-12" "2041-11-12" "2042-11-12" "2043-11-12"
[21] "2044-11-12" "2045-11-12" "2046-11-12" "2047-11-12" "2048-11-12"
[26] "2049-11-12" "2050-11-12" "2051-11-12" "2052-11-12" "2053-11-12"
[31] "2054-11-12" "2055-11-12" "2056-11-12" "2057-11-12" "2058-11-12"
[36] "2059-11-12" "2060-11-12" "2061-11-12" "2062-11-12" "2063-11-12"
[41] "2064-11-12" "2065-11-12" "2066-11-12" "2067-11-12" "2068-11-12"
[46] "2069-11-12" "2070-11-12" "2071-11-12" "2072-11-12" "2073-11-12"

# map_vec(\(x) ymd(paste(x + 2022, "-10", "-12"))) # alternative solution
```

19.8 Using map with nested data frames

A common use case for the `map` functions is creating new columns in nested data frames. Many beginners find it hard to work with nested data frames. But keep in mind that a nested column in a nested data frame is nothing but a list. In the following example, we have nested the `diamonds` data frame, excluding the `cut` of the diamonds:

```
(nested_cuts <- diamonds |>
  nest(data = -cut))

# A tibble: 5 x 2
  cut      data
  <ord>    <list>
1 Ideal   <tibble [21,551 x 9]>
2 Premium <tibble [13,791 x 9]>
3 Good    <tibble [4,906 x 9]>
4 Very Good <tibble [12,082 x 9]>
5 Fair    <tibble [1,610 x 9]>
```

Have a look at the `data` column. It is a list. So everything we have learned in the chapter on lists can be applied when using the `mutate()` function in conjunction with `map()`. Before we go through an example, let's have a look at the general structure of combining `mutate()` and `map()`:

```
<DATAFRAME> |>
  mutate(
```

```
<NEW_COLUMN_NAME> = map(<COLUMN_NAME>, .f = \((x) <FUNCTION_CALL>)
)
```

It's crucial to remember that nested columns are treated as lists. Therefore, when iterating through a nested column, `<COLUMN_NAME>` represents a list and `x` represents the individual element of that list. When using the `mutate()` function in combination with `map()`, the function is applied to each row of the data frame, allowing iteration through each element of the list. The output of the `map` function will either be a list or an atomic vector, which will be the value for each corresponding row in the newly created column `<NEW_COLUMN_NAME>`.

In the context of the diamond example, if we were to use `map()` to iterate over the values in the `data` column and return only the individual element of that list, we would duplicate the `data` column to a new column called `data_copied`.

```
nested_cuts |>
  mutate(
    data_copied = map(data, .f = \((x) x)
  )

# A tibble: 5 x 3
  cut      data           data_copied
  <ord>    <list>        <list>
1 Ideal   <tibble [21,551 x 9]> <tibble [21,551 x 9]>
2 Premium <tibble [13,791 x 9]> <tibble [13,791 x 9]>
3 Good    <tibble [4,906 x 9]>  <tibble [4,906 x 9]>
4 Very Good <tibble [12,082 x 9]> <tibble [12,082 x 9]>
5 Fair    <tibble [1,610 x 9]>  <tibble [1,610 x 9]>
```

Instead of returning the tibble in `.f`, we can perform any operation on the tibble `x` that is possible with tibbles. For instance, we could extract the number of rows from each tibble and store them in a new column as an integer value:

```
nested_cuts |>
  mutate(
    n_rows = map_int(data, .f = \((x) nrow(x))
  )

# A tibble: 5 x 3
  cut      data           n_rows
  <ord>    <list>        <int>
1 Ideal   <tibble [21,551 x 9]> 21551
```

```

2 Premium    <tibble [13,791 x 9]> 13791
3 Good       <tibble [4,906 x 9]>   4906
4 Very Good  <tibble [12,082 x 9]> 12082
5 Fair        <tibble [1,610 x 9]>   1610

```

A closer examination of the code will reveal that I used `map_int()` instead of `map()` because I knew the output would be an atomic vector of type integer. In this case, the `nrow()` function is being used to extract the number of rows from each tibble. However, it is also possible to chain multiple computations together using the pipe operator to transform tibbles and store them in a new column:

```

(nested_new_data <- nested_cuts |>
  mutate(
    new_data = map(data, .f = \((x)  x |>
      filter(price > 8000) |>
      select(depth, price, x, y, z)
    )
  )
)

# A tibble: 5 x 3
  cut      data          new_data
  <ord>    <list>        <list>
  1 Ideal   <tibble [21,551 x 9]> <tibble [2,684 x 5]>
  2 Premium <tibble [13,791 x 9]> <tibble [2,503 x 5]>
  3 Good    <tibble [4,906 x 9]>  <tibble [572 x 5]>
  4 Very Good <tibble [12,082 x 9]> <tibble [1,645 x 5]>
  5 Fair    <tibble [1,610 x 9]>  <tibble [201 x 5]>

```

The following examples demonstrate the capabilities of working with nested columns using the `purrr` package. We will delve deeper into these techniques in a later chapter. One powerful feature is the ability to store plots within a column.

For instance, suppose you wanted to create a scatterplot from each tibble in the `data` column. Within each of these tibbles, the `x` column represents the length of the diamond in millimeters and the `y` column represents the width of the diamond in millimeters. These two columns can be plotted on the x- and y-axis, respectively, to create the scatterplot:

```

(plots <- nested_cuts |>
  mutate(
    scatterplots_x_y = map(data, .f = \((x) {
      x |>
      ggplot(aes(x = x, y = y)) +

```

```

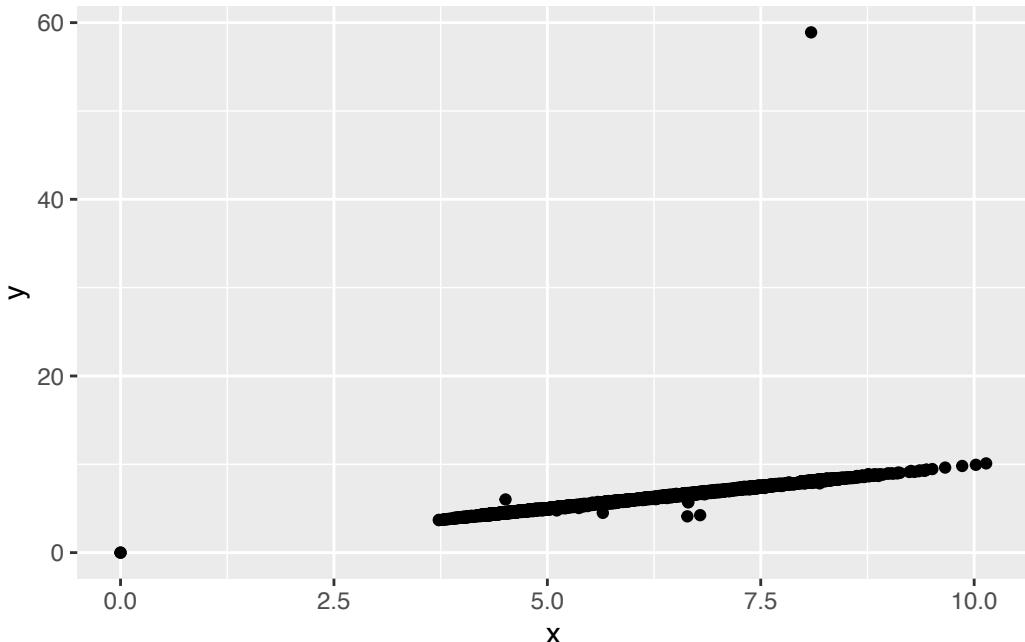
        geom_point()
    })
))

# A tibble: 5 x 3
  cut      data      scatterplots_x_y
  <ord>    <list>    <list>
1 Ideal    <tibble [21,551 x 9]> <gg>
2 Premium  <tibble [13,791 x 9]> <gg>
3 Good     <tibble [4,906 x 9]> <gg>
4 Very Good <tibble [12,082 x 9]> <gg>
5 Fair     <tibble [1,610 x 9]> <gg>

```

The tibble has been modified by the addition of a new column called `scatterplot_x_y`. Each value in this column contains a `ggplot2` object, represented as `S3:gg`. The plots can be pre-viewed by subsetting a specific value from the `scatterplot_x_y` column:

```
plots$scatterplots_x_y[[2]]
```



`S3:gg` objects are not the only type of objects that can be created using `map()` and `mutate()`. Another application of these two functions is fitting models to our data and storing the results

in a new column. For example, we could use `map()` and `mutate()` to fit a linear regression model to the `x` and `y` columns and store the model output in a new column:

```
(models <- nested_cuts |>
  mutate(
    fitted_model = map(data, \((x) {
      lm(x ~ y, data = x)
    })
  ))
```



```
# A tibble: 5 x 3
  cut      data          fitted_model
  <ord>    <list>        <list>
1 Ideal   <tibble [21,551 x 9]> <lm>
2 Premium <tibble [13,791 x 9]> <lm>
3 Good    <tibble [4,906 x 9]> <lm>
4 Very Good <tibble [12,082 x 9]> <lm>
5 Fair    <tibble [1,610 x 9]> <lm>
```

Models created with the function `lm()` are stored as SE: `lm` objects. For those with a background in statistics, it may be of interest to extract the effect size R^2 from these models. Within `map()`, we can call the `summary()` function on the `lm` object and access the `r.squared` attribute to extract the value of R^2 .

```
models |>
  mutate(
    r_squared = map_dbl(fitted_model, \((x) summary(x)$r.squared)
  ))
```



```
# A tibble: 5 x 4
  cut      data          fitted_model r_squared
  <ord>    <list>        <list>        <dbl>
1 Ideal   <tibble [21,551 x 9]> <lm>       0.968
2 Premium <tibble [13,791 x 9]> <lm>       0.880
3 Good    <tibble [4,906 x 9]> <lm>       0.996
4 Very Good <tibble [12,082 x 9]> <lm>       0.998
5 Fair    <tibble [1,610 x 9]> <lm>       0.988
```

We will delve much deeper into this concept in a later chapter. However, before we do so, it is important to become familiar with the more advanced `map2()` and `pmap()` functions, which will be covered in the next two chapters.

19.9 Using map with a list of data frames

There is a difference whether we iterate over the values of a column in a data frame, a single data frame, or multiple data frames. We have just seen the first case, where we iterate over the columns of a single data frame using `map()`. Now, let's discuss the last case, which is iterating over a list of data frames or tibbles. This is a common scenario when multiple data frames have been loaded into memory. To illustrate this, let's create a list containing three tibbles:

```
list_of_dframes <- list(
  d1 = tibble(
    d = c(1, 1, 1),
    p = c(34, 21, 9)
  ),
  d2 = tibble(
    d = c(2, 2, 2),
    p = c(21, 2, 76)
  ),
  d3 = tibble(
    d = c(3, 3, 3),
    p = c(54, 26, 11)
  )
)
```

Suppose you want to square the values in each `p` column. As before, we can use `map()` to perform this computation within the `.f` function.

```
list_of_dframes |>
  map(.f = \(dframe) {
    dframe |>
      mutate(p = p^2)
  })
```

```
$d1
# A tibble: 3 x 2
      d     p
<dbl> <dbl>
1     1   1156
2     1    441
3     1     81

$d2
# A tibble: 3 x 2
```

```

      d      p
<dbl> <dbl>
1     2    441
2     2      4
3     2   5776

$d3
# A tibble: 3 x 2
      d      p
<dbl> <dbl>
1     3   2916
2     3    676
3     3    121

```

Now, suppose you want to combine these three data frames into one data frame. Prior to purrr 1.0.0, the function `map_dfr()` was used to bind the data frames automatically. However, with the release of purrr 1.0.0, the new function `list_rbind()` has been introduced, which combines data frames stored in a list.

```

list_of_dframes |>
  map(.f = \(dframe) {
    dframe |>
      mutate(p = p^2)
  }) |>
  list_rbind()

# A tibble: 9 x 2
      d      p
<dbl> <dbl>
1     1   1156
2     1    441
3     1     81
4     2    441
5     2      4
6     2   5776
7     3   2916
8     3    676
9     3    121

```

Luckily, the `d` columns provides information about the origin of the data frames. This technique of binding data frames is particularly useful when multiple data frames are read into memory and need to be concatenated into a single data frame

Summary

- The `map` family of functions are used to iterate over the elements of atomic vectors, lists, and data frames and perform computations on them.
- Atomic vectors are homogeneous and one-dimensional data structures, while lists are heterogeneous, containing different types of data structures. Data frames are a specific type of list, where each element has the same length and data type.
- The `map` family of functions includes six functions: `map`, `map_lgl`, `map_int`, `map_dbl`, `map_chr`, and `map_vec`. The suffices `_lgl`, `_int`, `_dbl`, `_chr`, and `_vec` indicate the output data type of the function.
- When using one of the `map` functions in combination with `mutate` on a nested data frame, a nested column can be treated as a list.
- Nested data frames in combination with `mutate` and `map` are useful for creating plots or fitting models to the nested data.

20 How to use the `map2` and `pmap` function family effectively

What will this tutorial cover?

This tutorial builds upon the `map()` tutorial by showing how `map2()` and `pmap()` can be used to iterate over multiple values at the same time. We will delve deeply into both function families and then explore several practical applications.

Who do I have to thank?

For this tutorial, I'd like to thank the Twitter community for providing me with insightful ideas. I particularly thank [Marc Ruben](#), [no_one](#), [Brenton Wiernik](#), [bing_bong_telecom](#), and [Colin Fay](#)

20.1 An overview of `map2` and `pmap`

The `map` function family applies a computation to a single element, while `map2` and `pmap` apply a computation to multiple elements simultaneously. In the previous chapter, for example, we used `map` to generate a column of plots. However, without descriptive titles, these plots were difficult to differentiate. With `map2` and `pmap`, we can create plots that create plots from both the data and a name for the plot's title.

As an example, consider the scenario of simulating data. If you want to simulate the rolling of a dice, one simulated set could consist of numbers between 1 and 6, while another set could consist of numbers between 4 and 6. To simulate many of these sets with varying parameters, we need `map2` or `pmap`. Or more generally, whenever we need to perform computations on multiple values, these two functions are useful.

20.2 Introduction to `map2`

Let's begin with a discussion on `map2`. To better understand its functionality, let me show you a visual representation of the function:

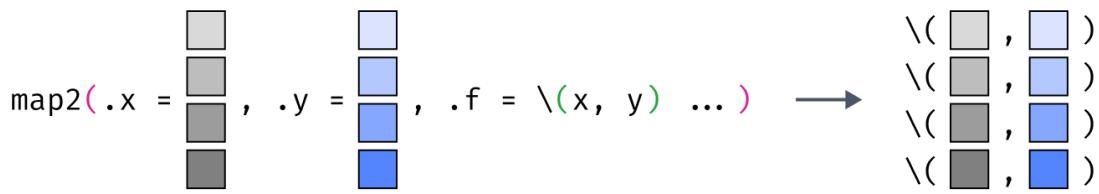


Figure 20.1: map2

`map2` takes in three parameters: `.x`, `.y`, and `.f`. The first two parameters, `.x` and `.y`, can be either atomic vectors or lists. These arguments are then passed to the anonymous function `.f`. It's important to note that the lengths of both `.x` and `.y` must match, or one of them must have a length of 1.

Let's consider a scenario where you want to compute the element-wise sum of two vectors. While this can be easily achieved using `sum()` (e.g. `c(1, 4) + c(4, 6)`), let's try to approach this problem from a different perspective and use `map2` instead:

```
map2(
  .x = c(1, 4),
  .y = c(4, 6),
  .f = \((x, y) x + y
)
```

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

```
[[2]]
[1] 10
```

To make the inner workings of the anonymous function `.f` more explicit, we can add a `print` statement:

```
library(glue)

map2(
  .x = c(1, 4),
  .y = c(4, 6),
  .f = \((x, y) {
```

```
    print(glue("{x} plus {y} equals = {x + y}"))
    x + y
  }
)
```

```
1 plus 4 equals = 5
4 plus 6 equals = 10
```

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

```
[[2]]
[1] 10
```

As you can observe, `.f` starts by processing the first element of both vectors, followed by the second element of both vectors. This is why it's crucial for both vectors to have the same length. If the lengths differ, `map2` throws an error:

```
map2(
  .x = c(1, 4),
  .y = c(4, 6, 9),
  .f = \((x, y) x + y
)
```

```
Error in `map2()`:
! Can't recycle `^.x` (size 2) to match `^.y` (size 3).
Backtrace:
1. purrr::map2(.x = c(1, 4), .y = c(4, 6, 9), .f = function(x, y) x + y)
```

The error message tells us that the sizes of the vectors are not equal (size 2 vs size 3). Interestingly, if the length of one of the vectors is 1, we won't get an error:

```
map2(
  .x = c(1, 4),
  .y = c(4),
  .f = \((x, y) x + y
)
```

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

```
[[2]]  
[1] 8
```

If one vector has a size of 1, `map2` will recycle the single element from that vector for each element in the other vector:

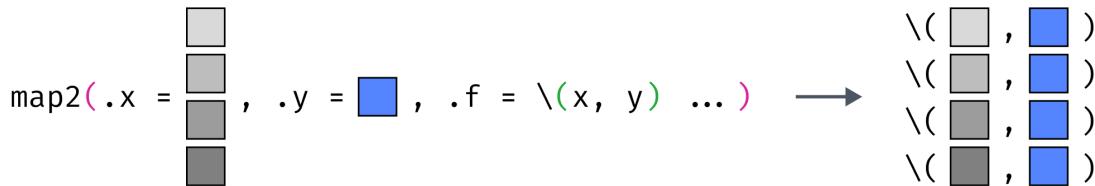


Figure 20.2: `map2` with one object of size one

Up until now, `map2` has produced a list as output, similar to what one would expect from `map`. Thankfully, the creators of the `purrr` package have implemented a consistent grammar for its functions, allowing us to use the same suffixes as in `map`. For example, to convert the output to a vector of doubles, we simply need to change the function name to `map2_dbl`:

```
map2_dbl(  
  .x = c(1, 4),  
  .y = c(4, 6),  
  .f = \(\(x, y) x + y  
)
```

```
[1] 5 10
```

Now that we have explored how to change the data type of the output, let's look at how to work with input that consists of both lists and vectors. To start, we will consider a simple example using a list with two elements and a vector with two elements:

```
map2_dbl(  
  .x = list(x = 1, y = 7),  
  .y = c(2, 3),  
  .f = \(\(x, y) x + y
```

```
)
```

```
x  y  
3 10
```

We can observe that `map2` returns a named atomic vector with names `x` and `y`. Next, let me demonstrate an error that occurs when using a nested list instead:

```
map2_dbl(  
  .x = list(list(5), y = 7),  
  .y = c(2, 3),  
  .f = \((x, y) x + y  
)
```

```
Error in `map2_dbl()`:  
In index: 1.  
Caused by error in `x + y`:  
! non-numeric argument to binary operator  
Backtrace:  
1. purrr::map2_dbl(...)  
2. purrr:::map2_("double", .x, .y, .f, ..., .progress = .progress)  
6. global .f(.x[[i]], .y[[i]], ...)
```

The code fails during the first iteration. The function attempts to compute `list(5) + 2`, which results in this cryptic error message:

```
Error in list(5) + 2 : non-numeric argument to binary operator
```

The binary operator in question is the plus operator `+`, and the non-numeric argument is the list `list(5)`. To resolve this error, a simple solution would be to convert the list into a double:

```
map2_dbl(  
  .x = list(list(5), y = 7),  
  .y = c(2, 3),  
  .f = \((x, y) as.double(x) + y  
)
```

```
y  
7 10
```

Clearly, the naming of the vector elements is not beneficial. We can remove the names by using the `unnname()` function:

```
map2_dbl(  
  .x = list(list(5), y = 7),  
  .y = c(2, 3),  
  .f = \((x, y) as.double(x) + y  
) |> unname()
```

```
[1] 7 10
```

Here is another mental exercise. In the following example, we will input an atomic vector and a data frame. Our goal is to calculate the sum of each element in the vector and the length of the data frame.

```
map2_dbl(  
  .x = c(1, 2, 3),  
  .y = mpg,  
  .f = \((x, y) x + nrow(y)  
)  
  
Error in `map2_dbl()`:  
! Can't recycle `.x` (size 3) to match `.y` (size 11).  
Backtrace:  
1. purrr::map2_dbl(.x = c(1, 2, 3), .y = mpg, .f = function(x, y) x + nrow(y))
```

I intentionally caused this error to demonstrate that `mpg` is not a list or data frame of length 1, and therefore it cannot be recycled for each element in the vector `c(1, 2, 3)`. In the previous chapter, we learned that a data frame is a special type of list, and each column in the data frame is essentially a list element. The error occurs because the lengths of both objects are different, with the atomic vector having a length of 3 and the data frame having a length of 11. This discrepancy cannot be resolved by `map2`.

Fortunately, there are solutions to this issue. One approach is to use `map` instead and directly reference `mpg` within the function.

```
map_dbl(  
  .x = c(1, 2, 3),  
  .f = \((x, y) x + nrow(mpg)  
)
```

```
[1] 235 236 237
```

Or, we can wrap the data frame in a list of length 1, which allows it to be recycled for each element in the atomic vector `c(1, 2, 3)`:

```
map2_dbl(  
  .x = c(1, 2, 3),  
  .y = list(mpg),  
  .f = \((x, y) x + nrow(y)  
)
```

```
[1] 235 236 237
```

20.3 Introduction to pmap

`map2` reaches its limits when you need to handle more than two arguments for an iteration. Simulations provide a good illustration of this scenario. For instance, consider the scenario where you need to generate random data sets for dice rolls with the following parameters:

```
(dice_rolls <- crossing(  
  size = c(10, 100),  
  min = c(1, 2),  
  max = c(6  
))  
  
# A tibble: 4 x 3  
  size   min   max  
  <dbl> <dbl> <dbl>  
1    10     1     6  
2    10     2     6  
3   100     1     6  
4   100     2     6
```

For the simulated data sets, there are three parameters that must be varied. These include the size of the set, the minimum number on the dice, and the maximum number on the dice. For instance, if you wanted to generate a set with the parameters `size = 10`, `min = 1`, and `max = 6`, you would run the `sample()` function as follows:

```

size <- 10
min <- 1
max <- 6

sample(min:max, size, TRUE)

```

```
[1] 6 4 3 2 1 2 4 5 5 1
```

`map2()` is unable to simulate the desired data sets for all parameter combinations. To accomplish this, you need to use the `pmap()` instead.

The key difference between `map2` and `pmap` is that `pmap` has the `.l` argument, which takes a list. The anonymous function `.f` should have as many arguments as the elements in the list provided to `.l`. To offer a different perspective on `pmap()`, here is a visual illustration of the function:

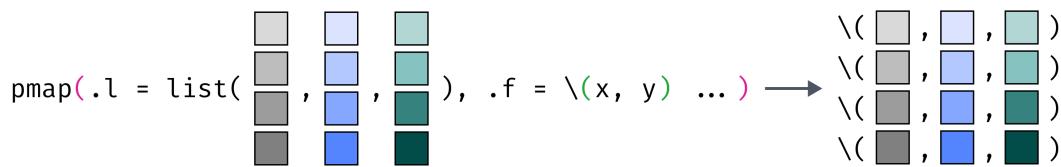


Figure 20.3: `pmap`

The three objects in the visual is just an example. You can have as many as you like. A practical example of this illustration could look something like this:

```

pmap_dbl(
  .l = list(
    c(1, 3, 4),
    c(5, 6, 7),
    c(8, 3, 9)),
  .f = \({el1, el2, el3} {
    print(glue("{el1} plus {el2} plus {el3} equals = {el1 + el2 + el3}"))
    el1 + el2 + el3
  }
)

```

```
1 plus 5 plus 8 equals = 14
```

```
3 plus 6 plus 3 equals = 12
4 plus 7 plus 9 equals = 20
```

```
[1] 14 12 20
```

I have made a few modifications to this function to illustrate two things.

- Take a look at the names of the arguments in the anonymous function (`e11`, `e12`, `e13`). In `pmap`, you have the freedom to assign any names to these arguments. Just make sure that the anonymous function has at least as many arguments as the number of elements in the list in `.l` (which is three in this example).
- Then, I have included a `print` statement to show which elements are processed for each iteration.

This code produces the same result as if you were to sum the three vectors using the `+` operator:

```
c(1, 3, 4) + c(5, 6, 7) + c(8, 3, 9)
```

```
[1] 14 12 20
```

Here is an important heads up, there's a catch when you name your list items. If the names of your list elements don't match the names of the arguments in the anonymous function, even though they're the same length (like `a`, `b`, `c` and `x`, `y`, `z`), you'll get an error:

```
pmap_dbl(
  .l = list(
    a = c(1, 3, 4),
    b = c(5, 6, 7),
    c = c(8, 3, 9)
  ),
  .f = \((x, y, z) x + y + z
)
```

```
Error in `pmap_dbl()`:
In index: 1.
Caused by error in `.\f()`:
! unused arguments (a = .l[[1]][[i]], b = .l[[2]][[i]], c = .l[[3]][[i]])
Run `rlang::last_error()` to see where the error occurred.
```

So when you have a named list, make sure to use the same names in the anonymous function to avoid any issues:

```

pmap_dbl(
  .l = list(
    a = c(1, 3, 4),
    b = c(5, 6, 7),
    c = c(8, 3, 9)
  ),
  .f = \((a, b, c) a + b + c
)

```

```
[1] 14 12 20
```

Now that we have seen the educational examples, let's return to our dice rolling scenario. Since a data frame is a type of list, its columns are list elements. Also, since `pmap` takes a list in `.l`, we can input our `dice_rolls` data frame into `.l`. Given that our data frame has three columns, the anonymous function must have three arguments as well. Just remember that a data frame is a named list, so the names of its elements must be used as the arguments in the anonymous function:

```

pmap_dbl(
  .l = dice_rolls,
  .f = \((size, min, max) {
    size + min + max
  }
)

```

```
[1] 17 18 107 108
```

Okay, let's not simply sum up the three values. Keep in mind that you can perform virtually any computation within the anonymous function. For our purpose, we want to simulate the dice rolls based on these three parameters. We can accomplish this using the `sample()` function as we did previously:

```

pmap(
  .l = dice_rolls,
  .f = \((size, min, max) {
    sample(min:max, size, TRUE)
  }
)

```

```

[[1]]
[1] 2 4 1 2 5 5 6 4 5 6

[[2]]
[1] 2 4 6 5 3 4 2 6 3 2

[[3]]
[1] 5 2 6 1 2 6 5 6 4 2 4 2 1 1 5 5 1 4 3 5 1 6 4 1 3 3 6 3 1 4 4 6 4 6 3 2 4
[38] 5 2 6 2 6 1 5 6 5 2 1 2 5 6 4 4 3 6 4 4 5 2 5 1 5 3 2 5 2 5 4 3 5 5 6 1 3
[75] 3 5 6 6 5 2 5 3 1 2 2 2 1 6 1 4 6 4 5 2 2 3 2 4 4 4

[[4]]
[1] 4 3 3 6 3 2 3 5 6 4 6 4 6 4 2 2 4 6 6 4 3 4 5 4 5 5 4 2 6 4 3 2 4 2 3 4 6
[38] 3 5 4 3 5 3 2 5 5 4 2 2 3 5 6 4 5 6 4 6 2 4 6 2 6 6 2 3 5 3 2 2 2 6 6 5 5
[75] 2 3 2 6 4 5 6 5 5 4 3 3 4 5 2 5 3 4 3 5 5 3 6 5 4 4

```

The result is a list of four simulations, each based on the three parameters. For now, this should provide enough to cover the basics of `pmap`.

20.4 The dot-dot-dot argument in `pmap`

`pmap` has another exciting feature called the dot-dot-dot argument, which is literally three dots The dots represent an *unspecified* number of additional arguments that are passed into the anonymous function.

You may have encountered the dot-dot-dot previously, such as in the `select()` function in the Tidyverse. The following is an excerpt from the official documentation showing its usage:

```
select(.data, ...)
```

You already know that once you specified the `.data` argument, you can add as many columns as you like:

```
select(.data = mpg, displ, manufacturer, model)
```

The same applies when using the dot-dot-dot in `pmap`. Let's say your data frame has a list of columns, where each row represents a word:

```
(words <- tribble(
  ~id, ~w, ~x, ~y, ~z,
  1, "t", "w", "o", NULL,
```

```

  2, "t", "r", "e", "e",
  3, "p", "o", "t", NULL
))

# A tibble: 3 x 5
  id w     x     y     z
  <dbl> <chr> <chr> <chr> <list>
1     1 t     w     o    <NULL>
2     2 t     r     e    <chr [1]>
3     3 p     o     t    <NULL>

```

You would like to create a character vector with these three words as vector elements. First, let's take a look at what the dot-dot-dot represents when we print it within the anonymous function:

```

pmap(
  .l = words,
  .f = \_(id, ...) {
    print(...)
  }
)

```

```

Warning: NAs introduced by coercionError in `pmap_chr()``:
In index: 1.
Caused by error in `print.default()``:
! invalid printing width
Backtrace:
1. purrr::pmap_chr(...)
2. purrr:::pmap_("character", .l, .f, ..., .progress = .progress)
6. global .f(...)
8. base::print.default(...)
```

You get an error. The reason is that the `print` function can only print a single object (`x`): `print(x, ...)`. But if we wrap the dot-dot-dot in a vector, it should work fine:

```

pmap(
  .l = words,
  .f = \_(id, ...) {
    c(...)
  }
)

```

```

)
[[1]]
      w   x   y
"t" "w" "o"
[[2]]
      w   x   y   z
"t" "r" "e" "e"
[[3]]
      w   x   y
"p" "o" "t"

```

The result is a list whose elements are character vectors. To get a single character vector, we need to reduce the vector elements to a single element. We can do this using the `reduce()` function:

```

pmap_chr(
  .l = words,
  .f = \ (id, ...) {
    reduce(c(...), paste0)
  }
)

```

```
[1] "two"  "tree" "pot"
```

See that I also changed the name of the function from `pmap` to `pmap_chr`.

The dot-dot-dot argument in `pmap` allows us to perform tasks similar to functions like `rowMeans`. For instance, if you wanted to find the average song ranking of songs in the `billboard` data set, you could do the following:

```

billboard |>
  select(track, where(is.numeric)) |>
  pmap_chr(
    .f = \ (track, ...) {
      paste(track, "mean ranking", round(mean(c(...), na.rm = TRUE), 2))
    }
  ) |>
  head(n = 10)

```

```
[1] "Baby Don't Cry (Keep... mean ranking 85.43"
[2] "The Hardest Part Of ... mean ranking 90"
[3] "Kryptonite mean ranking 26.47"
[4] "Loser mean ranking 67.1"
[5] "Wobble Wobble mean ranking 56.22"
[6] "Give Me Just One Nig... mean ranking 37.65"
[7] "Dancing Queen mean ranking 97"
[8] "I Don't Wanna mean ranking 52.05"
[9] "Try Again mean ranking 16.66"
[10] "Open My Heart mean ranking 67.75"
```

Here's another cool trick you can use with `pmap`. You can reference each element in the `...` argument by numbering them. For example, `..1` is the first element, `..2` is the second, and so on. With this, you can re-write the previous example to make use of the `...` argument:

```
pmap_dbl(
  .l = list(
    a = c(1, 3, 4),
    b = c(5, 6, 7),
    c = c(8, 3, 9)
  ),
  .f = \(...) ..1 + ..2 + ..3
)
```

```
[1] 14 12 20
```

This could be useful when your column names are lengthy and hard to type out.

20.5 `pmap` or `map2`?

The question on everyone's mind is whether to always use `pmap` instead of `map2`. Truth be told, using `pmap` in place of `map2` won't hurt. `pmap` can accomplish the same task and you'll only need to learn one function. If there's a reason for you to use both, go for it. I personally hadn't considered using one over the other until [Brenton Wiernik tweeted](#) about it. So, I think, there's no harm in always using `pmap`.

20.6 Use Case 1: Creating plots from a nested tibbles

For the rest of this chapter, let's explore a few more practical uses of `pmap`. One of them is definitely generating plots from nested data using `pmap`. This technique allows you to feed columns from your data frame into your anonymous function, creating a new column that holds a list of plots.

Imagine you want to visualize the distribution of TV watching hours among different marital statuses from the `gss_cat` dataset. To create histograms from this data frame, you'll need to follow these three steps:

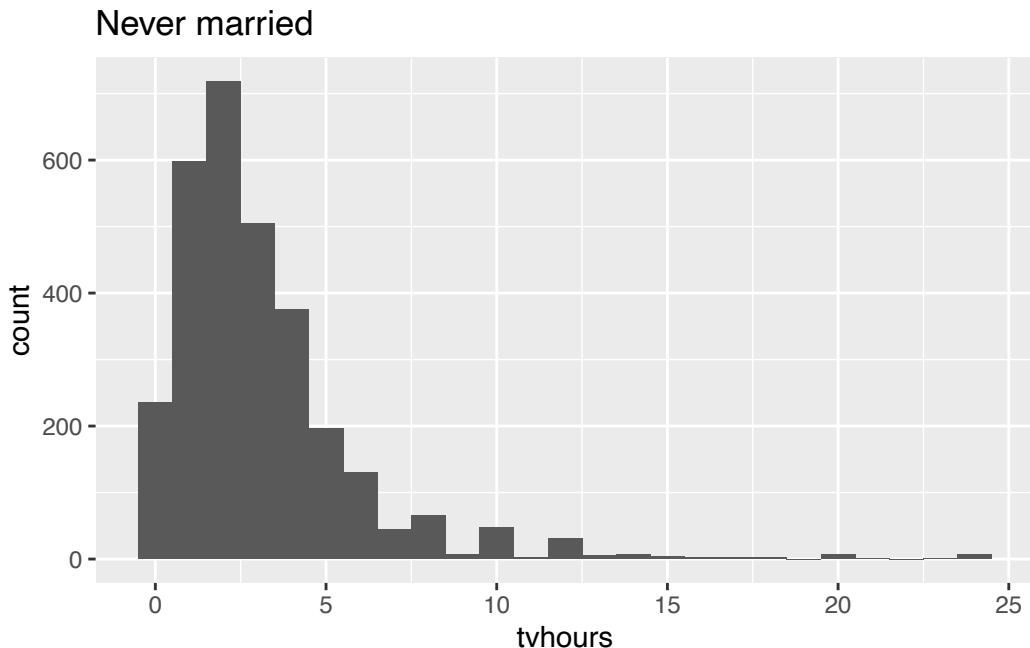
1. Nest the data frame by marital status
2. Create a new column to your data frame with `mutate` that stores the histograms you'll create
3. Use `pmap` and the information in your columns to return a `ggplot` object

Here's what the code looks like:

```
(histograms <- gss_cat |>
  nest(data = -marital) |>
  mutate(
    histograms = pmap(
      .l = list(marital, data),
      .f = \(marital, data) {
        ggplot(data, aes(x = tvhours)) +
          geom_histogram(binwidth = 1) +
          labs(
            title = marital
          )
      }
    )
  )))
# A tibble: 6 x 3
  marital      data      histograms
  <fct>     <list>     <list>
  1 Never married <tibble [5,416 x 8]> <gg>
  2 Divorced     <tibble [3,383 x 8]> <gg>
  3 Widowed      <tibble [1,807 x 8]> <gg>
  4 Married       <tibble [10,117 x 8]> <gg>
  5 Separated    <tibble [743 x 8]>   <gg>
  6 No answer    <tibble [17 x 8]>   <gg>
```

Storing the histograms in a plot is useless unless we can have a look at them. One option to do so is by accessing them via indexing:

```
histograms$histograms[[1]]
```



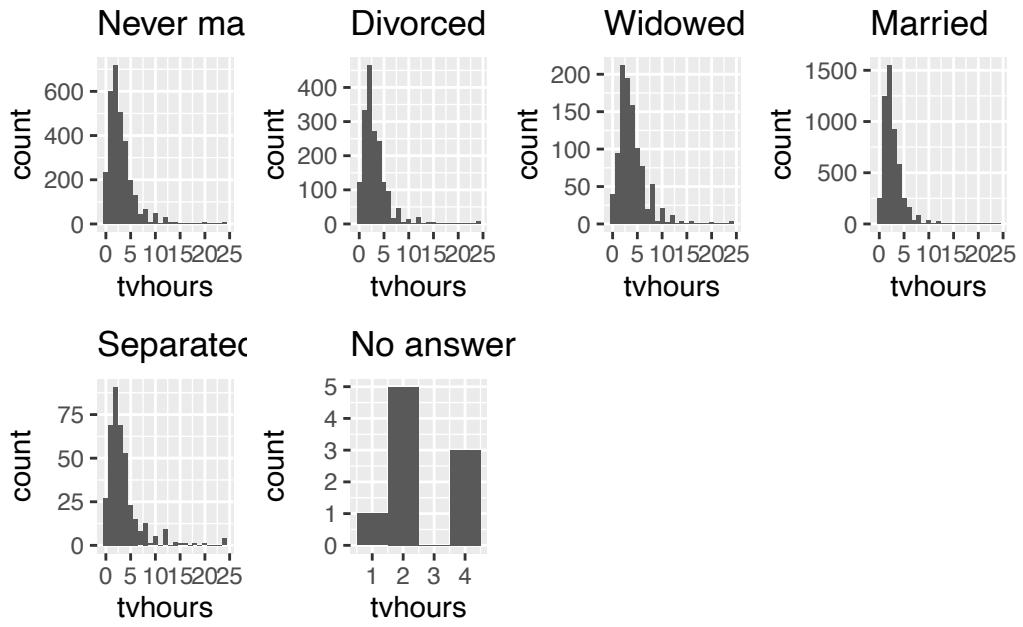
To view all the plots together, you can use the `reduce()` function in combination with the `patchwork` package. The `patchwork` package allows you to combine plots into a larger plot using the plus operator (+):

```
library(patchwork)  
  
histograms$histograms |>  
  reduce(`+`)
```



The drawback of this method is that it lacks customization options for the layout. To gain more control, we can use the `wrap_plots()` function from `patchwork`:

```
histograms$histograms |>
  wrap_plots(ncol = 4)
```



20.7 Use Case 2: Fitting models

In the previous example, we demonstrated that using `mutate()` and `pmap()` we can perform any operation on nested data frames. Instead of generating a plot within the anonymous function in `pmap()`, we can also fit models to the data. To start with, we will create a linear regression model for the diamonds data set. Our goal is to predict the price of a diamond based on its weight in carats and analyze how well the model fits for different diamond cut types.

```
(fitted_models <- diamonds |>
  nest(.by = cut) |>
  mutate(
    fitted_model = pmap(
      .l = list(cut, data),
      .f = \ (cut, data) {
        lm(price ~ carat, data = data)
      }
    )
  )))

```

```
# A tibble: 5 x 3
```

```

cut      data          fitted_model
<ord>   <list>        <list>
1 Ideal   <tibble [21,551 x 9]> <lm>
2 Premium <tibble [13,791 x 9]> <lm>
3 Good    <tibble [4,906 x 9]>  <lm>
4 Very Good <tibble [12,082 x 9]> <lm>
5 Fair    <tibble [1,610 x 9]>  <lm>

```

The linear regression models are stored in the `fitted_model` column:

```
fitted_models$fitted_model[[1]]
```

```

Call:
lm(formula = price ~ carat, data = data)

Coefficients:
(Intercept)      carat
-2300          8192

```

From these models, we can extract the R^2 to find out how well our model fits the data.

```

fitted_models |>
  mutate(
    r_squared = map_dbl(
      .x = fitted_model,
      .f = \((x) (
        summary(x)$r.squared
      )
    )
  )
)

# A tibble: 5 x 4
cut      data          fitted_model r_squared
<ord>   <list>        <list>        <dbl>
1 Ideal   <tibble [21,551 x 9]> <lm>       0.867
2 Premium <tibble [13,791 x 9]> <lm>       0.856
3 Good    <tibble [4,906 x 9]>  <lm>       0.851
4 Very Good <tibble [12,082 x 9]> <lm>       0.858
5 Fair    <tibble [1,610 x 9]>  <lm>       0.738

```

The key takeaway from this and the previous use case is that you can perform a wide range of computations on nested datasets by combining `mutate()` with `pmap()`. With this combination, you can perform tasks such as creating plots, fitting models, and extracting performance metrics. The possibilities are virtually limitless, and it is up to your imagination to determine how far you can take these examples.

20.8 Use Case 3: p-hacking

Please don't p-hack. It is an unacceptable practice among too many scientists still. For those who are unfamiliar with p-hacking, it is the process to fitting statistical models until one gets significant results. The purpose of this example is not to promote p-hacking, but rather to showcase the versatility and power of `pmap()`. I took inspiration for this use case from a blog post by Colin Fay, who wrote an article about this technique (<https://colinfay.me/purrr-statistics/>).

Suppose you got access to the `midwest` data set, which contains demographic information of midwest countries from the 2000 US census. The data set includes several numeric columns, and you'd like to perform a correlation test for every possible combination of these columns. To achieve this goal, the following four steps are necessary:

1. Select all numeric columns from the data frame
2. Create a data frame with two columns that represent all possible combinations of the numeric columns
3. Run a correlation test for each set of columns
4. Extract the p-values from the fitted models and filter those that are below 5%.

First, let's select only the numeric columns from the data frame:

```
midwest_numeric <- midwest |>
  select(where(is.numeric), -PID)

midwest_numeric |>
  glimpse()

Rows: 437
Columns: 24
$ area                  <dbl> 0.052, 0.014, 0.022, 0.017, 0.018, 0.050, 0.017, ~
$ poptotal               <int> 66090, 10626, 14991, 30806, 5836, 35688, 5322, 16~
$ popdensity              <dbl> 1270.9615, 759.0000, 681.4091, 1812.1176, 324.222~
$ popwhite                <int> 63917, 7054, 14477, 29344, 5264, 35157, 5298, 165~
$ popblack                <int> 1702, 3496, 429, 127, 547, 50, 1, 111, 16, 16559, ~
$ popamerindian           <int> 98, 19, 35, 46, 14, 65, 8, 30, 8, 331, 51, 26, 17~
```

```

$ popasian           <int> 249, 48, 16, 150, 5, 195, 15, 61, 23, 8033, 89, 3~
$ popother          <int> 124, 9, 34, 1139, 6, 221, 0, 84, 6, 1596, 20, 7, ~
$ percwhite         <dbl> 96.71206, 66.38434, 96.57128, 95.25417, 90.19877, ~
$ percblack         <dbl> 2.57527614, 32.90043290, 2.86171703, 0.41225735, ~
$ percamerindan    <dbl> 0.14828264, 0.17880670, 0.23347342, 0.14932156, 0~
$ percasiain        <dbl> 0.37675897, 0.45172219, 0.10673071, 0.48691813, 0~
$ percother         <dbl> 0.18762294, 0.08469791, 0.22680275, 3.69733169, 0~
$ popadults         <int> 43298, 6724, 9669, 19272, 3979, 23444, 3583, 1132~
$ perchsd           <dbl> 75.10740, 59.72635, 69.33499, 75.47219, 68.86152, ~
$ percollege        <dbl> 19.63139, 11.24331, 17.03382, 17.27895, 14.47600, ~
$ percprof          <dbl> 4.355859, 2.870315, 4.488572, 4.197800, 3.367680, ~
$ poppovertyknown   <int> 63628, 10529, 14235, 30337, 4815, 35107, 5241, 16~
$ percpovertyknown  <dbl> 96.27478, 99.08714, 94.95697, 98.47757, 82.50514, ~
$ percbelowpoverty  <dbl> 13.151443, 32.244278, 12.068844, 7.209019, 13.520~
$ percchildbelowpovert <dbl> 18.011717, 45.826514, 14.036061, 11.179536, 13.02~
$ percadultpoverty  <dbl> 11.009776, 27.385647, 10.852090, 5.536013, 11.143~
$ percelderlypoverty <dbl> 12.443812, 25.228976, 12.697410, 6.217047, 19.200~
$ inmetro            <int> 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0~

```

Next, let's find all combinations of columns. Fortunately, we can use the `tidy_comb_all()` function from the `tidystringdist` package to easily obtain this data set:

```

library(tidystringdist)

(combinations <- tidy_comb_all(names(midwest_numeric)))

# A tibble: 276 x 2
  V1     V2
  * <chr> <chr>
1 area   poptotal
2 area   popdensity
3 area   popwhite
4 area   popblack
5 area   popamerindian
6 area   popasian
7 area   popother
8 area   percwhite
9 area   percblack
10 area  percamerindan
# ... with 266 more rows

```

The result is a data frame with 276 unique column combinations, which is too many to analyze manually. Fortunately, we can use the `pmap()` function to run a correlation analysis on these columns. To do this, we pass the columns as arguments to an anonymous function within `pmap()` and perform a statistical test on the data. To ensure that the correlation test results are returned as a data frame, we wrap the results in a list, creating a list column. Finally, we combine the resulting data frames using `list_rbind()`:

```
(correlations <- pmap(
  .l = combinations,
  .f = \((V1, V2) {
    tibble(
      V1 = V1,
      V2 = V2,
      cor_test = list(cor.test(midwest_numeric[[V1]], midwest_numeric[[V2]])))
    )
  }
) |>
  list_rbind())

# A tibble: 276 x 3
  V1     V2       cor_test
  <chr> <chr>     <list>
1 area   poptotal <htest>
2 area   popdensity <htest>
3 area   popwhite <htest>
4 area   popblack <htest>
5 area   popamerindian <htest>
6 area   popasian <htest>
7 area   popother <htest>
8 area   percwhite <htest>
9 area   percblack <htest>
10 area  percamerindian <htest>
# ... with 266 more rows
```

Finally, to finish the analysis, we extract the p-values from the correlation tests and filter for those that are below a significance level of 5%:

```
correlations |>
  mutate(
    p_value = map_dbl(cor_test, \((x) x$p.value)
  ) |>
```

```

filter(p_value < 0.05)

# A tibble: 195 x 4
  V1      V2       cor_test   p_value
  <chr>  <chr>     <list>      <dbl>
1 area   popamerindian <htest>    7.75e- 5
2 area   percamerindan <htest>    2.51e- 2
3 area   percasiain   <htest>    1.29e- 2
4 area   perchsd      <htest>    2.10e- 3
5 area   percollege   <htest>    1.63e- 3
6 area   percpovertyknown <htest>  1.71e- 2
7 area   percadultpoverty <htest>  3.38e- 2
8 area   inmetro      <htest>    2.72e- 2
9 poptotal popdensity <htest>    4.71e-157
10 poptotal popwhite  <htest>    0
# ... with 185 more rows

```

Surprisingly, a large number of 195 out of 276 tests were significant.

Let's abstract what this use case has taught us. In the first two use cases, the nested data was placed into the anonymous function. This time, instead of nesting the data, we used the data frame of column combinations as a kind of database to select the right columns for the `cor.test` function.

This concludes this chapter on `map2()` and `pmap()`, and I hope it has given you a glimpse into the versatility and potential of these functions. Although there is much more to explore, I hope that these explanations have equipped you with enough knowledge to delve deeper into more complex analysis on your own.

Summary

- `map2` and `pmap` allow us to process multiple elements simultaneously.
- `map2` iterates over two elements at a time, while `pmap` can handle any number of elements
- When iterating over a list or vectors, it is important to make sure the anonymous function has the same number of arguments as elements in the list or vectors.
- The length of the objects being processed by `map2` and `pmap` should be the same, except when one of the objects has a length of one. In this case, the single-length object is recycled for each element of the other object.
- `map2` and `pmap` are commonly used in simulations, storing plots in columns, and storing statistical models in columns.

21 How to use the `walk` function family effectively

What will this tutorial cover?

In this tutorial, we will explore the `walk` family of functions, which were designed for side-effects - when a function modifies things outside of itself. Although they are similar to `map`, `map2`, and `pmap`, the key difference is that `walk` ignores the output of the function and is called solely for its side effects. We will start by gaining a better understanding of side effects and exploring a few illustrative examples of `walk`. We will then delve into three common use cases for these functions.

Who do I have to thank?

For this tutorial, I'd like to thank [Josep Espasa](#), [Philipp Baumann](#), [Samplaying](#), [Spencer Schien](#), [Brenton Wiernik](#), [Data Science with R](#), [Pedro Bittencourt](#), [Oskar Gauffin](#) for providing me with insights and ideas for this tutorial.

The functions `map`, `map2`, and `pmap` share a common characteristic: they do not produce side effects. If you have some programming experience, you may have already encountered this term. However, for those who are unfamiliar with it, let's take a brief moment to explain.

21.1 What are side-effects?

Side-effects occur when functions modify something outside themselves. In the context of purrr, this function is the anonymous function `.f`. Whenever `.f` modifies something outside of its own scope, a side-effect occurs.

In contrast, when the anonymous function is free of side-effects, it acts as a closed box. The function will not alter or affect anything outside of the box, and will only transform the input into an output. In other words, the function is a pure function, meaning it solely depends on its input and does not rely on or modify anything external to itself.

According to this definition, a `print` statement is a side-effects since it produces output outside of the function by printing to the console:

```
res <- map(  
  .x = 1:3,  
  .f = \((x) {  
    print(x)  
  }  
)
```

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

Side-effects occur when objects that are not included in the arguments of an anonymous function are modified. For instance, if a variable is changed within `map`, it constitutes a side-effect:

```
my_variable <- 2  
  
res2 <- map(  
  .x = 1:3,  
  .f = \((x) {  
    my_variable <<- x  
  }  
)  
  
my_variable
```

```
[1] 3
```

The `<<-` operator is often referred to as *super assignment* since it enables the modification of variables outside of a function. Without this operator and with only `<-`, it is not possible to change variables outside of a function's scope (more information about the `<<-` operator can be found here: <https://adv-r.hadley.nz/environments.html#env-basics>).

Another common example of a side-effect is writing a file to disk. In this case, the function modifies the state outside of its own scope by creating a new artifact that did not exist before the function was executed - the file.

```
ignored_output <- map(  
  .x = 1:3,  
  .f = \((x) {
```

```
    file.create(paste0("csv_files/", x, ".csv"))
  }
)
```

You may have observed that in each of the examples I have provided, I created a variable to store the result of `map`. However, my actual intention was not to use the output of `map`, but rather to generate side-effects. While this approach may work, it can lead to confusion and make the code harder to read, which is generally considered bad programming practice.

The developers of `purrr` created the `walk` functions to help programmers avoid writing code that is confusing and muddy. `walk` functions are specifically designed to enable the creation of side-effects, and they ignore the output of the anonymous function `.f`.

One of the advantages of using `walk` functions is that they operate in much the same way as `map`, `map2`, and `pmap`, so R-programmers can quickly adapt to them without needing to learn a lot of new concepts. However, the difference lies in the types of side-effects that can be generated and how they operate.

21.2 Educational examples of `walk`

In the upcoming examples, I will guide you through a few educational use cases. I aim to keep this section brief before delving into more compelling applications of `walk`.

The `walk` family of functions operate in a similar way to the functions we have previously discussed:

- `walk` is `map` for side-effects
- `walk2` is `map2` for side effects
- `pwalk` is `pmap` for side effects

In addition, there is a fourth function named `iwalk`, but we will not cover it in this chapter. Nonetheless, everything you have learned about `map`, `map2`, and `pmap` is applicable to `walk`, `walk2`, and `pwalk`.

`walk` works on a single list of atomic vectors. For instance, consider this example in which we print the result of each element:

```
walk(
  .x = 1:3,
  .f = \(x) {
    print(x)
  }
)
```

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

The most intriguing aspect is the part that remains unseen. Although we only observe the print statements, `walk` did not return any output. However, this is only partially correct. If you store the output of `walk` in a variable, the variable will contain the original input `.x`:

```
walk_output <- walk(  
  .x = 1:3,  
  .f = \ (x) {  
    print(x)  
  }  
)
```

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

```
walk_output
```

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

The same is true for `walk2` and `pwalk`. Nevertheless, since these functions are employed for side-effects, we will rarely store their output in a variable.

`walk2` is comparable to `map2` as it takes two objects as inputs. Once again, let's utilize a print statement for side-effects:

```
walk2(  
  .x = 1:3,  
  .y = 4:6,  
  .f = \ (x, y) {  
    print(x + y)  
  }  
)
```

```
[1] 5  
[1] 7  
[1] 9
```

`pwalk` is similar to `pmap` and requires a list as input for `.l`:

```
pwalk(  
  .l = list(  
    1:3, 4:6, 7:9  
)  
  ,  
  .f = \ (x, y, z) {  
    print(x + y + z)  
  }  
)
```

```
[1] 12  
[1] 15  
[1] 18
```

After having reviewed an example of each, let's explore practical applications of these functions.

21.3 Saving plots to disk with `pwalk`

Previously, we covered the method for storing plots in a data frame column and how to combine multiple plots into a single one using `patchwork`. However, we have yet to discuss the process of saving these plots to disk, which is considered a side-effect. This is where `pwalk` becomes particularly relevant, as it offers a convenient way to achieve this goal.

Here's the example that we created in the preceding chapter:

```
(histograms <- gss_cat |>  
  nest(data = -marital) |>  
  mutate(  
    histogram = pmap(  
      .l = list(marital, data),  
      .f = \ (marital, data) {  
        ggplot(data, aes(x = tvhours)) +  
          geom_histogram(binwidth = 1) +  
          labs(  
            title = marital  
          )  
      }  
    )  
)
```

```
# A tibble: 6 x 3
  marital      data      histogram
  <fct>     <list>    <list>
1 Never married <tibble [5,416 x 8]> <gg>
2 Divorced     <tibble [3,383 x 8]> <gg>
3 Widowed      <tibble [1,807 x 8]> <gg>
4 Married       <tibble [10,117 x 8]> <gg>
5 Separated    <tibble [743 x 8]>   <gg>
6 No answer    <tibble [17 x 8]>   <gg>
```

The given data frame contains histograms depicting the number of hours that individuals watch TV, categorized by their marital status. Each histogram is labeled with the corresponding marital status. Our next step is to save these plots to disk with descriptive names.

With `pwalk` we need pass the columns of the data frame (which correspond to the named list elements) as arguments to the anonymous function `.f`. However, it's worth mentioning that providing only the necessary two columns of the data frame as arguments to `.f` will result in an error, as illustrated below.

```
pwalk(
  .l = histograms,
  .f = \(marital, histogram) {
    NULL
  }
)
```

```
Error in `pmap()``:
  In index: 1.
Caused by error in `.`f()``:
! unused argument (data = .l[[2]][[i]])
Run `rlang::last_error()` to see where the error occurred.
```

The error message indicates that an argument is missing, specifically `data = .l[[2]][[i]]`. The number 2 in this context refers to the second column of the `histograms` data frame, which is `data`:

```
histograms |> colnames()
```

```
[1] "marital"    "data"        "histogram"
```

To address this problem, we have two possible solutions. One option is to include all columns of the data frame in the anonymous function. Alternatively, we can use the dot-dot-dot argument to allow for an unspecified number of additional arguments.

In this code snippet, I have explicitly included all columns in the anonymous function:

```
pwalk(  
  .l = histograms,  
  .f = \ (marital, data, histogram) {  
    NULL  
  }  
)
```

If your data frame had a large number of columns, specifying all of them individually would be unwieldy. In such cases, I would recommend using the dot-dot-dot argument as an alternative approach:

```
pwalk(  
  .l = histograms,  
  .f = \ (marital, histogram, ...) {  
    NULL  
  }  
)
```

With the arguments for the anonymous function now defined, we can delve into the function's internal workings. The primary side-effect of this function is writing plots to disk. The plot data is derived from the supplied arguments, with the `histograms` argument providing the ggplot objects and the `marital` argument providing the marital status used for the filename.

Before you run this code, there are a couple of important things to consider. First, note that the `fs` library is loaded and the `dir_create` function is used to create a directory named `histograms` before calling `pwalk`. If a directory already exists, `dir_create` does nothing, whereas if it does not exist, the function creates it. Additionally, the `.progress` argument has been specified in `pwalk` to provide a visual indication of progress of the iterations.

```
library(fs)  
  
dir_create("histograms")  
  
pwalk(  
  .l = histograms,  
  .f = \ (marital, histogram, ...) {
```

```

ggsave(
  filename = paste0("histograms/", marital, "_histogram.png"),
  plot = histogram,
  dpi = 300
)
},
.progress = TRUE
)

```

Once you've run this code, you will find six PNG files in a directory named `histograms`. Now suppose you want to store a larger number of histograms in a different subfolder for each year:

```
css_cat$year |> unique()
```

```
[1] 2000 2002 2004 2006 2008 2010 2012 2014
```

To save these histograms to disk, we need to modify the creation of the data frame by nesting the `data` except for the `year` and `marital` status:

```

(histograms_per_year <- gss_cat |>
  nest(data = -c(marital, year)) |>
  mutate(
    histogram = pmap(
      .l = list(marital, data),
      .f = \ (marital, data) {
        ggplot(data, aes(x = tvhours)) +
          geom_histogram(binwidth = 1) +
          labs(
            title = marital
          )
      }
    )
  )))

```

```
# A tibble: 45 x 4
  year marital      data      histogram
  <int> <fct>       <list>     <list>
1 2000 Never married <tibble [712 x 7]> <gg>
2 2000 Divorced     <tibble [441 x 7]> <gg>
3 2000 Widowed      <tibble [273 x 7]> <gg>
```

```

4 2000 Married      <tibble [1,278 x 7]> <gg>
5 2000 Separated    <tibble [112 x 7]>   <gg>
6 2000 No answer    <tibble [1 x 7]>   <gg>
7 2002 Divorced    <tibble [445 x 7]>   <gg>
8 2002 Married     <tibble [1,269 x 7]> <gg>
9 2002 Separated    <tibble [96 x 7]>   <gg>
10 2002 Never married <tibble [708 x 7]> <gg>
# ... with 35 more rows

```

With the information stored in this data frame, we can save these plots into their respective subfolders. It is important to note the initial use of `walk` to create the subfolders before proceeding with `pwalk` to store the plots on disk.

With this information, we can store these plots in the subfolders. Pay attention to the first call of `walk`. I used `walk` to create the subfolders first. Only then do I call `pwalk` to save the plots to disk.

```

# Create the subfolders
walk(
  .x = histograms_per_year$year |> unique(),
  .f = \(year) {
    dir_create(paste0("histograms_per_year/", year))
  }
)

# Save the plots to disk
pwalk(
  .l = histograms_per_year,
  .f = \(year, marital, histogram, ...) {
    ggsave(
      filename = paste0("histograms_per_year/", year,
                        "/", marital, "_",
                        year, "_", "_histogram.png"),
      plot = histogram,
      dpi = 300
    )
  },
  .progress = TRUE
)

```

Let's expand our perspective beyond these specific examples. When saving plots from nested data frames, it is essential to identify which columns should *not* be nested. The data within these columns will either be used to create subfolders or name files. You can create the

directories for the plots either outside the `walk` function, or inside it. However, if you create the directories within the plot-generating function, you may end up running the function more frequently than necessary. Other uses other than saving plots to disk are storing the data as CSV files and writing the data to a database via an API.

21.4 Creating folders based on information within a data frame

Suppose you are the coach of a running club consisting of four members who participate in three different types of races: half-marathons, marathons, and 10km races. As it is now the start of 2023, you want to organize their certificates and travel documents into folders spanning from 2023 to 2026. Given the four runners, three race types, and three-year time frame, a total of 48 folders must be created.

Before you create these folders with `pwalk`, you need to create a data frame that stores the information. In one of our chapters we came across the function `crossing()`. Let's use it to create this data frame:

```
(folder_information <- crossing(
  runner = c("Mike", "Natasha", "Leonie", "Sabrina"),
  competition = c("half-marathon", "marathon", "10km"),
  year = 2023:2026
))

# A tibble: 48 x 3
  runner competition    year
  <chr>   <chr>        <int>
1 Leonie  10km          2023
2 Leonie  10km          2024
3 Leonie  10km          2025
4 Leonie  10km          2026
5 Leonie  half-marathon 2023
6 Leonie  half-marathon 2024
7 Leonie  half-marathon 2025
8 Leonie  half-marathon 2026
9 Leonie  marathon       2023
10 Leonie marathon       2024
# ... with 38 more rows
```

Next, we use this data frame for the `.l` argument in `pwalk()` and create each folder inside the anonymous function with `dir_create()` from the `ls` package. It doesn't matter if a folder

already exists. If it does, `dir_create()` will simply ignore that folder and the next iteration will continue.

```
pwalk(  
  .l = folder_information,  
  .f = \ (runner, competition, year) {  
    dir_create(  
      path = paste0("runners/", runner, "/", year, "/", competition)  
    )  
  }  
)
```

This approach can also be used to create missing folders. Consider, for instance, the scenario where you organize your tax returns into folders, but forgot to create a few of them. Alternatively, suppose you maintain your diary as Markdown files and wish to fill in any subfolders you failed to create in previous years. The potential applications for this method are virtually limitless.

21.5 Organize images into folders based on metadata information

For the next use case I have to thank [Eli Pousson](#). He pointed me to the package `exiftoolr` which allows to extract meta information from images. The package is a wrapper around [ExifToll developed by Phil Harvey](#). If you follow along, make sure you have both the ExifToll software and the `exiftoolr` package installed.

The use case is as follows. You have a list of images in a folder. You want to sort these images into subfolders according to when they were created. The subfolders should be sorted by year and month:

```
* 2004  
  - Jan  
  - Feb  
* 2005  
  - Aug  
  - Oct
```

The function `exif_read` from the package `exiftoolr` extracts dozens of values from images. Here is an overview of the output from one of my images:

```
Classes 'exiftoolr' and 'data.frame': 1 obs. of 97 variables:  
 $ SourceFile : chr "images/CIMG5775.JPG"
```

```

$ ExifToolVersion      : num 12.6
$ FileName             : chr "CIMG5775.JPG"
$ Directory            : chr "images"
$ FileSize              : int 1890606
$ FileModifyDate        : chr "2023:01:07 15:42:58+01:00"
$ FileAccessDate        : chr "2023:02:12 08:08:59+01:00"
$ FileInodeChangeDate   : chr "2023:02:12 07:41:25+01:00"
$ FilePermissions        : int 100700
$ FileType              : chr "JPEG"
$ FileTypeExtension      : chr "JPG"
$ MIMEType              : chr "image/jpeg"
$ ExifByteOrder          : chr "II"
$ Make                  : chr "CASIO COMPUTER CO.,LTD."
$ Model                 : chr "EX-Z500"
$ XResolution           : int 72
$ YResolution           : int 72
$ ResolutionUnit         : int 2
$ Software               : num 1
$ ModifyDate             : chr "2007:08:04 00:47:29"
$ YCbCrPositioning       : int 1
$ ExposureTime           : num 0.0167
$ FNumber                : num 2.7
$ ExposureProgram         : int 2
$ ExifVersion             : chr "0221"
$ DateTimeOriginal        : chr "2007:08:04 00:47:29"

$ CreateDate             : chr "2007:08:04 00:47:29"

$ ComponentsConfiguration: chr "1 2 3 0"
$ CompressedBitsPerPixel : num 3.33
$ ExposureCompensation    : int 0
$ MaxApertureValue         : num 2.64
$ MeteringMode             : int 5
$ LightSource              : int 0
$ Flash                   : int 9

```

For this use case we use the value `CreateData`. Of course, you may also wish to use other metadata values to sort and save your files in a different way, depending on your specific needs.

To get started, we'll be using the `magick` package and its `image_read()` and `image_write()` functions, which enable you to select an image from a given path and save it to a different location.

The first step is to save the file paths for your images. For example, let's say you have saved your images in a folder named `images/`. Note that this folder may include files with different extensions (e.g. `.jpg` in lower case and `.JPG` in upper case). One approach for storing these file paths in a character vector is to use the `dir_ls` function, along with the `glob` argument. Alternatively, you may choose to use a regular expression to extract all the file paths at once. Examples of both approaches are provided for your reference.

```
# library(fs) # Make sure you have loaded this package

image_paths_jpg_lowercase <- dir_ls(path = "images/",
                                     glob = "*.jpg", type = "file")
image_paths_jpg_uppercase <- dir_ls(path = "images/",
                                      glob = "*.JPG", type = "file")

image_paths <- rbind(image_paths_jpg_lowercase, image_paths_jpg_uppercase)

# Alternative
# image_paths <- dir_ls(path = "images/",
# .               regexp = "*\\.[Jj][Pp][Ee]?[Gg]", type = "file")
```

Here is an overview of the first ten paths in my vector of image paths:

```
image_paths[1:10]
```

```
[1] "images/1005455_10201801428460577_1650175180_n.jpg"
[2] "images/CIMG2788.JPG"
[3] "images/1017220_572782476093473_1477751556_n.jpg"
[4] "images/CIMG5033.JPG"
[5] "images/2012-06-07_16.27.12.jpg"
[6] "images/CIMG5732.JPG"
[7] "images/20120518_004540 - Kopie.jpg"
[8] "images/CIMG5775.JPG"
[9] "images/20131109_022223.jpg"
[10] "images/CIMG5878.JPG"
```

Our next step is to extract the meta information from each image and locate the date the image was created. I'll share the code with you, but before running the function, let's take a moment to review it together.

```
library(exiftoolr)
library(fs)
```

```

library(magick)
library(lubridate)

walk(
  .x = image_paths,
  .f = \(path) {
    metadata <- exif_read(path) # Reads the meta information from the image
    filename <- metadata$FileName # Stores the filename, not the path
    original_datetime <- metadata$CreateDate # Reads the date when the image was created

    # Only read the image and store it in another folder
    # if an origin date exists
    if (!is.null(original_datetime)) {
      # TODO: Read the image and store it in the right folder

      # If there is not CreateDate information
      # print a statement telling the user the information doesn't exist.
    } else {
      # TODO: Write print message
    }

  },
  .progress = TRUE # Print the progress
)

```

I'd like to draw your attention to the following code snippets:

- To begin, at the start of the anonymous function, I am gathering all the necessary data required to store each image in its corresponding folder. This includes the exif object, file name, and the date the image was created.
- It's worth noting that not all images contain the necessary metadata, specifically the creation date. In this case, we check to see if the `original_datetime` data exists (if `(!is.null(original_datetime))`). If it does, we read the image and save it to the appropriate folder. If it doesn't exist, we use a `print` statement to notify us that further action is required. Alternatively, you could opt to store these images in a separate, unstructured folder.
- Additionally, you may have noticed that I have set the `.progress` argument to `TRUE`. Since processing images can be a time-consuming task, especially when dealing with a large number of files, it can be helpful to keep track of the function's progress.

To dive into further detail, it's important to consider how we process images that contain a creation date. Typically, the date is stored as a datetime, which includes the date and time the image was taken. However, for our purposes, we only need to access the month in which

the image was created. Once we have this information, we can construct the new folder path and create the folder using the `dir_create` function. Finally, we read the image from disk using `image_read` and write it to the appropriate path using `image_write`.

```
library(exiftoolr)
library(fs)
library(magick)
```

Linking to ImageMagick 6.9.12.3

Enabled features: cairo, fontconfig, freetype, heic, lcms, pango, raw, rsvg, webp
Disabled features: fftw, ghostscript, x11

```
library(lubridate)
```

Attaching package: 'lubridate'

The following objects are masked from 'package:base':

```
date, intersect, setdiff, union
```

```
walk(
  .x = image_paths,
  .f = \(path) {
    metadata <- exif_read(path)
    filename <- metadata$FileName
    original_datetime <- metadata$CreateDate

    if (!is.null(original_datetime)) {
      # ****
      # This is new
      date <- original_datetime |> ymd_hms()
      year <- date |> year()
      month <- date |> month(label = TRUE)

      folder_path_export <- paste0("images_processed/", year, "/", month)

      dir_create(folder_path_export)
```

```

current_image <- image_read(path)

image_write(
  image = current_image,
  path = paste0(folder_path_export, "/", filename)
)
# ****
} else {
  print(paste("No date found for:", filename))
}

},
.progress = TRUE
)

```

Using ExifTool version 12.56

```

[1] "No date found for: 1005455_10201801428460577_1650175180_n.jpg"
[1] "No date found for: 1017220_572782476093473_1477751556_n.jpg"

=====>----- 22% | ETA: 11s

[1] "No date found for: 239119_0_gross_110.jpg"
[1] "No date found for: 5334_100093037919_591727919_1929677_673919_n.jpg"
[1] "No date found for: 936443_10151694619437378_111021535_n.jpg"
[1] "No date found for: IMG 039.jpg"
[1] "No date found for: IMG-20140620-WA0007.jpg"

=====>----- 48% | ETA: 7s

[1] "No date found for: 1005455_10201801428460577_1650175180_n.jpg"
[1] "No date found for: 1017220_572782476093473_1477751556_n.jpg"

=====>----- 66% | ETA: 5s

[1] "No date found for: 239119_0_gross_110.jpg"
[1] "No date found for: 5334_100093037919_591727919_1929677_673919_n.jpg"

```

```
=====>---- 86% | ETA: 2s
```

```
[1] "No date found for: 936443_10151694619437378_111021535_n.jpg"  
[1] "No date found for: IMG 039.jpg"  
[1] "No date found for: IMG-20140620-WA0007.jpg"
```

```
=====>- 98% | ETA: 0s
```

It's worth noting that the approach used to extract metadata from image files can be applied to other file types as well, such as PDFs, .docx files, and more. If you're planning to do an annual hard drive cleaning, you might consider using the walk functions of the Tidyverse to streamline the process and organize your files more efficiently.

21.6 Intermediate tests

Up until now, we haven't chained any other functions after `walk`. However, there's no reason why we shouldn't. Since `walk` returns the input invisibly, we can use it as input in a pipe of functions, without affecting our computations. This gives us the ability to run tests in the middle of chains.

As an example, consider this - admittedly nonsensical - case of checking the number of rows in a list of data frames. In this example, we test whether the data frame has more than 13,000 rows. If it does, we `print` a statement. If it doesn't, we don't print anything. After this check, we continue and fit linear models to the data.

```
models <- split(diamonds, diamonds$cut) |>  
  walk(  
    .f = \ (x) {  
      ifelse(  
        test = nrow(x) > 13000,  
        yes = print(paste(x$cut[1] |> as.character(),  
                           "has more than 13000 rows")),  
        no = NA)  
    }  
  ) |>  
  map(  
    .f = \ (x) {  
      summary(lm(price ~ carat, data = x))  
    }  
  )
```

```
[1] "Premium has more than 13000 rows"  
[1] "Ideal has more than 13000 rows"
```

Removing `walk()` from the example would not affect the results. There are also other uses of `walk()` in the middle of chains:

- When you need to perform statistical tests on a list of models
- When you want to print out outliers in your data
- When you want to save intermediate results of your computation to disk

Summary

- The `walk` family of functions was designed for side-effects, which occur when a function modifies things outside of itself. Side-effects can include `print` statements, writing data to disk, or changing variables outside the function.
- The walk family of functions includes `walk`, `walk2`, `pwalk`. These functions have a structure similar to `map`, `map2`, and `pmap`, with the key difference being that `walk` ignores the output of the anonymous `.f()` function and calls it solely for its side-effects.
- `walk()` is commonly used for creating folders, storing plots to disk, and running intermediate tests in a chain of operations.

Part X

Improve your code's performance

22 How to utilize your computer's parallel processing capabilities using `future` and `furrr`

What will this tutorial cover?

This tutorial introduces the `future` and `furrr` packages, which utilize modern computer's parallel processing capabilities to accelerate computations in R. `furrr` is constructed on top of `future` and employs `purrr`-like functions. The tutorial delves into the functions' mechanics of both packages and explores a use case of reading multiple CSV files. Additionally, we will conduct two simulations to demonstrate the functions' speed enhancement.

💡 Who do I have to thank?

I have to thank [Henrik Bengtsson](#) and [Davis Vaughan](#), the core developers of `future` and `furrr`, respectively. In addition, I found the tutorial video by [UNMCARC on YouTube](#) to be extremely useful in creating this tutorial. I also gained valuable insights from [FXQuantTrader's](#) response on Stackoverflow regarding how to split a vector into equally sized bins.

Speed is generally not a significant concern for the majority of R users. The R language and the Tidyverse were created with a focus on “getting things done”, or as Hadley Wickham expressed on Twitter, “Tidyverse is a set of packages that try to help you do data science.” (<https://twitter.com/hadleywickham/status/1143546400556425218>).

The user base of R reflects this philosophy as they are primarily people without formal programming education who learn the language on the job. The data sets these users typically handle are relatively small, often no larger than 2 GB in size (<https://r4ds.had.co.nz/introduction.html>). As data sets increase in size, code execution time becomes more significant. At this juncture, R users typically need to consider optimizing their code and utilize their machines' computational capabilities to the fullest extend possible.

Fortunately, the developers of `future` and `furrr` have designed packages that harness the parallel processing potential of modern computers. `future` simplifies the process of leveraring

these capabilities, while **furrr** builds on **future** to utilize the parallel processing with **purrr**-like functions. One promise of **future**, as highlighted by its developer Henrik Bengtsson in a [YouTube video](#), is that you write it once and it runs anywhere.

Before I go into the nitty-gritty, let me demonstrate an example to explain the concept more clearly. I will present two code snippets. The first one does not utilize parallel processing capabilities, while the other one utilizes all the cores of my M1 chip that is integrated into my Macbook Pro. You do not need to comprehend the code in detail at this moment, as we are primarily concerned with the output.

The subsequent code snippet calls a slow function that takes 30 seconds to complete when executed sequentially:

```
library(tictoc)
library(future)
library(furrr)

plan(sequential)

tic()
future_walk(.x = c(5, 5, 5, 5, 5, 5), .f = \(x) Sys.sleep(x))
toc()
```

30.043 sec elapsed

Now, let's compare the same computations when performed in parallel:

```
plan(multisession, workers = 6)

tic()
future_walk(.x = c(5, 5, 5, 5, 5, 5), .f = \(x) Sys.sleep(x))
toc()
```

5.265 sec elapsed

Five seconds. We managed to boost the computation speed by a factor of six.

As a general guideline for this tutorial, if you feel that some of your computations are taking too long, it may be beneficial to make use of **future** and **furrr**. However, if you are satisfied with the current speed of your computations, you need not worry about this tutorial. You can instead invest your time in learning other topics and revisit this tutorial when there is a need to speed up your computations.

22.1 An analogy on parallel processing

I am not an expert in computer hardware or parallel processing, but I'll do my best to provide you with the most comprehensive overview I can. The goal is to help you gain a better understanding of the key concepts that are essential for future and furrr.

The best explanation I found on parallel processing came from the book Introduction to High-Performance Computing (<http://www.hpc-carpentry.org/hpc-chapel/11-parallel-intro/index.html>) by The Carpentries. They use the example of painting a wall to illustrate the concept. Let's say you have four walls to paint, which is the problem at hand. Now, you can break this down into four individual tasks, painting each wall. In theory, these tasks are independent, meaning you can either paint the walls one after another, or in parallel. We refer to these four tasks as "concurrent," which just describes the property of a task, and doesn't necessarily mean that they are executed in parallel or sequentially. It all depends on the resources available. For instance, if you are painting the walls alone, the tasks are concurrent, but you need to work sequentially. However, with two painters, you can execute the concurrent tasks in parallel.

So, the painters are what we call **workers** in parallel processing. Generally, you can assign the number of workers equivalent to the number of cores present in your machine. For instance, my personal laptop is a MacBook Pro with an M1 chip that has eight cores. This means that I can create at least eight workers to operate in parallel. The term worker will become important in just a moment when we begin to use the `plan()` function of `future`.

22.2 The future package

The cross-platform `future` package is a high-level API that allows non-technical people to take advantage of parallel processing capabilities when working in R. The package was developed by [Henrik Bengtsson](#), an assistant professor of epidemiology and biostatistics at the University of California. The first version of `future` was released on GitHub in 2015. At the time of writing this tutorial, `future` has reached version 1.31.0.

`future` works in three steps:

- Step 1: You need to decide how you want to parallelize your code. You have four options: running the code sequentially, running in multiple R sessions, running on multiple cores, or running on a cluster of multiple machines. You can use the `plan()` function to make this decision. Note that when you work in R-Studio, using multiple cores is not recommended and doesn't work on Windows, so most of the time, you will work in multiple R-Sessions.
- Step 2: Once you've decided how to parallelize your code, you need to determine which part of your code should be run in parallel. You can use the `future()` function to make this decision.

- Step 3: You need to retrieve the value of your computations run in future. You can use the `value()` function to get these values.

Let's use my laptop as an example again. I'm currently using an Apple MacBook Pro with an M1 chip that has 8 cores. If I didn't know the number of cores, I could use the `availableCores()` function to check it:

```
library(future)
availableCores()

system
8
```

To decide whether you want to run your code sequentially or in parallel, you use the `plan()` function. By default, when you load the `future` package, the code runs sequentially. Here's an example of setting up a sequential process. The following code computes the square of three numbers using a slow function called `slow_function_square_number`:

```
plan(sequential)

# This function takes more than 2 seconds to run
slow_function_square_number <- function(i) {
  Sys.sleep(2)
  i^2
}

# start timer
tic()

num_one <- slow_function_square_number(2)
num_two <- slow_function_square_number(4)
num_three <- slow_function_square_number(9)

num_one
```

```
[1] 4
```

```
num_two
```

```
[1] 16
```

```
num_three
```

```
[1] 81
```

```
# end timer  
toc()
```

```
6.033 sec elapsed
```

Take a look at `plan(sequential)` and the function calls to `slow_function_square_number()`. This tells the computer to not use the parallel processing capabilities and instead run the computations one after another. Keep in mind that each call of `slow_function_square_number()` takes at least 2 seconds to run because the function was intentionally delayed by two seconds.

To run the computations in parallel, we need to make the following changes:

- First, let's change the `plan` and use `plan(multisession)`. This plan will create multiple R-Sessions that run in parallel. By default, this function uses as many workers as are available.
- Second, we'll wrap the three function calls `slow_function_square_number()` inside `future()`. By doing so, we're telling `future` that these computations should run in parallel based on the number of sessions we specified in `plan(multisession)`.
- Third, we'll retrieve the values of our `future()` computations using `value()`.
- Fourth, we'll call `plan(sequential)` to close the multisession workers. This is considered good practice in the `future` package.

```
plan(multisession)  
  
# This function takes more than 2 seconds to run  
slow_function_square_number <- function(i) {  
  Sys.sleep(2)  
  i^2  
}  
  
# start timer  
tic()  
  
num_one <- future({slow_function_square_number(2)})  
num_two <- future({slow_function_square_number(4)})  
num_three <- future({slow_function_square_number(9)})
```

```
value(num_one)
```

```
[1] 4
```

```
value(num_two)
```

```
[1] 16
```

```
value(num_three)
```

```
[1] 81
```

```
# end timer  
toc()
```

```
2.124 sec elapsed
```

```
plan(sequential)
```

We have achieved a threefold increase in speed. Now, this is the explicit way of using `future` where the functions `future()` and `value()` are explicitly named. Alternatively, one can use `future()` implicitly through the use of the `%<-%` operator. Instead of the explicitly declaring `f <- future({ exp})`, one writes `f %<-% {exp}`:

```
plan(multisession)
```

```
# This function takes more than 2 seconds to run  
slow_function_square_number <- function(i) {  
  Sys.sleep(2)  
  i^2  
}  
  
# start timer  
tic()  
  
num_one %<-% {slow_function_square_number(2)}
```

```

num_two %<-% {slow_function_square_number(4)}
num_three %<-% {slow_function_square_number(9)}

# end timer
toc()

```

0.097 sec elapsed

```
plan(sequential)
```

When is this technique useful? One scenario is when there is a need to read multiple files into memory. For example, if you have numerous CSV files stored on your disk and would like to read them in parallel. In the code below, I generate 5000 CSV files and store them in a folder named `diamonds_data`:

```

library(fs)

dir_create("diamonds_data")
diamonds_samples <- map(1:1000, ~ slice_sample(diamonds, n = 4000))
iwalk(diamonds_samples, ~ write_csv(., paste0("diamonds_data/", .y, ".csv")))

```

Then, using the `dir_ls` function from the `fs` package, I retrieve the path names of these files:

```

csv_files <- dir_ls(path = "diamonds_data/", glob = "*.csv", type = "file")
csv_files |> head()

```

```

diamonds_data/1.csv    diamonds_data/10.csv   diamonds_data/100.csv
diamonds_data/1000.csv  diamonds_data/101.csv   diamonds_data/102.csv

```

First, I execute the non-parallelized slow version:

```

plan(sequential)

tic()
diamonds_data_complete_slow <- purrr::map(.x = csv_files,
                                             .f = \((file_path) read_csv(file_path) |>
                                              mutate(name = file_path)) |>
list_rbind()
toc()

```

```
29.519 sec elapsed
```

To expedite this computation, a few modifications are necessary. Firstly, we need to divide the dataset into bins. This is done using the function call `split(csv_files, rep_len(1:3, length(csv_files)))`, which is a helpful tip I discovered on Stackoverflow. I would like to give credit to [FXQuantTrader](#) for sharing it. Following this, I execute `plan(multisession)` and use three implicit `future` calls, one for each bin, and then execute `map()`. Finally, the three lists are combined into a dataframe using `list_rbind`:

```
# Split CSV paths into a list of length
# three. Each list element holds roughly the same number of CSV paths
# Tip: https://stackoverflow.com/questions/3318333/split-a-vector-into-chunks
csv_files_splits <- split(csv_files, rep_len(1:3, length(csv_files)))

plan(multisession)

tic()

diamonds_data_one %<-% purrr::map(.x = csv_files_splits[[1]],
                                     .f = \`(file_path) read_csv(file_path)
                                     |> mutate(name = file_path))

diamonds_data_two %<-% purrr::map(.x = csv_files_splits[[2]],
                                     .f = \`(file_path) read_csv(file_path)
                                     |> mutate(name = file_path))

diamonds_data_three %<-% purrr::map(.x = csv_files_splits[[3]],
                                       .f = \`(file_path) read_csv(file_path)
                                       |> mutate(name = file_path))

# Combine data frames into one data frame
diamonds_data_complete_fast <- bind_rows(
  diamonds_data_one |> list_rbind(),
  diamonds_data_two |> list_rbind(),
  diamonds_data_three |> list_rbind()
)

toc()
```

```
14.371 sec elapsed
```

```
plan(sequential)

# Remove data sets
rm(diamonds_data_one)
rm(diamonds_data_two)
rm(diamonds_data_three)
```

Executing this code takes only 15 seconds. It is worth noting that I have repeated my code thrice, which is not ideal coding practice. Rather than this approach, I could have implemented a loop. Fortunately, there is an alternative solution available in the form of the **furrr** package.

22.3 The **furrr** package

furrr, created and maintained by [Davis Vaughan](#), is built on top of **future**. Davis is a Software Engineer at R-Studio and started contributing to the package in 2018, releasing the first version on GitHub. The primary concept behind **furrr** is to harness the capabilities of the **future** package with **purrr**-like functions. The idea is straightforward: **furrr** includes the typical **purrr** functions for iterations, each of which is prefixed with **future_**:

- `map -> future_map`
- `map2 -> future_map2`
- `pmap -> future_pmap`
- `walk -> future_walk`
- `walk2 -> future_walk2`
- `pwalk -> future_pwalk`

The **furrr** functions are intended to be a “near drop in replacement for **purrr** functions” (<https://furrr.futureverse.org/>). The developers have made it remarkably straightforward to combine **furrr** with **future**. You need to ensure two things. First, you must still define the `plan()` and specify whether you want to execute the computation sequentially, in multisession, or in multicore. Second, you must replace your **purrr** function with the corresponding **furrr** function. That’s all there is to it.

We previously increased the speed of reading our files by a factor of five by splitting the CSV file paths into bins and using implicit `future()` calls. With **furrr**, the code now looks as follows:

```
library(furrr)

plan(multisession)
```

```

tic()
diamonds_data_complete_fast_futuremap <- future_map(
  .x = csv_files,
  .f = \`(file_path) read_csv(file_path) |>
    mutate(name = file_path)
) |>
  list_rbind()
toc()

```

14.813 sec elapsed

```
plan(sequential)
```

Again, the code takes 15 seconds to execute. The advantage is that the code is significantly more straightforward to comprehend, and we no longer need to contemplate how to utilize the parallel processing abilities of `future`.

22.4 Simulating the speed of furrr

To fully appreciate the time-saving benefits of `furrr` and `future`, we can conduct a simulation using various parameters. In this simulation, I once again use the same code as before to load numerous CSV files into memory, but with varying two parameters: the number of cores for the multisession and the number of files that I will read into `future_map()`. Rather than delving into the intricacies of the code, I want to discuss the simulation's results instead.

```

simulation_data <- crossing(cores = c(1, 3, 8),
                           nr_of_files = c(50, 100, 300, 500, 1000))

res <- pmap(
  .l = simulation_data,
  .f = \`(cores, nr_of_files) {

    plan(multisession, workers = cores)

    tic()
    temp_res <- future_map(
      .x = csv_files |> sample(nr_of_files, replace = TRUE),
      .f = \`(file_path) read_csv(file_path) |>
        mutate(name = file_path)
    )
  }
)

```

```

) |>
  list_rbind()

time_results <- toc()

plan(sequential)
rm(temp_res)

tibble(
  cores = cores,
  nr_of_files = nr_of_files,
  duration = time_results$toc - time_results$tic
)
}

) |> list_rbind()

```

```

1.324 sec elapsed
2.553 sec elapsed
7.555 sec elapsed
13.245 sec elapsed
28.694 sec elapsed
0.759 sec elapsed
1.29 sec elapsed
4.101 sec elapsed
7.588 sec elapsed
18.688 sec elapsed
0.764 sec elapsed
1.071 sec elapsed
3.114 sec elapsed
5.683 sec elapsed
15.084 sec elapsed

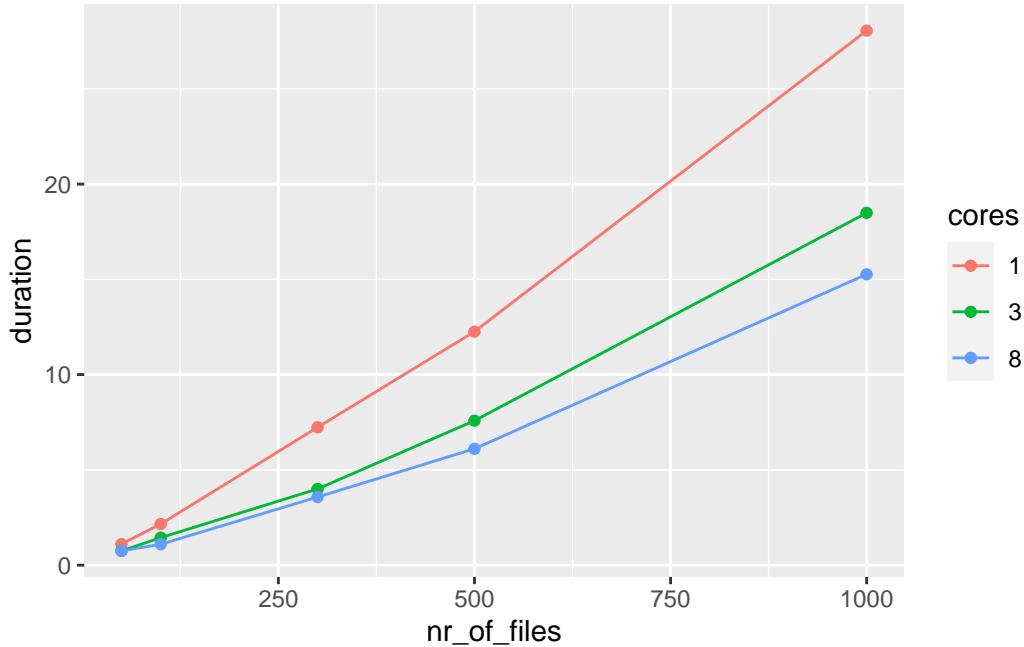
```

The simulation results are stored in the `res` data frame. We can visualize the results with `ggplot2`:

```

res |>
  mutate(cores = as_factor(cores)) |>
  ggplot(aes(x = nr_of_files, y = duration,
             color = cores)) +
  geom_line() +
  geom_point()

```



It turns out that the more files that have to be read into the memory, the longer the calculations take. This is no surprise. We also see that the calculations are always faster with more cores. The other finding is that using three or eight cores is massively more efficient than using one core. However, it doesn't make too much difference whether you use 3 or 8 cores.

One might assume that **furrr** provides the same computational benefits when used with nested data. However, when working with grouped and nested data, **furrr** is actually slower than the **purrr** functions. To illustrate this, consider the following simulation. We generate datasets of different sizes, nest the simulated data, and compute linear models for each nested dataset:

```
simulation_data <- crossing(cores = c(1, 3, 8), bootstraps = c(1, 3, 5, 7))

res_nested <- pmap(
  .l = simulation_data,
  .f = \(cores, bootstraps) {

    plan(multisession, workers = cores)

    bootstrapped_data <- map(1:bootstraps,
      \(x) slice_sample(diamonds,
        prop = 1, replace = TRUE)) |>
      
```

```

list_rbind()

tic()

simulation_res <- bootstrapped_data |>
  nest(data = -cut) |>
  mutate(
    model = future_map(.x = data,
                       .f = \((x) lm(x ~ z, data = x))
  )

time_results <- toc()

plan(sequential)

tibble(
  cores = cores,
  duration = time_results$toc - time_results$tic,
  nrow = nrow(bootstrapped_data)
)

}

) |> list_rbind()

```

0.019 sec elapsed
 0.049 sec elapsed
 0.042 sec elapsed
 0.051 sec elapsed
 0.162 sec elapsed
 0.277 sec elapsed
 0.443 sec elapsed
 0.612 sec elapsed
 0.301 sec elapsed
 0.468 sec elapsed
 0.732 sec elapsed
 1.015 sec elapsed

```

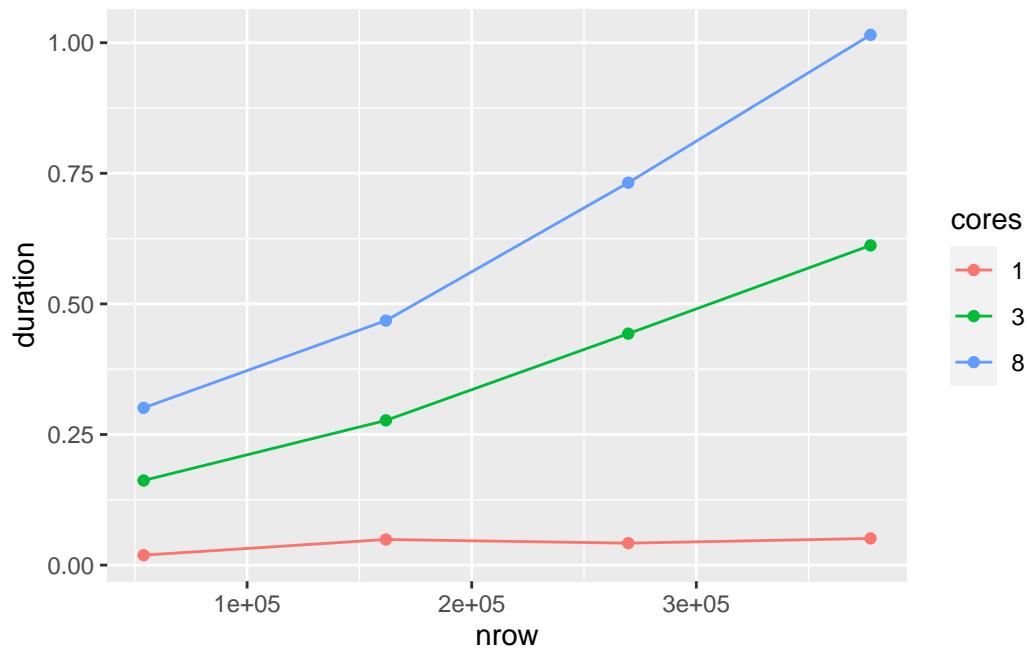
res_nested |>
  mutate(
    cores = as.factor(cores)) |>

```

```

mutate(cores = as_factor(cores)) |>
  ggplot(aes(x = nrow, y = duration,
             color = cores)) +
  geom_line() +
  geom_point()

```



Ironically, utilizing only one worker is the fastest method. This is a well-known caveat in **furrr** and may be related to the issue of [data transfer](#). More information about this issue can be found here: <https://furrr.futureverse.org/articles/gotchas.html#grouped-data-frames>. I would like to provide you with a solution to this problem, but regrettably, I have not discovered one yet. Even chatGPT didn't find a workaround (Prompt: "Can you rewrite this code so it actually runs faster with furrr").

i Summary

- **future** is a high-level API that can tap into the parallel processing capabilities of modern computers. **furrr** is build on top of **future** by using purrr-like functions to speed up iterations in R.
- If you want to use **future**, there are three simple steps you need to follow: First, you need to set a `plan()`. Then, you wrap the computations that you want to run in parallel in `future()`. Finally, you retrieve the results using `value()`.

- You can run `future` explicitly by using `future()` and `)value()`, or implicitly by using the `%<-%` operator.
- If you want to measure how fast your computations are running, you can use the `tic` and `toc` functions from the `tictoc` package.
- While `furrr` is a great tool, it doesn't work well with grouped and nested data. In fact, it can even run slower than `purrr` functions in certain cases.

23 How to speed up your data analysis with `dplyr`

What will this tutorial cover?

In this tutorial, we will take a brief tour through the `dplyr` package. `dplyr` was built on top of `data.table` and leverages the strengths of `data.table`, which is speed, and `dplyr`, which is expressiveness.

💡 Who do I have to thank?

For this tutorial, I have to thank the developers of `dplyr` (see [Contributors on GitHub](#)), in particular Hadley Wickham and Romain François, and the developers of `data.table` in particular [Matt Dowle](#).

In 2009, `data.table` was released by [Matt Dowle](#) on GitHub. The objective of `data.table` is to enhance the speed and memory efficiency of handling massive data sets in R, such as those that occupy multiple gigabytes in RAM.

`data.table` delivers on its promise of being fast. A few benchmark tests have shown its speed superiority over `dplyr` and base R (see [Tyson Barrett](#), [Brodie Gaslam](#), and [Iyar Lin](#)). For data sets larger than 5GB, `data.table` outperforms `dplyr` substantially. However, speed is not the only factor to consider when selecting a package for manipulating data; readability is also important. Opinions on the readability of `data.table` vary greatly, as evidenced by debates on [Reddit - “dplyr vs data.table”](#) and [Stackoverflow - “data.table vs dplyr: can one do something well the other can’t or does poorly?”](#). Some users appreciate the `data.table`'s conciseness, while others find it challenging to read.

Fortunately, making that decision is no longer necessary. In 2016, the first version of `dplyr` was released on [GitHub](#). The aim of `dplyr` is to convert `dplyr` code to `data.table` code, capitalizing on `data.table`'s advantages in speed and `dplyr`'s advantage in expressiveness.

It is worth noting, however, that `dplyr` is not as speedy as `data.table`. The conversion of `dplyr` syntax to `data.table` involves some overhead (see [Why is dplyr slower than data.table?](#) for more information), which is largely unnoticeable and negligible.

`dplyr` supports nearly all `dplyr` verbs (see <https://dplyr.tidyverse.org/reference/index.html>) and can be set up with just a few steps:

- Load `dplyr` and `data.table`
- Convert your data frames or tibbles into a lazy data table using `lazy_dt()`. Lazy, because the `dplyr` functions only run when results are requested by functions like `as_tibble()` or `as.data.frame()`
- Execute your `dplyr` functions with the lazy data table.
- Convert your data frame back to a tibble with `as_tibble()` or to a data frame with `as.data.frame()` to indicate that you are done with your data transformations

Below is a simple example where we calculate the mean of each `id` in the data frame:

```
library(data.table)
library(dplyr)
library(tidyverse)

dt <- lazy_dt(data.frame(x = 1:10, id = 1:2))
(dt_new <- dt |>
  summarise(mean = mean(x), .by = id) |>
  as_tibble())

# A tibble: 2 x 2
  id   mean
  <int> <dbl>
1     1     5
2     2     6
```

If you use both `data.table` and `dplyr`, you may be interested in knowing how to write equivalent `data.table` code for a computation in `dplyr`. To see how `dplyr` converts `dplyr` code to `data.table`, you can use the `show_query()` function at the end of your data transformations:

```
dt |>
  filter(x != 5) |>
  summarise(mean = mean(x), .by = id) |>
  show_query()

`_DT1`[x != 5, .(mean = mean(x)), keyby = .(id)]
```

`dplyr` is a nifty add-on to your toolkit when you care about speed. By using it in tandem with `furrr` and `future`, this package allows you to eke-out all the performance benefits from R you need.

Summary

- The primary goal of `data.table` is to facilitate the manipulation of large datasets, even those that are several gigabytes in size, in a fast and memory-efficient manner.
- `dplyr` leverages the strengths of `data.table`, which is speed, and `dplyr`, which is expressiveness.
- Although `dplyr` is slightly slower than `data.table`, the difference is negligible.
- Before using `dplyr`, you must first convert your data frames or tibbles into a lazy data table. Once you've executed the `dplyr` functions, you can convert the lazy data table back into a data frame or tibble to conclude the data transformation.