

Tidyverse crash course @ iDiv

Emilio Berti

23/09/2022

Today's program

- General intro (10 minutes).
- Packages and hands-on sessions.
- Datasets: *kalenji.csv* and *garmin.csv*.

Today's program

- General intro (10 minutes).
- Packages and hands-on sessions.
- Datasets: *kalenji.csv* and *garmin.csv*.



Figure 1: Sampling equipment

Today's program

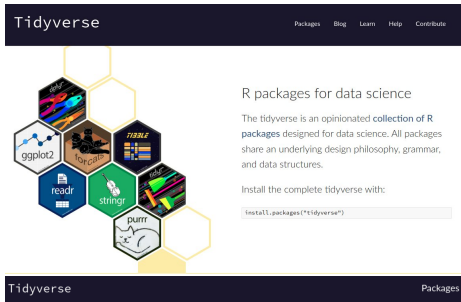
- General intro (10 minutes).
- Packages and hands-on sessions.
- Datasets: *kalenji.csv* and *garmin.csv*.



Figure 2: Sampling equipment

- 1 Introduce packages, functionalities, and data wrangling.
- 2 Transform tables into what you want.
- 3 Work with more than one table: merge them, compare them, etc.

What is the tidyverse?



The screenshot shows the Tidyverse website. At the top is a dark blue header with the word "Tidyverse" in white and navigation links: Packages, Blog, Learn, Help, and Contribute. Below the header is a grid of hexagonal icons representing various R packages: dplyr, ggplot2, forcats, tidyr, readr, stringr, and purrr. To the right of the icons, the text reads: "R packages for data science. The tidyverse is an opinionated collection of R packages designed for data science. All packages share an underlying design philosophy, grammar, and data structures. Install the complete tidyverse with: `install.packages('tidyverse')`". At the bottom of the page, there is a dark blue footer with the word "Tidyverse" on the left and "Packages" on the right.

Posts



GitHub Actions for R developers, v2

Gábor Csárdi

programming

We have updated our GitHub Actions at `r-lib/actions`. Consider upgrading to the new v2 version, for faster and more reliable GHA jobs.

2022/06/01



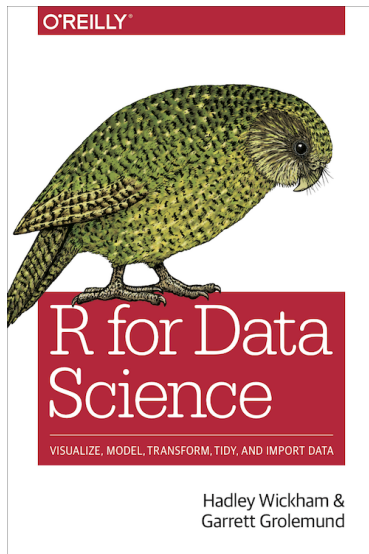
roxygen2 7.2.0

Hadley Wickham

package

roxygen2 7.2.0 brings improvements to `NAMESPACE` generation, better multiparameter argument inheritance, and improved warnings.

2022/05/13



The image shows the front cover of the book "R for Data Science" by Hadley Wickham and Garrett Golemund. The top of the cover has a red banner with the "O'REILLY" logo in white. Below the banner is a detailed illustration of a green parrot with yellow and black patterns on its head and neck. The title "R for Data Science" is written in large, white, serif font on a red background. Below the title, in smaller white capital letters, is the subtitle "VISUALIZE, MODEL, TRANSFORM, TIDY, AND IMPORT DATA". At the bottom right, the authors' names "Hadley Wickham & Garrett Golemund" are listed in black text.

Where to find these slides

<https://emilio-berti.github.io/Teaching.html>

Why the tidyverse? Because it's for lazy people.

d

##		date	km	time
## 1		10/27/2021	6.0	0:34:28
## 2		10/27/2021	6.0	0:33:24
## 3		10/31/2021	10.3	0:59:19
## 4		11/6/2021	8.7	0:54:00
## 5		11/13/2021	8.7	0:49:00
## 6		11/17/2021	4.6	0:25:28
## 7		11/21/2021	10.4	1:02:00
## 8		11/30/2021	6.0	0:34:08
## 9		11/30/2021	6.0	0:33:03
## 10		12/5/2021	6.2	0:35:00
## 11		12/9/2021	6.2	0:37:33
## 12		12/19/2021	6.2	0:35:22
## 13		12/22/2021	6.2	0:36:00
## 14		12/28/2021	6.2	0:35:00
## 15		12/31/2021	6.2	0:33:12
## 16		1/2/2022	6.2	0:33:43
## 17		1/12/2022	6.0	0:33:44
## 18		1/12/2022	6.0	0:33:45

Why the tidyverse? Because it's for lazy people.

```
as_tibble(d)
```

```
## # A tibble: 51 x 3
##   date          km time
##   <chr>        <dbl> <chr>
## 1 10/27/2021     6 0:34:28
## 2 10/27/2021     6 0:33:24
## 3 10/31/2021    10.3 0:59:19
## 4 11/6/2021     8.7 0:54:00
## 5 11/13/2021    8.7 0:49:00
## 6 11/17/2021    4.6 0:25:28
## 7 11/21/2021    10.4 1:02:00
## 8 11/30/2021     6 0:34:08
## 9 11/30/2021     6 0:33:03
## 10 12/5/2021    6.2 0:35:00
## # ... with 41 more rows
```


Why the tidyverse? Because it makes the code more readable.

```
sub_d <- d[d$km > 4, ]  
sub_d <- sub_d[sub_d$km < 6, ]  
summary(sub_d$km)
```

```
##      Min. 1st Qu.  Median    Mean 3rd Qu.    Max.  
##      4.6      4.6      4.6      4.6      4.6      4.6
```

Why the tidyverse? Because it makes the code more readable.

```
d %>%  
  as_tibble() %>%  
  filter(km > 4, km < 6) %>%  
  pull(km) %>%  
  summary()
```

```
##      Min. 1st Qu.  Median    Mean 3rd Qu.    Max.  
##      4.6     4.6     4.6     4.6     4.6     4.6
```

Why the tidyverse? Because you will learn SQL without knowing it.

```
d %>%  
  as_tibble() %>%  
  filter(km > 4, km < 6) %>%  
  pull(km) %>%  
  summary()
```

##	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
##	4.6	4.6	4.6	4.6	4.6	4.6

In SQL:

```
SELECT  
km  
FROM  
d  
WHERE km > 4 AND km < 6;
```

Basic ideas of tidyverse

- 1 Tables are the fundamental units of your analysis.
- 2 Tidy data: forces good practices for data science.
- 3 Complex table manipulations can be evaluated on the fly.
- 4 Table manipulations can be organized in modules.
- 5 Modules can be pipelined: `%>%`.

Basic ideas of tidyverse

Total distance and best time for each route - on the fly.

```
read_csv("kalenji.csv", show_col_types = FALSE) %>%
  transmute(
    time = minute(time) + hour(time) * 60, #total minutes
    Distance = round(km) #round distance
  ) %>%
  group_by(Distance) %>% #evaluate by group
  summarize(
    `Total distance` = sum(Distance),
    `Fastest time` = min(time)
  ) %>%
  arrange(desc(`Total distance`)) %>%
  knitr::kable()
```

Distance	Total distance	Fastest time
6	246	28
9	54	45
10	20	59
8	8	43
5	5	25

tibble - goal

Goal: table data keeping only good features of *data.frame*.

Pros:

- cleaner
- better problem detection
- no implicit modifications
- better print method
- can handle complex names: e.g., variable with spaces and / \$.

Cons:

- Some packages still requires a *data.frame* object.

tibble - what it looks like

```
d
```

##		date	km	time
## 1		10/27/2021	6.0	0:34:28
## 2		10/27/2021	6.0	0:33:24
## 3		10/31/2021	10.3	0:59:19
## 4		11/6/2021	8.7	0:54:00
## 5		11/13/2021	8.7	0:49:00
## 6		11/17/2021	4.6	0:25:28
## 7		11/21/2021	10.4	1:02:00
## 8		11/30/2021	6.0	0:34:08
## 9		11/30/2021	6.0	0:33:03
## 10		12/5/2021	6.2	0:35:00
## 11		12/9/2021	6.2	0:37:33
## 12		12/19/2021	6.2	0:35:22
## 13		12/22/2021	6.2	0:36:00
## 14		12/28/2021	6.2	0:35:00
## 15		12/31/2021	6.2	0:33:12
## 16		1/2/2022	6.2	0:33:43
## 17		1/12/2022	6.0	0:33:44
## 18		1/12/2022	6.0	0:33:45

tibble - from data.frames

```
as_tibble(d)
```

```
## # A tibble: 51 x 3
##   date          km time
##   <chr>        <dbl> <chr>
## 1 10/27/2021     6 0:34:28
## 2 10/27/2021     6 0:33:24
## 3 10/31/2021   10.3 0:59:19
## 4 11/6/2021     8.7 0:54:00
## 5 11/13/2021    8.7 0:49:00
## 6 11/17/2021    4.6 0:25:28
## 7 11/21/2021   10.4 1:02:00
## 8 11/30/2021     6 0:34:08
## 9 11/30/2021     6 0:33:03
## 10 12/5/2021    6.2 0:35:00
## # ... with 41 more rows
```

tibble - from matrices

```
mat <- matrix(1:100, 50, 2)
colnames(mat) <- c("x", "y")
as_tibble(mat)
```

```
## # A tibble: 50 x 2
##       x     y
##   <int> <int>
## 1     1     51
## 2     2     52
## 3     3     53
## 4     4     54
## 5     5     55
## 6     6     56
## 7     7     57
## 8     8     58
## 9     9     59
## 10    10     60
## # ... with 40 more rows
```

tibble - from lists

```
l <- list(x = 1:10, y = 1:10)
as_tibble(l)
```

```
## # A tibble: 10 x 2
##       x     y
##   <int> <int>
## 1     1     1
## 2     2     2
## 3     3     3
## 4     4     4
## 5     5     5
## 6     6     6
## 7     7     7
## 8     8     8
## 9     9     9
## 10    10    10
```

tibble - create new data

```
tibble(x = LETTERS[1:10], y = 1:10)
```

```
## # A tibble: 10 x 2
```

```
##       x           y
```

```
##   <chr> <int>
```

```
## 1 A           1
```

```
## 2 B           2
```

```
## 3 C           3
```

```
## 4 D           4
```

```
## 5 E           5
```

```
## 6 F           6
```

```
## 7 G           7
```

```
## 8 H           8
```

```
## 9 I           9
```

```
## 10 J          10
```

tibble - compatibility issues with packages

```
moveHMM::prepData(as_tibble(1))
```

```
## Warning: Unknown or uninitialised column: `ID`.
```

```
## <error/vctrs_error_subscript_oob>
```

```
## Error in `vectbl_as_col_location()`:
```

```
## ! Can't subset columns past the end.
```

```
## i Location 2 doesn't exist.
```

```
## i There is only 1 column.
```

```
## ---
```

```
## Backtrace:
```

```
## 1. base::tryCatch(moveHMM::prepData(as_tibble(1)), error = function(e) {
```

```
## 5. moveHMM::prepData(as_tibble(1))
```

```
## 7. tibble::`[.tbl_df`(x, i)
```

```
## 8. tibble:::vectbl_as_col_location(j, length(x), names(x), j_arg = j_arg)
```

tibble - compatibility issues: as.data.frame()

```
moveHMM::prepData(as.data.frame(l))
```

##	ID	step	angle	x	y
## 1	Animal1	156.8744	NA	1	1
## 2	Animal1	156.8276	0.0004610422	2	2
## 3	Animal1	156.7574	0.0006155637	3	3
## 4	Animal1	156.6639	0.0007703128	4	4
## 5	Animal1	156.5472	0.0009253809	5	5
## 6	Animal1	156.4073	0.0010808595	6	6
## 7	Animal1	156.2444	0.0012368401	7	7
## 8	Animal1	156.0586	0.0013934143	8	8
## 9	Animal1	155.8500	0.0015506736	9	9
## 10	Animal1	NA	NA	10	10

tibble - printing options

```
options(tibble.print_max = 4, tibble.print_min = 4)
as_tibble(1)
```

```
## # A tibble: 10 x 2
##       x     y
##   <int> <int>
## 1     1     1
## 2     2     2
## 3     3     3
## 4     4     4
## # ... with 6 more rows
```

tibble - practical 1 - problem

- 1 Create two vectors `x` and `y` with 10 random numbers (e.g. `runif()`).
- 2 Create a tibble with columns `x` and `y`.

tibble - practical 1 - solution

- 1 Create two vectors `x` and `y` with 10 random numbers (e.g. `runif()`).
- 2 Create a tibble with columns `x` and `y`.

```
x <- runif(10)
y <- runif(10)
tibble(x, y)
```

```
## # A tibble: 10 x 2
##       x       y
##   <dbl> <dbl>
## 1 0.367  0.290
## 2 0.970  0.823
## 3 0.521  0.779
## 4 0.0317 0.608
## # ... with 6 more rows
```

tibble - practical 2 - problem

- 1 Create a vector `x` with 10 random numbers (e.g. `runif()`).
- 2 Create a vector `y` with 5 random numbers (e.g. `runif()`).
- 3 Create a `data.frame` with columns `x` and `y`.
- 4 Create a tibble with columns `x` and `y`.

tibble - practical 2 - solution

- 1 Create a vector `x` with 10 random numbers (e.g. `runif()`).
- 2 Create a vector `y` with 5 random numbers (e.g. `runif()`).
- 3 Create a `data.frame` with columns `x` and `y`.

```
x <- runif(10)
y <- runif(5)
data.frame(x, y)
```

```
##           x           y
## 1  0.93175237 0.9835965
## 2  0.99575042 0.2037430
## 3  0.01037968 0.7845552
## 4  0.70850942 0.5372133
## 5  0.72394401 0.4363460
## 6  0.05190563 0.9835965
## 7  0.02298047 0.2037430
## 8  0.25231979 0.7845552
## 9  0.59340886 0.5372133
## 10 0.20372332 0.4363460
```

tibble - practical 2 - solution

- 1 Create a vector `x` with 10 random numbers (e.g. `runif()`).
- 2 Create a vector `y` with 5 random numbers (e.g. `runif()`).
- 3 Create a data.frame with columns `x` and `y`.
- 4 Create a tibble with columns `x` and `y`.

```
x <- runif(10)
y <- runif(5)
tibble(x, y)
```

```
## <error/tibble_error_incompatible_size>
## Error:
## ! Tibble columns must have compatible sizes.
## * Size 10: Existing data.
## * Size 5: Column at position 2.
## i Only values of size one are recycled.
## ---
## Backtrace:
## 1. base::tryCatch(...)
## 5. tibble::tibble(x, y)
## 6. tibble:::tibble_quos(xs, .rows, .name_repair)
## 7. tibble:::vectbl_recycle_rows(res, first_size, j, given_col_names[[j])
```

tibble - practical 2 - solution

```
x <- runif(10)
tibble(x, y = 0)
```

```
## # A tibble: 10 x 2
##       x       y
##   <dbl> <dbl>
## 1 0.410     0
## 2 0.529     0
## 3 0.150     0
## 4 0.515     0
## # ... with 6 more rows
```


Goal: load tabular data from delimited files (comma-separated).

Pros:

- fast and friendly
- support many types of data
- informative table summary and problem reports

Cons:

- syntax can be sometimes frustrating

readr - read data

```
d <- read.csv("kalenji.csv")  
str(d)
```

```
## 'data.frame':    51 obs. of  3 variables:  
## $ date: chr  "10/27/2021" "10/27/2021" "10/31/2021" "11/6/2021" ...  
## $ km : num  6 6 10.3 8.7 8.7 4.6 10.4 6 6 6.2 ...  
## $ time: chr  "0:34:28" "0:33:24" "0:59:19" "0:54:00" ...
```


readr - read data

```
d <- read_csv("kalenji.csv")
```

```
## Rows: 51 Columns: 3
```

```
## -- Column specification -----
```

```
## Delimiter: ","
```

```
## chr   (1): date
```

```
## dbl   (1): km
```

```
## time  (1): time
```

```
##
```

```
## i Use `spec()` to retrieve the full column specification for this data.
```

```
## i Specify the column types or set `show_col_types = FALSE` to quiet this
```

readr - read data - suppress messages

```
d <- read_csv("kalenji.csv", show_col_types = FALSE) #suppress summary
```

readr - returns a tibble

readr loads directly a tibble:

```
read_csv("kalenji.csv", show_col_types = FALSE)
```

```
## # A tibble: 51 x 3
##   date          km time
##   <chr>        <dbl> <time>
## 1 10/27/2021     6 34'28"
## 2 10/27/2021     6 33'24"
## 3 10/31/2021  10.3 59'19"
## 4 11/6/2021    8.7 54'00"
## # ... with 47 more rows
```

readr - good column guessing

readr is smart:

```
d <- read_csv("kalenji.csv", show_col_types = FALSE)
d$time[2] - d$time[1]
```

```
## Time difference of -64 secs
```

readr - read general text delimited file

```
d <- read_delim("kalenji.txt", #general delimited file  
               delim = ";",  
               show_col_types = FALSE)
```

readr - write to file

readr can, of course, write to files.

```
write_csv(d, "copy-of-kalenji.csv")
```

readr - practical 1 - problem

- 1 Create a tibble with 10 random x and 10 random y .
- 2 Save it as *random-numbers.csv*.
- 3 Save it as *random-numbers.txt* using ; as separator.
- 4 Load them both.

readr - practical 1 - solution

- 1 Create a tibble with 10 random x and 10 random y .
- 2 Save it as *random-numbers.csv*.
- 3 Save it as *random-numbers.txt* using ; as separator.
- 4 Load them both.

```
d <- tibble(x = runif(10), y = runif(10))  
# write files  
write_csv(d, "random-numbers.csv")  
write_delim(d, "random-numbers.txt", delim = ";")  
# read files  
d_csv <- read_csv("random-numbers.csv", show_col_types = FALSE)  
d_semi <- read_delim("random-numbers.csv", show_col_types = FALSE, delim =
```


readr - really faster?

For a 24 Mb table with 29,000 rows:

	test	replications	elapsed	relative
2	base	10	4.942	5.291
3	data.table	10	0.934	1.000
1	readr	10	3.457	3.701

Before going deeper: *magrittr* pipe %>%.

command_1 **PIPE** *command_2* **PIPE** *command_3*

PIPE = take output from left and pass it as input to right.

EXECUTE *command_1* AND PASS ITS OUTPUT TO *command_2* AND PASS ITS OUTPUT TO *command_3*.

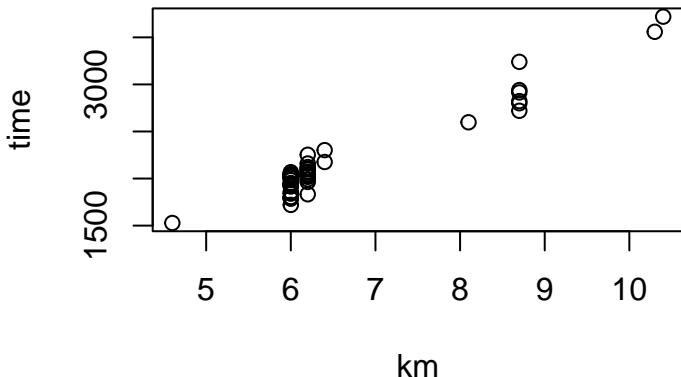
```
round(mean(abs(x))) = x %>% abs() %>% mean() %>% round()
```

magrittr - what's a pipe?

Before going deeper: *magrittr* pipe `%>%`.

command PIPE command PIPE command

```
d <- read_csv("kalenji.csv", show_col_types = FALSE)
d <- d[, c("km", "time")]
plot(d)
```

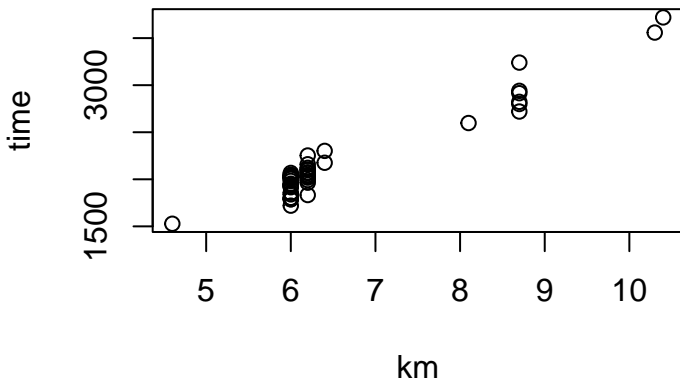


magrittr - pipe your way to glory!

Before going deeper: *magrittr* pipe `%>%`.

command PIPE command PIPE command

```
#make sure input/output is of the correct type  
read_csv("kalenji.csv", show_col_types = FALSE) %>% #output = tibble  
  select(km, time) %>% #output = tibble  
  plot() #output = figure
```

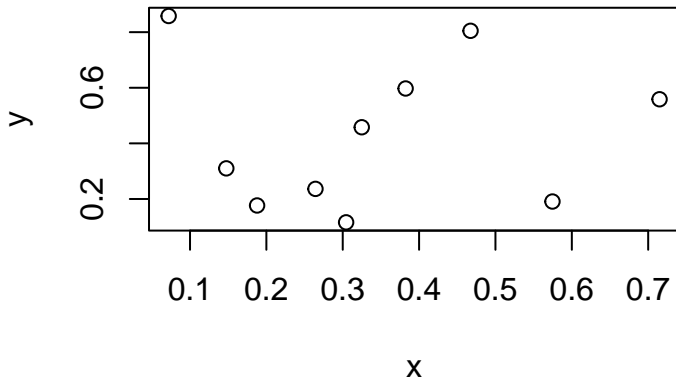


- 1 Create a tibble with x , y (`runif(10)`) and plot it without saving objects in the RAM.

magrittr - pipe practical 1 - solution

- 1 Create a tibble with x , y (`runif(10)`) and plot it without saving objects in the RAM.

```
tibble(x = runif(10), y = runif(10)) %>%  
  plot()
```

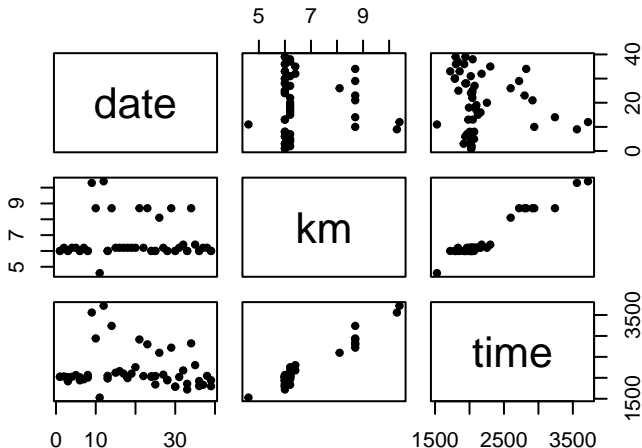


- 1 Load *kalenji.csv* and plot it without saving objects in the RAM.

magrittr - pipe practical 2 - solution

- 1 Load *kalenji.csv* and plot it without saving objects in the RAM.

```
read_csv("kalenji.csv", show_col_types = FALSE) %>% plot(pch = 20)
```



dplyr - goal

Goal: Standardize data manipulation.

Pros:

- intuitive verbs: *select*, *filter*, etc.
- extremely versatile.
- compact and modular complex data manipulations.

Cons:

- more complex manipulations can be scaring.

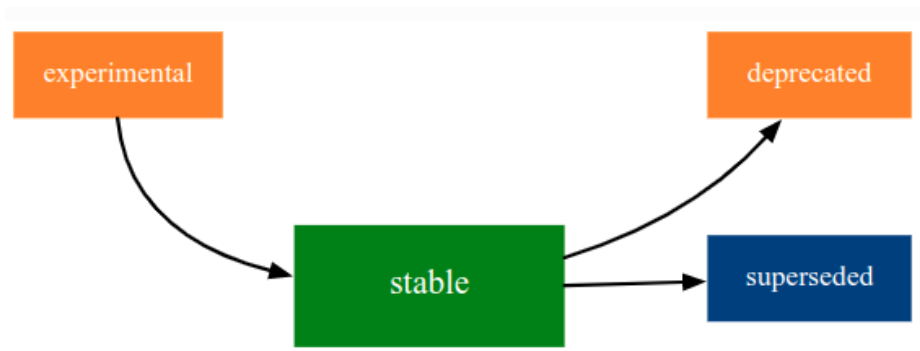


Figure 3: Life cycle of tidyverse functions

```
mutate_all {dplyr}
```

Mutate multiple columns

Description

lifecycle superseded

Scoped verbs (`_if`, `_at`, `_all`) have been superseded by the use of `across(.)`.

Figure 4: A superseded example

Common verbs:

- `filter()`: retain rows conditionally.
- `select()`: select columns and drop the rest.
- `mutate()`: create a new column.
- `pull()`: extract one column as a vector.
- `summarize()`: summarize columns.
- `group_by()`: *groups* tables into *sub-tables*. Each groups is manipulated separately.
- `arrange()`: sort rows by values.
- `slice()`: select rows by ID, randomly, firsts, etc.
- `join()`: **join tables**.

dplyr - filter()

`filter()` retains rows conditionally.

dplyr - filter() - example

`filter()` retains rows conditionally.

```
d %>% filter(km < 6)
```

```
## # A tibble: 1 x 2
##       km time
##   <dbl> <time>
## 1    4.6 25'28"
```

dplyr - filter() - more examples

filter() retains rows conditionally.

```
d %>% filter(km < 6)
d %>% filter(km == 6)
d %>% filter(km > 4, km < 8) #greater than 4 AND less than 8
d %>% filter(km < 4 | km > 8) #less than 4 OR greater than 8
```


dplyr - filter() - practical - problem

- 1 Retain only observations with even number of km ($\text{km} \% 2 == 0$).
- 2 Retain only observations between 5 and 7 km.

dplyr - filter() - practical - solution

- ④ Retain only observations with even number of km (`%% 2 == 0`).

```
d %>% filter(km %% 2 == 0)
```

```
## # A tibble: 24 x 2
##       km time
##   <dbl> <time>
## 1     6 34'28"
## 2     6 33'24"
## 3     6 34'08"
## 4     6 33'03"
## # ... with 20 more rows
```

dplyr - filter() - practical - solution

- 1 Retain only observations with even number of km (`%% 2 == 0`).
- 2 Retain only observations between 5 and 7 km.

```
d %>% filter(km >= 5, km <= 7)
```

```
## # A tibble: 41 x 2
##       km time
##   <dbl> <time>
## 1     6 34'28"
## 2     6 33'24"
## 3     6 34'08"
## 4     6 33'03"
## # ... with 37 more rows
```

dplyr - select()

`select()` selects some columns and drops the rest.

dplyr - select() - example

`select()` selects some columns and drops the rest.

```
d %>% select(km) #d[, "km"]
```

```
## # A tibble: 51 x 1
##       km
##   <dbl>
## 1     6
## 2     6
## 3  10.3
## 4   8.7
## # ... with 47 more rows
```

dplyr - select() - also reversed

You can also *non*-select columns with -

```
d %>% select(-km)
```

```
## # A tibble: 51 x 1
##   time
##   <time>
## 1 34'28"
## 2 33'24"
## 3 59'19"
## 4 54'00"
## # ... with 47 more rows
```

dplyr - tidyselect intermezzo

What if we want to select several columns conditionally?

```
tidysel <- tibble(  
  Outcome = runif(10, 2, 3),  
  `Predictor 1` = runif(10, -1, 1),  
  `Predictor 2` = runif(10, 10, 150),  
  `First levels` = rep(c("A", "B"), 5),  
  `Second levels` = rep(c("C", "D"), 5)  
)  
tidysel
```

```
## # A tibble: 10 x 5  
##   Outcome `Predictor 1` `Predictor 2` `First levels` `Second levels`  
##   <dbl>         <dbl>         <dbl> <chr>         <chr>  
## 1     2.37         0.328         72.2 A          C  
## 2     2.78        -0.619         87.7 B          D  
## 3     2.12        -0.924        133. A          C  
## 4     2.66         0.282         64.6 B          D  
## # ... with 6 more rows
```

tidyselect allows this flexibility.

dplyr - tidysel intermezzo - select columns starting with "Pred"

```
tidysel %>% select(starts_with("Pred"))
```

```
## # A tibble: 10 x 2
##   `Predictor 1` `Predictor 2`
##         <dbl>         <dbl>
## 1         0.328         72.2
## 2        -0.619         87.7
## 3        -0.924        133.
## 4         0.282         64.6
## # ... with 6 more rows
```


dplyr - tidyselect intermezzo - select columns containing "levels"

```
tidysel %>% select(contains("levels"))
```

```
## # A tibble: 10 x 2
##   `First levels` `Second levels`
##   <chr>         <chr>
## 1 A           C
## 2 B           D
## 3 A           C
## 4 B           D
## # ... with 6 more rows
```

dplyr - tidysel intermezzo - select only numeric columns

For this, we need to use also the `where` helper, which checks conditions on columns.

Conditions must return TRUE or FALSE, e.g. `is.numeric`.

```
tidysel %>% select(where(is.numeric))
```

```
## # A tibble: 10 x 3
##   Outcome `Predictor 1` `Predictor 2`
##   <dbl>      <dbl>      <dbl>
## 1    2.37      0.328      72.2
## 2    2.78     -0.619      87.7
## 3    2.12     -0.924     133.
## 4    2.66      0.282      64.6
## # ... with 6 more rows
```

dplyr - tidyselct intermezzo - where

where is extremely useful for many verbs in *dplyr*. We will see more cases later.

You can already appreciate its simplicity, though:

```
# in base R
pca <- prcomp(tidysel[, c("Outcome", "Predictor 1", "Predictor 2")])
```

```
# in tidyverse
pca <- tidysel %>%
  select(where(is.numeric)) %>%
  prcomp()
```

Application: PCAs on climate data

dplyr - tidysel intermezzo - selection alternatives

In base R there are several way to do the same as:

```
tidysel %>% select(where(is.numeric))
```

This is not safe:

```
tidysel[, c(1, 2, 3)]  
#what if the position of columns changes?
```

This is not scalable:

```
tidysel[, c("Outcome", "Predictor 1", "Predictor 2")]  
#what if you add a new numeric column?
```

The only safe and scalable way except the tidyverse one is:

```
coltypes <- sapply(colnames(tidysel), \(x) class(tidysel[[x]]))  
tidysel[, coltypes == "numeric"]
```



Figure 5: Don't drink soap.

dplyr - mutate()

`mutate()` creates new columns (or modifies existing ones).

dplyr - mutate() - example

`mutate()` creates new columns (or modifies existing ones).

```
library(lubridate) #this will save you time  
d <- read_csv("kalenji.csv", show_col_types = FALSE)  
d <- d %>% mutate(Date = as_date(date, format = "%m/%d/%y"))
```

dplyr - mutate() - compact it please.

```
d <- read_csv("kalenji.csv", show_col_types = FALSE)
d <- d %>% mutate(Date = as_date(date, format = "%m/%d/%y"))
d
```

```
## # A tibble: 51 x 4
##   date          km time    Date
##   <chr>        <dbl> <time> <date>
## 1 10/27/2021     6   34'28" 2020-10-27
## 2 10/27/2021     6   33'24" 2020-10-27
## 3 10/31/2021   10.3  59'19" 2020-10-31
## 4 11/6/2021     8.7  54'00" 2020-11-06
## # ... with 47 more rows
```

Can you compress this even more and make it more readable?

Do we need the old *date* column?

dplyr - mutate() - compacted

```
d <- read_csv("kalenji.csv", show_col_types = FALSE) %>%  
  mutate(date = as_date(date, format = "%m/%d/%y"))
```

dplyr - mutate() - practical - problem

- 1 Create three new columns (*month*, *week*, *day*) using *lubridate* `month()`, `week()`, and `day()`.

dplyr - mutate() - practical - solution

- 1 Create three new columns (*month*, *week*, *day*) using *lubridate* *month()*, *week()*, and *day()*.

```
d <- d %>%  
  mutate(  
    month = month(date, label = TRUE, abbr = FALSE),  
    week = week(date),  
    day = day(date)  
  )  
d
```

```
## # A tibble: 51 x 6  
##   date          km time  month    week  day  
##   <date>      <dbl> <time> <ord>    <dbl> <int>  
## 1 2020-10-27     6  34'28" October    43    27  
## 2 2020-10-27     6  33'24" October    43    27  
## 3 2020-10-31    10.3 59'19" October    44    31  
## 4 2020-11-06     8.7 54'00" November    45     6  
## # ... with 47 more rows
```

dplyr - mutate() - tidyselect

mutate() can make use of *tidyselect*, but within the **across()** helper.

The syntax always looks like: mutate(**across**(**where**(*condition*), fn)).

```
tidysel %>% mutate(across(starts_with("Pred"), round, 1))
```

```
## # A tibble: 10 x 5
##   Outcome `Predictor 1` `Predictor 2` `First levels` `Second levels`
##   <dbl>         <dbl>         <dbl> <chr>         <chr>
## 1     2.37           0.3           72.2 A           C
## 2     2.78          -0.6           87.7 B           D
## 3     2.12          -0.9          133. A           C
## 4     2.66           0.3           64.6 B           D
## # ... with 6 more rows
```

```
tidysel %>% mutate(across(where(is.character), tolower))
```

```
## # A tibble: 10 x 5
##   Outcome `Predictor 1` `Predictor 2` `First levels` `Second levels`
##   <dbl>         <dbl>         <dbl> <chr>         <chr>
## 1     2.37           0.328           72.2 a           c
## 2     2.78          -0.619           87.7 b           d
## 3     2.12          -0.924          133. a           c
```

dplyr - pull()

`pull()` extracts one column as vector.

dplyr - pull() - example

`pull()` extracts one column as vector.

```
d %>% pull(km)
```

[illegible]

dplyr - pull() - practical - problem

- 1 Get the `table()` of *month* and sort it by count.

dplyr - pull() - practical - solution

- Get the `table()` of *month* and sort it by count.

```
d %>%  
  pull(month) %>%  
  table() %>%  
  sort()
```

```
## .  
##      May      June      July      August September      April      October      N  
##      0        0        0        0          0          3          3  
## December  January February      March  
##      6        10        11        12
```


dplyr - summarize()

`summarize()` summarizes columns.

dplyr - summarize() - example

summarize() summarizes columns.

```
d %>% summarize(  
  `shortest distance` = min(km),  
  `average distance` = mean(km),  
  `maximum distance` = max(km)  
)
```



```
## # A tibble: 1 x 3  
##   `shortest distance` `average distance` `maximum distance`  
##           <dbl>           <dbl>           <dbl>  
## 1             4.6             6.58             10.4
```

dplyr - summarize() - practical 1 - problem

- 1 Summarize *Total km* as sum of all distances and *Total time* as sum of all times.

dplyr - summarize() - practical 1 - solution

- Summarize *Total km* as sum of all distances and *Total time* as sum of all times (as hours).

```
d %>%  
  summarize(  
    `Total km` = sum(km),  
    `Total time` = (time %>% sum() %>% as.numeric()) / 3600  
  )
```

```
## # A tibble: 1 x 2  
##   `Total km` `Total time`  
##       <dbl>       <dbl>  
## 1       335.        30.7
```

dplyr - summarize() - tidyselect

```
tidysel %>% summarize(  
  across(contains("Pred"), #tidyselect  
    list(min = min, max = max), #summary stats  
    .names = "{col} : {fn}") #names of columns  
)
```

```
## # A tibble: 1 x 4
```

```
##   `Predictor 1 : min` `Predictor 1 : max` `Predictor 2 : min` `Predictor  
##           <dbl>           <dbl>           <dbl>  
## 1          -0.924           0.679           24.2
```

dplyr - summarize() - practical 2 - problem

- 1 Summarize all numeric variables to get mean and median values.

dplyr - summarize() - practical 2 - solution

- 1 Summarize all numeric variables to get mean and median values.

```
d %>%
  summarize(across(where(is.numeric),
                    list(mean = mean, median = median),
                    .names = "{fn} of {col}"))

## # A tibble: 1 x 6
##   `mean of km` `median of km` `mean of week` `median of week` `mean of d
##           <dbl>           <dbl>           <dbl>           <dbl>           <d
## 1           6.58           6.2           19.6           11           1
## # ... with 1 more variable: `median of day` <int>
```

dplyr - group_by()

dplyr operations can also be performed on groups within tables by using `group_by()`.

For instance, the sum of all distances for the whole *kalenji* dataset is:

```
d %>% summarize(`Total distance` = sum(km))
```

```
## # A tibble: 1 x 1
##   `Total distance`
##               <dbl>
## 1             335.
```


dplyr - group_by() - example

To get the summary *sum* for each month separately, we just need to specify *month* as the grouping variable. dplyr would then perform the next operation on each group separately:

```
d %>%  
  group_by(month) %>%  
  summarize(`Total distance` = sum(km))
```

```
## # A tibble: 7 x 2  
##   month   `Total distance`  
##   <ord>         <dbl>  
## 1 January         60.8  
## 2 February        73.9  
## 3 March           78.6  
## 4 April           18.2  
## # ... with 3 more rows
```

dplyr - group_by() - practical 1 - problem

- 1 Count the number of observation in each month

dplyr - group_by() - practical 1 - solution

- 1 Count the number of observation in each month

```
d %>%  
  group_by(month) %>%  
  summarize(n = length(km))
```

```
## # A tibble: 7 x 2  
##   month      n  
##   <ord>   <int>  
## 1 January    10  
## 2 February   11  
## 3 March      12  
## 4 April       3  
## # ... with 3 more rows
```

dplyr - group_by() - practical 1 - tally()

- 1 Count the number of observation in each month

```
d %>%  
  group_by(month) %>%  
  tally()  
  
## # A tibble: 7 x 2  
##   month      n  
##   <ord>    <int>  
## 1 January    10  
## 2 February   11  
## 3 March      12  
## 4 April       3  
## # ... with 3 more rows
```

dplyr - group_by() - practical 2 - problem

- 1 Calculate the total distance per week within months

dplyr - group_by() - practical 2 - solution

- 1 Calculate the total distance per week within months

```
d %>%  
  group_by(month, week) %>%  
  summarize(`Total km` = sum(km)) %>%  
  filter(week > 3) #simply to show you why week and month
```

```
## `summarise()` has grouped output by 'month'. You can override using the  
## `.groups` argument.
```

```
## # A tibble: 24 x 3  
## # Groups:   month [7]  
##   month      week `Total km`  
##   <ord>    <dbl>     <dbl>  
## 1 January      4       18.2  
## 2 January      5        6.2  
## 3 February     5        12  
## 4 February     6       18.2  
## # ... with 20 more rows
```

The output is still grouped.

dplyr - group_by() - practical 2 - ungroup()

If you don't need groups, `ungroup()` the table, or you may get unexpected results later on.

```
d %>%  
  group_by(month, week) %>%  
  summarize(`Total km` = sum(km), .groups = "drop")
```

or

```
d %>%  
  group_by(month, week) %>%  
  summarize(`Total km` = sum(km)) %>%  
  ungroup()
```

dplyr - group_by() - practical 2 - lifecycle?

`.groups` lifecycle experimental Grouping structure of the result.

- "drop_last": dropping the last level of grouping. This was the only supported option before version 1.0.0.
- "drop": All levels of grouping are dropped.
- "keep": Same grouping structure as `.data`.
- "rowwise": Each row is its own group.

When `.groups` is not specified, it is chosen based on the number of rows of the results:

- If all the results have 1 row, you get "drop_last".
- If the number of rows varies, you get "keep".

In addition, a message informs you of that choice, unless the result is ungrouped, the option "dplyr.summarise.inform" is set to FALSE, or when `summarise()` is called from a function in a package.

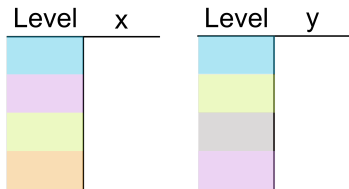
Figure 6: Always read the documentation

Lunch break

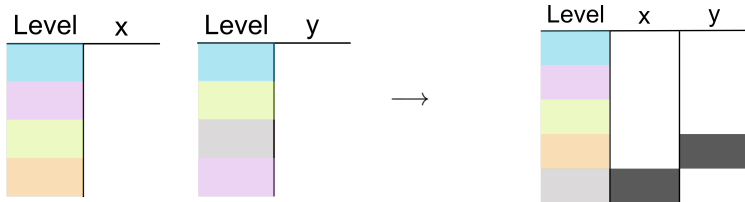


Figure 7: We'll be back at 1.

dplyr - join()



dplyr - join()



dplyr - join() - example

```
left_table <- tibble(Group = LETTERS[1:3], x = rnorm(3))
right_table <- tibble(Group = LETTERS[2:4], y = rnorm(3))
```

left_table

```
## # A tibble: 3 x 2
##   Group      x
##   <chr>  <dbl>
## 1 A      0.486
## 2 B     -0.554
## 3 C      0.692
```

right_table

```
## # A tibble: 3 x 2
##   Group      y
##   <chr>  <dbl>
## 1 B     -1.51
## 2 C     -1.78
## 3 D      0.447
```

dplyr - join() - types

Output tables has:

- ❶ `full_join(left_table, right_table)` - all levels
- ❷ `inner_join(left_table, right_table)` - only levels in both
- ❸ `left_join(left_table, right_table)` - only levels in *left_table*
- ❹ `right_join(left_table, right_table)` - only levels in *right_table*

left_table

```
## # A tibble: 3 x 2
##   Group      x
##   <chr>  <dbl>
## 1 A      0.486
## 2 B     -0.554
## 3 C      0.692
```

right_table

```
## # A tibble: 3 x 2
##   Group      y
##   <chr>  <dbl>
## 1 B     -1.51
## 2 C     -1.78
## 3 D      0.447
```

dplyr - join() - full join

left_table

```
## # A tibble: 3 x 2
##   Group      x
##   <chr>   <dbl>
## 1 A       0.486
## 2 B      -0.554
## 3 C       0.692
```

right_table

```
## # A tibble: 3 x 2
##   Group      y
##   <chr>   <dbl>
## 1 B      -1.51
## 2 C      -1.78
## 3 D       0.447
```

```
full_join(left_table, right_table)
```

```
## Joining, by = "Group"
```

```
## # A tibble: 4 x 3
##   Group      x      y
##   <chr>   <dbl> <dbl>
## 1 A       0.486  NA
## 2 B      -0.554 -1.51
## 3 C       0.692 -1.78
## 4 D       NA      0.447
```

dplyr - join() - inner join

```
left_table                                right_table
## # A tibble: 3 x 2                      ## # A tibble: 3 x 2
##   Group      x                          ##   Group      y
##   <chr>   <dbl>                        ##   <chr>   <dbl>
## 1 A       0.486                        ## 1 B      -1.51
## 2 B      -0.554                        ## 2 C      -1.78
## 3 C       0.692                        ## 3 D       0.447

inner_join(left_table, right_table)

## Joining, by = "Group"

## # A tibble: 2 x 3
##   Group      x      y
##   <chr>   <dbl> <dbl>
## 1 B      -0.554 -1.51
## 2 C       0.692 -1.78
```

dplyr - join() - left join

```
left_table                                right_table
## # A tibble: 3 x 2                      ## # A tibble: 3 x 2
##   Group      x                          ##   Group      y
##   <chr>   <dbl>                        ##   <chr>   <dbl>
## 1 A       0.486                        ## 1 B      -1.51
## 2 B      -0.554                        ## 2 C      -1.78
## 3 C       0.692                        ## 3 D       0.447

left_join(left_table, right_table)

## Joining, by = "Group"

## # A tibble: 3 x 3
##   Group      x      y
##   <chr>   <dbl> <dbl>
## 1 A       0.486 NA
## 2 B      -0.554 -1.51
## 3 C       0.692 -1.78
```


dplyr - join() - right join

```
left_table                                right_table
## # A tibble: 3 x 2                      ## # A tibble: 3 x 2
##   Group      x                          ##   Group      y
##   <chr>   <dbl>                        ##   <chr>   <dbl>
## 1 A       0.486                        ## 1 B      -1.51
## 2 B      -0.554                        ## 2 C      -1.78
## 3 C       0.692                        ## 3 D       0.447

right_join(left_table, right_table)

## Joining, by = "Group"

## # A tibble: 3 x 3
##   Group      x      y
##   <chr>   <dbl> <dbl>
## 1 B      -0.554 -1.51
## 2 C       0.692 -1.78
## 3 D       NA     0.447
```

dplyr - join() - by?

Joining requires (at least) one *joining* variable. Avoid ambiguity: specify it.

```
left_table %>% left_join(right_table, by = "Group")
```

```
## # A tibble: 3 x 3
##   Group      x      y
##   <chr> <dbl> <dbl>
## 1 A      0.486 NA
## 2 B     -0.554 -1.51
## 3 C      0.692 -1.78
```

dplyr - join() - by multiple variables

You can join by **multiple** variables.

```
left_table <- tibble(`First level` = sample(LETTERS, 100, replace = TRUE),  
                    `Second level` = sample(letters, 100, replace = TRUE),  
                    x = rnorm(100))  
right_table <- tibble(`First level` = sample(LETTERS, 100, replace = TRUE),  
                    `Second level` = sample(letters, 100, replace = TRUE),  
                    y = rnorm(100))  
inner_join(left_table, right_table, by = c("First level", "Second level"))
```

```
## # A tibble: 17 x 4  
##   `First level` `Second level`      x      y  
##   <chr>         <chr>         <dbl> <dbl>  
## 1 N           j           0.194 -0.537  
## 2 K           i           0.856  0.128  
## 3 S           c           0.0649 0.739  
## 4 Z           u           2.22   2.16  
## # ... with 13 more rows
```

dplyr - join() - by different names

Variables do not need to have the same name.

```
left_table <- tibble(`First level` = sample(LETTERS, 100, replace = TRUE),  
                    `Second level` = sample(letters, 100, replace = TRUE),  
                    x = rnorm(100))  
right_table <- tibble(`First group` = sample(LETTERS, 100, replace = TRUE),  
                    `Second group` = sample(letters, 100, replace = TRUE),  
                    y = rnorm(100))  
inner_join(left_table, right_table,  
           by = c("First level" = "First group",  
                 "Second level" = "Second group"))
```

```
## # A tibble: 14 x 4  
##   `First level` `Second level`      x      y  
##   <chr>        <chr>        <dbl> <dbl>  
## 1 U          z          -2.19 -0.125  
## 2 B          l          -0.345 -1.94  
## 3 S          b          -0.286  1.75  
## 4 B          s           0.855 -1.39  
## # ... with 10 more rows
```

dplyr - join() - practical - problem

I have the name of the routes.

```
routes <- read_csv("routes.csv", show_col_types = FALSE)
routes
```

```
## # A tibble: 6 x 2
##   distance `route name`
##       <dbl> <chr>
## 1      4.6 stadium
## 2       6 work
## 3      6.4 first bridge
## 4      8.7 second bridge
## # ... with 2 more rows
```

- 1 Assign the route names to the *kalenji.csv* table.
- 2 Count the number of km I ran for each route.
- 3 Arrange them in decreasing order.
- 4 Rename missing routes as: "no name".

dplyr - join() - practical - solution

- 1 Assign the route names to the *kalenji.csv* table.
- 2 Count the number of km I ran for each route.
- 3 Arrange them in decreasing order.
- 4 Rename missing routes as: "no name".

```
d %>%  
  left_join(routes, by = c("km" = "distance")) %>%  
  group_by(`route name`) %>%  
  summarise(`Total km` = sum(km)) %>%  
  arrange(desc(`Total km`)) %>%  
  mutate(`route name` = ifelse(is.na(`route name`), "no name", `route name`))
```

```
## # A tibble: 6 x 2  
##   `route name` `Total km`  
##   <chr>         <dbl>  
## 1 work          144  
## 2 no name       101.  
## 3 second bridge  52.2  
## 4 train lake    20.7  
## # ... with 2 more rows
```

The *garmin.csv* dataset

garmin dataset

```
garmin <- read_csv("garmin.csv", show_col_types = FALSE)
```

```
## New names:
```

```
## * `Avg Run Cadence` -> `Avg Run Cadence...10`
```

```
## * `Max Run Cadence` -> `Max Run Cadence...11`
```

```
## * `Avg Run Cadence` -> `Avg Run Cadence...20`
```

```
## * `Max Run Cadence` -> `Max Run Cadence...21`
```

```
garmin
```

```
## # A tibble: 74 x 41
```

```
##   `Activity Type` Date           Favorite Title      Distance Calorie
```

```
##   <chr>           <dtm>           <lgl>   <chr>         <dbl>    <dbl>
```

```
## 1 Running      2022-09-08 08:19:19 FALSE    Stara Z~    5.35      48
```

```
## 2 Running      2022-09-04 10:48:08 FALSE    Leipzig~   8.41      69
```

```
## 3 Running      2022-09-02 07:04:01 FALSE    Leipzig~   6.76      52
```

```
## 4 Running      2022-09-01 16:42:50 FALSE    Leipzig~   5.7       45
```

```
## # ... with 70 more rows, and 34 more variables: `Avg HR` <dbl>, `Max HR`
```

```
## #   `Avg Run Cadence...10` <dbl>, `Max Run Cadence...11` <dbl>,
```

```
## #   `Avg Pace` <chr>, `Best Pace` <chr>, `Total Ascent` <chr>,
```

```
## #   `Total Descent` <chr>, `Avg Stride Length` <dbl>,
```

```
## #   `Avg Vertical Ratio` <dbl>, `Avg Vertical Oscillation` <dbl>,
```


garmin dataset - practical 1 - problem

- 1 Filter only *Running* entries.
- 2 Select only relevant columns: *Date*, *Distance*, *Time*, *Avg HR*.
- 3 Create new column $Speed = Distance / Time$ (km/h).
- 4 Plot them using `pairs()`.

garmin dataset - practical 1 - solution

- 1 Filter only *Running* entries.
- 2 Select only relevant columns: *Date*, *Distance*, *Time*, *Avg HR*.

```
garmin <- read_csv("garmin.csv", show_col_types = FALSE) %>%  
  filter(`Activity Type` == "Running") %>%  
  select(Date, Distance, Time, `Avg HR`)
```

New names:

```
## * `Avg Run Cadence` -> `Avg Run Cadence...10`  
## * `Max Run Cadence` -> `Max Run Cadence...11`  
## * `Avg Run Cadence` -> `Avg Run Cadence...20`  
## * `Max Run Cadence` -> `Max Run Cadence...21`
```

garmin dataset - practical 1 - solution

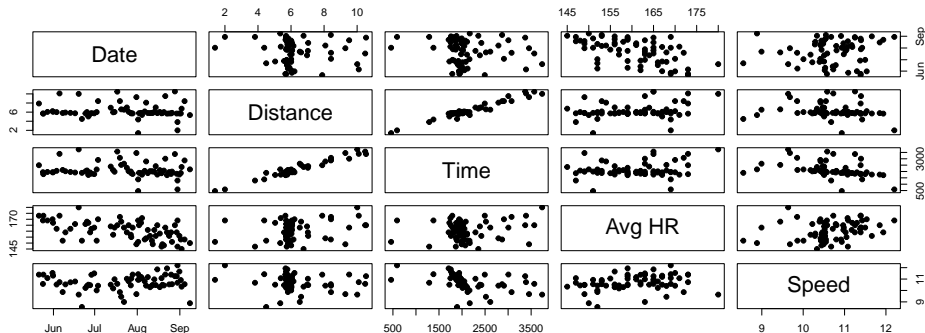
- 1 Filter only *Running* entries.
- 2 Select only relevant columns: *Date*, *Distance*, *Time*, *Avg HR*.
- 3 Create new column *Speed* = *Distance* / *Time* (km/h).

```
garmin <- garmin %>%  
  mutate(Hours = hour(Time) + minute(Time) / 60 + second(Time) / 3600,  
         Speed = Distance / Hours) %>%  
  select(-Hours)
```

garmin dataset - practical 1 - solution

- 1 Filter only *Running* entries.
- 2 Select only relevant columns.
- 3 Create new column $Speed = Distance / Time$ (km/h).
- 4 Plot them using `pairs()`.

```
pairs(garmin, pch = 20, cex = 1.5)
```



garmin dataset - practical 2 - problem

- 1 Round distances for both datasets *kalenji* and *garmin* to integers.
- 2 Calculate average *Speed* per *Distance*.
- 3 (Full) join the tables returning *Distance*, *Time*, and *Average Speed* (km/h).

garmin dataset - practical 2 - solution

```
kalenji <- read_csv("kalenji.csv", show_col_types = FALSE) %>%  
  mutate(Hours = hour(time) + minute(time) / 60 + second(time) / 3600,  
         Speed = km / Hours) %>%  
  transmute(Distance = km, Time = time, Speed)
```

garmin dataset - practical 2 - solution

- 1 Round distances for both datasets *kalenji* and *garmin* to integers.
- 2 Calculate average *Speed* per *Distance*.
- 3 Join the tables returning *Distance*, *Time*, and *Average Speed* (km/h).

For each dataset: `round() %>% summarize() %>% join()`.

garmin dataset - practical 2 - solution

- 1 Round distances for both datasets *kalenji* and *garmin* to integers.
- 2 Calculate average *Speed* per *Distance*.
- 3 Join the tables returning *Distance*, *Time*, and *Average Speed* (km/h).

For each dataset: `round() %>% summarize() %>% join()`.

However, one `round() %>% summarize()` can be nested into `join()`:

```
left_table %>%  
  round() %>%  
  summarize() %>%  
  join(  
    right_table() %>%  
      round() %>%  
      summarize()  
  )
```


garmin dataset - practical 2 - solution

- 1 Round distances for both datasets *kalenji* and *garmin* to integers.
- 2 Calculate average *Speed* per *Distance*.
- 3 Join the tables returning *Distance*, *Time*, and *Average Speed* (km/h).

```
avg_speed <- kalenji %>%  
  mutate(Distance = round(Distance)) %>%  
  group_by(Distance) %>%  
  summarize(`Average Speed` = mean(Speed)) %>%  
  full_join(  
    garmin %>%  
      mutate(Distance = round(Distance)) %>%  
      group_by(Distance) %>%  
      summarize(`Average Speed` = mean(Speed)),  
    by = "Distance",  
    suffix = c(" - kalenji", " - garmin")  
  )
```

Line indentations (should) matter

garmin dataset - practical 2 - solution

- 1 Round distances for both datasets *kalenji* and *garmin* to integers.
- 2 Calculate average *Speed* per *Distance*.
- 3 Join the tables returning *Distance*, *Time*, and *Average Speed* (km/h).

```
avg_speed
```

```
## # A tibble: 10 x 3
##   Distance `Average Speed - kalenji` `Average Speed - garmin`
##   <dbl>           <dbl>           <dbl>
## 1         5         10.8         10.2
## 2         6         11.0         10.9
## 3         8         11.2         10.5
## 4         9         10.8         9.68
## # ... with 6 more rows
```

garmin dataset - practical 2 - line indentations matter

```
avg_speed <- kalenji %>%  
  mutate(Distance = round(Distance)) %>%  
  group_by(Distance) %>%  
  summarize(`Average Speed` = mean(Speed)) %>%  
  full_join(  
    garmin %>%  
      mutate(Distance = round(Distance)) %>%  
      group_by(Distance) %>%  
      summarize(`Average Speed` = mean(Speed)),  
    by = "Distance",  
    suffix = c(" - kalenji", " - garmin")  
  )
```

garmin dataset - practical 2 - no consistent indentations = you're a monster

```
avg_speed <- kalenji %>%
  mutate(Distance = round(Distance)) %>% group_by(Distance) %>%
  summarize(
    `Average Speed` = mean(Speed)) %>%
  full_join(garmin %>%
    mutate(Distance = round(Distance)) %>% group_by(Distance) %>%
    summarize(`Average Speed` = mean(Speed)), by = "Distance",
    suffix = c(
      " - kalenji",
      " - garmin"
    ))
```

garmin dataset - practical 2 - how do you even write?

Nel mezzo
del
cammin di nostra vita, mi
ritrovai perunaselvaoscura
che
la
diritta
via era smarrita.

garmin dataset - practical 2 - writing and coding are not that different

Nel mezzo del cammin di nostra vita,
mi ritrovai per una selva oscura
che la diritta via era smarrita.

One line = one operation

One block = one coherent procedure

lines -> blocks -> pipeline

garmin dataset - practical 2 - considerations

- 1 Round distances for both datasets *kalenji* and *garmin* to integers.
- 2 Calculate average *Speed* per *Distance*.
- 3 Join the tables returning *Distance*, *Time*, and *Average Speed* (km/h).

All manipulations were done *on the fly*, i.e. without saving objects to memory unless specifically stated: `avg_speed <-` is the only new object. Old tables are still the same:

```
garmin$Distance #not rounded
```

```
## [1] 5.35 8.41 6.76 5.70 5.70 2.00 3.82 5.69 10.54 5.84 5.72
## [13] 5.67 9.52 6.01 5.81 5.74 6.03 5.78 7.04 6.06 5.83 5.84
## [25] 5.81 8.08 6.15 5.38 5.83 5.77 4.38 1.40 9.29 5.71 5.72
## [37] 6.07 7.02 6.53 6.61 8.42 10.49 6.04 5.96 6.97 8.42 6.01
## [49] 5.78 5.58 5.04 6.01 5.71 4.50 10.01 5.75 5.84 5.91 5.81 1
## [61] 6.00 6.07 6.21 6.05 5.66 5.71 7.89
```

garmin dataset - practical 2 - more considerations

Let's say we want to compare *Average Speed* between the two methods, e.g. `boxplot()`.

How would you manipulate this table?

```
avg_speed
```

```
## # A tibble: 10 x 3
##   Distance `Average Speed - kalenji` `Average Speed - garmin`
##   <dbl>          <dbl>          <dbl>
## 1         5         10.8         10.2
## 2         6         11.0         10.9
## 3         8         11.2         10.5
## 4         9         10.8         9.68
## # ... with 6 more rows
```

Tidy data: one row = one observation.

Goal: To achieve **tidy** data.

Pros:

- Very powerful: few commands to achieve tidy data.
- Extremely versatile.

Cons:

- Syntax can be counter-intuitive.

The most frequent use of *tidyr* is for *pivoting*: from long to wide and back again.

tidyr - long and wide tables

The most frequent use of *tidyr* is for *pivoting*: from long to wide and back again.

Long table

```
long <- tibble(  
  Group = LETTERS[1:3],  
  x = rnorm(3)  
)  
long
```

```
## # A tibble: 3 x 2  
##   Group      x  
##   <chr>  <dbl>  
## 1 A      0.138  
## 2 B     -0.731  
## 3 C     -0.915
```

Wide table

```
wide <- tibble(  
  A = rnorm(1),  
  B = rnorm(1),  
  C = rnorm(1)  
)  
wide
```

```
## # A tibble: 1 x 3  
##       A      B      C  
##   <dbl> <dbl> <dbl>  
## 1 -2.53 -1.62 -1.33
```

tidyr - wide to long

`pivot_longer()`: wide to long.

```
wide %>% pivot_longer(cols = everything(),  
                      names_to = "Group",  
                      values_to = "x")
```

```
## # A tibble: 3 x 2  
##   Group      x  
##   <chr> <dbl>  
## 1 A     -2.53  
## 2 B     -1.62  
## 3 C     -1.33
```

tidyr - wide to long and back again

`pivot_longer()`: wide to long.

```
wide %>% pivot_longer(cols = everything(),  
                      names_to = "Group",  
                      values_to = "x")
```

```
## # A tibble: 3 x 2  
##   Group      x  
##   <chr> <dbl>  
## 1 A     -2.53  
## 2 B     -1.62  
## 3 C     -1.33
```

`pivot_wider()`: long to wide.

```
long %>% pivot_wider(names_from = Group,  
                    values_from = x)
```

```
## # A tibble: 1 x 3  
##       A      B      C  
##   <dbl> <dbl> <dbl>  
## 1 0.138 -0.731 -0.915
```

tidyr - practical - problem

```
avg_speed
```

```
## # A tibble: 10 x 3
##   Distance `Average Speed - kalenji` `Average Speed - garmin`
##   <dbl>          <dbl>          <dbl>
## 1         5         10.8         10.2
## 2         6         11.0         10.9
## 3         8         11.2         10.5
## 4         9         10.8         9.68
## # ... with 6 more rows
```

- 1 Make `avg_speed` table longer: three columns *Distance*, *Speed*, and *Method*.
- 2 *Method* should have values *kalenji* or *garmin* only.
- 3 Boxplot: *Speed* ~ *Method*.

- 1 Make `avg_speed` table longer: three columns *Distance*, *Speed*, and *Method*.

```
avg_speed <- avg_speed %>%  
  pivot_longer(cols = contains("Speed"),  
               names_to = "Method",  
               values_to = "Speed")
```

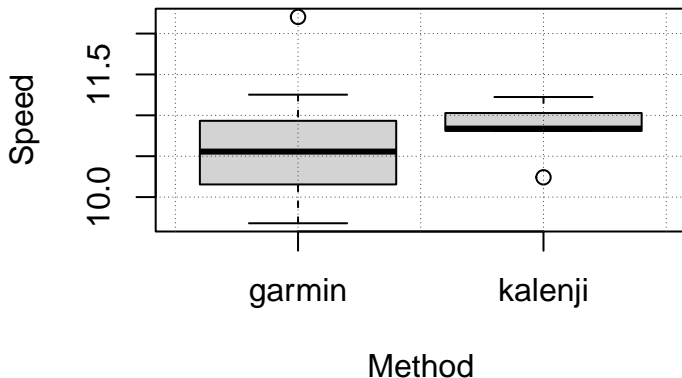

- 1 Make `avg_speed` table longer: three columns *Distance*, *Speed*, and *Method*.
- 2 *Method* should have values *kalenji* or *garmin* only.

```
avg_speed <- avg_speed %>%  
  mutate(Method = gsub("[:alpha:]]+ [:alpha:]]+ - ", "", Method))
```

tidyr - practical - solution

- 1 Make `avg_speed` table longer: three columns *Distance*, *Speed*, and *Method*.
- 2 *Method* should have values *kalenji* or *garmin* only.
- 3 Boxplot: *Speed* ~ *Method*.

```
boxplot(Speed ~ Method, data = avg_speed)  
grid(col = "grey20", lwd = .5)
```



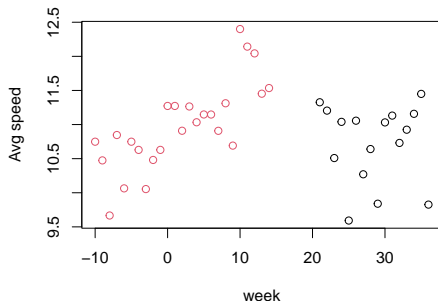
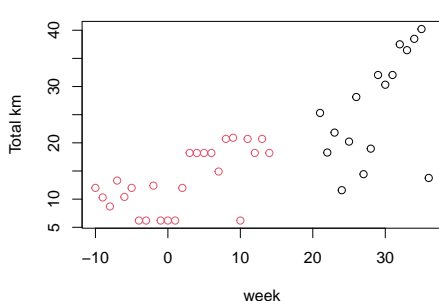
- 1 Plot total distance per week, coloring points according to sampling method.
- 2 Plot average speed per week, coloring points according to sampling method.

tidyverse - final practical - solution

```
d <- read_csv("kalenji.csv", show_col_types = FALSE) %>%
  mutate(date = as_date(date, format = "%m/%d/%y"),
         week = week(date),
         speed = km / ( hour(time) + minute(time) / 60 )) %>%
  group_by(week) %>%
  summarize(`Total km` = sum(km),
           `Avg speed` = mean(speed)) %>%
  mutate(method = "Kalenji") %>%
  bind_rows(
    read_csv("garmin.csv", show_col_types = FALSE) %>%
      filter(`Activity Type` == "Running") %>%
      mutate(Date = as_date(Date),
             week = week(Date),
             speed = Distance / ( hour(Time) + minute(Time) / 60 )) %>%
      group_by(week) %>%
      summarize(`Total km` = sum(Distance),
               `Avg speed` = mean(speed)) %>%
      mutate(method = "Garmin")
  ) %>%
  mutate(week = ifelse(week > 40, week - 53, week)) #these are last year we
```

tidyverse - final practical - plot

```
par(mfrow = c(1, 2))  
with(d, plot(week, `Total km`, col = as.factor(method)))  
with(d, plot(week, `Avg speed`, col = as.factor(method)))
```



I probably wasn't so accurate with the Kalenji watch.

The end

