Tidyverse

Qingyao Zhang

2025-10-20

The first task: recoding

- R help
- R search, e.g., https://search.r-project.org/
- AI

Data manipulation

- Recode values
- Computate row means and sums
- Compute column means and sums
- Subset and split

  Split one dataset to multiple datasets.
- combine and join

  Join multiple datasets into one datasets by rows or columns.
- aggregate

  Aggregate students datasets into class datasets.
- …

Hadley Wickham and his `tidyverse` family

Hadley Wickham, Chief Scientist at RStudio, is the recipient of the 2019 COPSS Presidents' Award. This award is presented annually to a young member of one of the participating societies of COPSS in recognition of outstanding contributions to the profession of statistics. The award citation recognized Wickham "for influential work in statistical computing, visualization, graphics, and data analysis; for developing and implementing an impressively comprehensive computational infrastructure for data analysis through R software; for making statistical thinking and computing accessible to large audience; and for enhancing an appreciation for the important role of statistics among data scientists."

`tidyverse` family

The 'tidyverse' is a set of packages that work in harmony because they share common data representations and 'API' design. This package is designed to make it easy to install and load multiple 'tidyverse' packages in a single step. Learn more about the 'tidyverse' at https://www.tidyverse.org.

- `tibble`, a tibble is a special data.frame.
- `dplyr`, `dplyr` is a grammar of data manipulation.
- `tidyr`, `tidyr` provides a set of functions that help you get to tidy data.
- `readr`, `readr` provides a fast and friendly way to read rectangular data (like csv, tsv, and fwf).
- `ggplot2`, `ggplot2` is a system for declaratively creating graphics.
- `forcats`, `forcats` provides a suite of useful tools that solve common problems with factors.
- `lubridate`, `lubridate` provides a set of functions for working with date-times.
- `stringr`, `stringr` provides a cohesive set of functions designed to make working with strings as easy as possible.
- `purrr`, `purrr` enhances R's functional programming (FP) toolkit by providing a complete and consistent set of tools for working with functions and vectors.

```
# install tidyverse if it is not installed
if (!requireNamespace("tidyverse", quietly = TRUE)) {
  install.packages("tidyverse")
}
# attach tidyverse
library(tidyverse)
## -- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
## v dplyr     1.1.4     v readr     2.1.5
## v forcats   1.0.1     v stringr   1.5.2
## v ggplot2   4.0.0     v tibble    3.3.0
```

```
## v lubridate 1.9.4     v tidyr     1.3.1
## v purrr     1.1.0
## -- Conflicts ---------------------------------------- tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
## i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to b
```

tibble

```
vignette("tibble")
```

Creating

- It never changes an input's type (i.e., no more stringsAsFactors = FALSE!).

```
head(data.frame(x = letters))
##   x
## 1 a
## 2 b
## 3 c
## 4 d
## 5 e
## 6 f
head(tibble(x = letters))
## # A tibble: 6 x 1
##   x
##   <chr>
## 1 a
## 2 b
## 3 c
## 4 d
## 5 e
## 6 f
```

This makes it easier to use with list-columns:

```
# data.frame()
data.frame(x = 1:3, y = list(1:5, 1:10, 1:20))
```

```
# Function I() inhibits the conversion of vectors in data.frame().
data.frame(x = 1:3, y = I(list(1:5, 1:10, 1:20)))
##   x            y
## 1 1 1, 2, 3,....
## 2 2 1, 2, 3,....
## 3 3 1, 2, 3,....
data.frame(x = 1:3, y = list(1:3, 4:6, 7:9))
##   x y.1.3 y.4.6 y.7.9
## 1 1     1     4     7
## 2 2     2     5     8
## 3 3     3     6     9

# tibble()
tibble(x = 1:3, y = list(1:5, 1:10, 1:20))
## # A tibble: 3 x 2
##       x y
##   <int> <list>
## 1     1 <int [5]>
## 2     2 <int [10]>
## 3     3 <int [20]>
```

List-columns are often created by `tidyr::nest()`, but they can be useful to create by hand.

- It never adjusts the names of variables:

```
names(data.frame(`crazy name` = 1))
## [1] "crazy.name"
names(tibble(`crazy name` = 1))
## [1] "crazy name"
```

- **It evaluates its arguments lazily and sequentially**:

```
tibble(x = 1:5, y = x ^ 2)
## # A tibble: 5 x 2
##       x     y
##   <int> <dbl>
## 1     1     1
## 2     2     4
## 3     3     9
## 4     4    16
## 5     5    25
```

It never uses row.names(). The whole point of tidy data is to store variables in a consistent way. So it never stores a variable as special attribute.

- It only recycles vectors of length 1. This is because recycling vectors of greater lengths is a frequent source of bugs.

Coercion

To complement tibble(), tibble provides as_tibble() to coerce objects into tibbles.

```
library(Keng)
data("depress")
depress_tbl <- as_tibble(depress)
class(depress)
## [1] "data.frame"
class(depress_tbl)
## [1] "tbl_df"     "tbl"        "data.frame"
```

Printing

```
head(depress[1:2])
##      id class
## 1 30107     3
## 2 30112     3
## 3 30116     3
## 4 30118     3
## 5 30122     3
## 6 30123     3
head(depress_tbl[1:2])
## # A tibble: 6 x 2
##      id class
##   <int> <int>
## 1 30107     3
## 2 30112     3
## 3 30116     3
## 4 30118     3
## 5 30122     3
## 6 30123     3
```

Subsetting

Tibbles are quite strict about subsetting. `[` always returns another tibble.

```
# data.frame
head(depress[1]) #return a data.frame
##      id
## 1 30107
## 2 30112
## 3 30116
## 4 30118
## 5 30122
## 6 30123
head(depress["id"]) #return a data.frame
##      id
## 1 30107
## 2 30112
## 3 30116
## 4 30118
## 5 30122
## 6 30123
depress[,1] #return a vector
##  [1]  30107  30112  30116  30118  30122  30123  30124  30125  30126  30127
## [11]  30128  30130  30131  30134  30135  30136  30138  30139  30140  30141
## [21]  30143  30144  30146  30148  30150  30151  30154  30156  30157  50209
## [31]  50211  50212  50217  50218  50219  50221  50223  50224  50226  50227
## [41]  50231  50235  50236  50237  50238  50239  50242  50245  50247  50249
## [51]  50251  50253  50256  50257  50258  90411  90415  90416  90418  90420
## [61]  90422  90423  90425  90428  90433  90434  90435  90436  90438  90439
## [71]  90441  90442  90443  90445 120518 120519 120520 120521 120524 120525
## [81] 120529 120532 120534 120536 120538 120539 120540 120542 120544 120549
## [91] 120551 120552 120553 121004
depress[[1]] #return a vector
##  [1]  30107  30112  30116  30118  30122  30123  30124  30125  30126  30127
## [11]  30128  30130  30131  30134  30135  30136  30138  30139  30140  30141
## [21]  30143  30144  30146  30148  30150  30151  30154  30156  30157  50209
## [31]  50211  50212  50217  50218  50219  50221  50223  50224  50226  50227
## [41]  50231  50235  50236  50237  50238  50239  50242  50245  50247  50249
## [51]  50251  50253  50256  50257  50258  90411  90415  90416  90418  90420
## [61]  90422  90423  90425  90428  90433  90434  90435  90436  90438  90439
## [71]  90441  90442  90443  90445 120518 120519 120520 120521 120524 120525
## [81] 120529 120532 120534 120536 120538 120539 120540 120542 120544 120549
## [91] 120551 120552 120553 121004
```

```
# tibble
depress_tbl[1] #return a tibble
## # A tibble: 94 x 1
##       id
##    <int>
##  1 30107
##  2 30112
##  3 30116
##  4 30118
##  5 30122
##  6 30123
##  7 30124
##  8 30125
##  9 30126
## 10 30127
## # i 84 more rows
depress_tbl[,1] #return a tibble
## # A tibble: 94 x 1
##       id
##    <int>
##  1 30107
##  2 30112
##  3 30116
##  4 30118
##  5 30122
##  6 30123
##  7 30124
##  8 30125
##  9 30126
## 10 30127
## # i 84 more rows
depress_tbl[[1]] #return a vector
##  [1]  30107  30112  30116  30118  30122  30123  30124  30125  30126  30127
## [11]  30128  30130  30131  30134  30135  30136  30138  30139  30140  30141
## [21]  30143  30144  30146  30148  30150  30151  30154  30156  30157  50209
## [31]  50211  50212  50217  50218  50219  50221  50223  50224  50226  50227
## [41]  50231  50235  50236  50237  50238  50239  50242  50245  50247  50249
## [51]  50251  50253  50256  50257  50258  90411  90415  90416  90418  90420
## [61]  90422  90423  90425  90428  90433  90434  90435  90436  90438  90439
## [71]  90441  90442  90443  90445 120518 120519 120520 120521 120524 120525
## [81] 120529 120532 120534 120536 120538 120539 120540 120542 120544 120549
## [91] 120551 120552 120553 121004
```

Tibbles never do **partial matching**.

```
depress$class
##  [1]  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3
## [26]  3  3  3  3  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5
## [51]  5  5  5  5  5  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9 12
## [76] 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12
depress$cla
##  [1]  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3
## [26]  3  3  3  3  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5
## [51]  5  5  5  5  5  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9 12
## [76] 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12
depress_tbl$cla
## Warning: Unknown or uninitialised column: `cla`.
## NULL
```

Recycling

When constructing a tibble, only values of length 1 are recycled.

```
# data.frame
data.frame(a = 1, b = 1:6)
##   a b
## 1 1 1
## 2 1 2
## 3 1 3
## 4 1 4
## 5 1 5
## 6 1 6
data.frame(a = 1:2, b = 1:6)
##   a b
## 1 1 1
## 2 2 2
## 3 1 3
## 4 2 4
## 5 1 5
## 6 2 6
# tibble
tibble(a = 1, b = 1:6)
## # A tibble: 6 x 2
```

```
##       a     b
##   <dbl> <int>
## 1     1     1
## 2     1     2
## 3     1     3
## 4     1     4
## 5     1     5
## 6     1     6
```

```
tibble(a = 1:2, b = 1:6)
```

Arithmetic operations

Unlike data frames, tibbles don't support **arithmetic operations on all columns**. The result is silently coerced to a data frame. Do not rely on this behavior, it may become an error in a forthcoming version.

```
df <- data.frame(a = 1:3, b = 4:6)
df + 1
##   a b
## 1 2 5
## 2 3 6
## 3 4 7
tbl <- as_tibble(df)
tbl * 2
##   a  b
## 1 2  8
## 2 4 10
## 3 6 12
```

dplyr

`dplyr` functions expect **tidy data** and work with **pipes**.

tidy data Each variable is in its own column, Each observation, or case, is in its own row.
pipes x |> f(y) becomes f(x,y)

To use functions in `dplyr`, attach `dplyr` or `tidyverse` firstly.

```
library(dplyr)
```

Learn

- Package Web, https://dplyr.tidyverse.org/
- Manual, `help("dplyr")`
- **Cheat Sheet**, https://www.rstudio.org/links/data_transformation_cheat_sheet, https://rstudio.github.io/cheatsheets/html/data-transformation.html

The functions in `dplyr` are grouped by their roles in the following:

Summarize Cases (Computation within columns)

Apply summary functions to columns to create a new table of **summary statistics**. Summary functions take vectors as input and return one value back.

`summarize()` computes table of summaries.

```
depress |>
  summarize(dm1_mean = mean(dm1),
            dm2_mean = mean(dm2),
            dm3_mean = mean(dm3),
            sample_size = n())
##   dm1_mean dm2_mean dm3_mean sample_size
## 1 1.907447 2.017021 2.012234          94
```

Summary Functions

`summarize()` applies summary functions to columns to create a new table. Summary functions take vectors as input and return single values as output.

Count

- `dplyr::n()`: number of values/rows
- `dplyr::n_distinct()`: # of uniques
- `sum(!is.na())`: # of non-NAs
- `count()`

Count number of rows in each group defined by the variables in ….. Also `tally()`, `add_count()`, and `add_tally()`.

```
depress |> count(intervention)
##   intervention  n
## 1            0 45
## 2            1 49
depress |> count(intervention, class)
##   intervention class  n
## 1            0     5 26
## 2            0     9 19
## 3            1     3 29
## 4            1    12 20
depress |> count(intervention, class, gender)
##   intervention class gender  n
## 1            0     5      0 12
## 2            0     5      1 14
## 3            0     9      0  7
## 4            0     9      1 12
## 5            1     3      0 12
## 6            1     3      1 17
## 7            1    12      0 10
## 8            1    12      1 10
depress |> tally()
##    n
## 1 94
```

Position

- `mean()`: mean, also `mean(!is.na())`
- `median()`: median

Logical

- `any()`
- `all()`
- `mean()`: proportion of TRUEs
- `sum()`: # of TRUEs

Order

- `dplyr::first()`: first value
- `dplyr::last()`: last value
- `dplyr::nth()`: value in the nth location of vector

Rank

- `quantile()`: nth quantile
- `min()`: minimum value
- `max()`: maximum value

Spread

- `IQR()`: Inter-Quartile Range
- `mad()`: median absolute deviation
- `sd()`: standard deviation
- `var()`: variance

Group Cases

`group_by()`

`group_by()` creates a "grouped" copy of a table grouped by columns in ….. dplyr functions then will manipulate each "group" separately and combine the results.

```
depress |>
  group_by(intervention) |>
  summarize(dm1_mean = mean(dm1),
            dm2_mean = mean(dm2),
            dm3_mean = mean(dm3),
            sample_size = n())
## # A tibble: 2 x 5
##   intervention dm1_mean dm2_mean dm3_mean sample_size
##          <int>    <dbl>    <dbl>    <dbl>       <int>
## 1            0     1.98     2.14     2.12          45
## 2            1     1.84     1.90     1.92          49
```

Alternate grouping syntax with `.by` as an argument:

```
depress |>
  summarize(dm1_mean = mean(dm1),
            dm2_mean = mean(dm2),
            dm3_mean = mean(dm3),
            sample_size = n(),
            .by = intervention)
##   intervention dm1_mean dm2_mean dm3_mean sample_size
## 1            1 1.842857 1.903061 1.915306          49
## 2            0 1.977778 2.141111 2.117778          45
```

`rowwise()`

`rowwise()` groups data into individual rows. dplyr functions will compute results for each row.

```
depress |>
  rowwise() |>
  summarize(size = n()) |>
  table()
## size
##  1
## 94
```

`ungroup()`

`ungroup()` Returns ungrouped copy of table.

Manipulate Cases

Extract Cases

Row functions return a subset of rows as a new table.

`filter()`

`filter()` extracts rows that meet logical criteria.

```
depress |>
  filter(gender == 0) |>
  nrow()
## [1] 41
```

Logical and boolean operations to use with `filter()`:

`&`, `|`, `!`, `xor()`, `==`, `!=`, `<`, `<=`, `>`, `>=`, `is.na()`, `!is.na()`, `%in%`.

```
depress |>
  filter(class %in% c(5, 9)) |>
  nrow()
## [1] 45
```

See `?base::Logic` and `?Comparison` for help.

`distinct()`

`distinct()` removes rows with duplicate values.

```
depress |>
  distinct(id) |>
  nrow()
## [1] 94
```

`slice()`

`slice()` selects rows by position.

```
depress[1:5] |>
  slice(1:6)
##      id class grade elite intervention
## 1 30107     3     1     1            1
## 2 30112     3     1     1            1
## 3 30116     3     1     1            1
## 4 30118     3     1     1            1
## 5 30122     3     1     1            1
## 6 30123     3     1     1            1
```

`slice()` related functions:

- `slice_head()` and `slice_tail()` select the first or last rows..
- `slice_min()` and `slice_max()` select rows with the smallest or largest values of a variable.
- `slice_sample()` randomly selects rows.

`arrange()`

`arrange()` arranges rows by values of a column or columns (low to high), use with `desc()` to order from high to low.

```
depress[1:7] |>
  arrange(age) |>
  head()
##       id class grade elite intervention gender age
## 1 120532    12     1     0            1      1  14
## 2  30112     3     1     1            1      1  15
## 3  30123     3     1     1            1      0  15
## 4  30128     3     1     1            1      0  15
## 5  30150     3     1     1            1      1  15
## 6  50211     5     1     1            0      0  15

depress[1:7] |>
  arrange(desc(age)) |>
  head()
##      id class grade elite intervention gender age
## 1 30107     3     1     1            1      0  17
## 2 30118     3     1     1            1      0  17
## 3 30126     3     1     1            1      1  17
## 4 30127     3     1     1            1      1  17
## 5 30131     3     1     1            1      1  17
## 6 30139     3     1     1            1      0  17
```

Manipulate Variables

Column functions return a set of columns as a new vector or table.

Extract variables

`pull()`

`pull()` extracts column values as a vector, by name or index.

```
depress$age
##  [1] 17 15 16 17 16 15 16 16 17 17 15 16 17 16 16 16 16 17 17 17 17 16 16 17 15
## [26] 17 17 17 16 16 15 16 16 16 16 16 16 17 16 15 16 16 16 17 16 16 17 17 16 17
## [51] 17 15 15 16 17 15 15 15 16 16 16 16 15 15 15 16 15 17 16 15 16 15 15 16 17
```

```
## [76] 15 15 15 15 15 15 14 15 16 15 16 16 15 16 16 16 16 16 15
depress |>
  pull(age)
##  [1] 17 15 16 17 16 15 16 16 17 17 15 16 17 16 16 16 16 17 17 17 17 16 16 17 15
## [26] 17 17 17 16 16 15 16 16 16 16 16 16 17 16 15 16 16 16 17 16 16 17 17 16 17
## [51] 17 15 15 16 17 15 15 15 16 16 16 16 15 15 15 16 15 17 16 15 16 15 15 16 17
## [76] 15 15 15 15 15 15 14 15 16 15 16 16 15 16 16 16 16 16 15
```

`select()`

`select()` extracts columns as a table.

```
depress |>
  select(ecr1avo, ecr4avo, ecr6avo, ecr9avo) |>
  psych::corr.test()
## Call:psych::corr.test(x = select(depress, ecr1avo, ecr4avo, ecr6avo,
##     ecr9avo))
## Correlation matrix
##         ecr1avo ecr4avo ecr6avo ecr9avo
## ecr1avo    1.00    0.09    0.03    0.21
## ecr4avo    0.09    1.00    0.43    0.44
## ecr6avo    0.03    0.43    1.00    0.28
## ecr9avo    0.21    0.44    0.28    1.00
## Sample Size
## [1] 94
## Probability values (Entries above the diagonal are adjusted for multiple tests.)
##         ecr1avo ecr4avo ecr6avo ecr9avo
## ecr1avo    0.00    0.74    0.80    0.12
## ecr4avo    0.37    0.00    0.00    0.00
## ecr6avo    0.80    0.00    0.00    0.02
## ecr9avo    0.04    0.00    0.01    0.00
##
##  To see confidence intervals of the correlations, print with the short=FALSE option
```

Tidyverse selections implement a dialect of R where operators make it easy to select variables:

- `:` for selecting a range of consecutive variables.
- `!` for taking the complement of a set of variables.
- `&` and `|` for selecting the intersection or the union of two sets of variables.
- `c()` for combining selections.

In addition, you can use selection helpers. Some helpers select specific columns:

- `everything()`: Matches all variables.
- `last_col()`: Select last variable, possibly with an offset.
- `group_cols()`: Select all grouping columns with `group_by()`.

Other helpers select variables by matching patterns in their names:

- `starts_with()`: Starts with a prefix.
- `ends_with()`: Ends with a suffix.
- `contains()`: Contains a literal string.
- `matches()`: Matches a regular expression.
- `num_range()`: Matches a numerical range like x01, x02, x03.

Or from variables stored in **a character vector**:

- `all_of()`: Matches variable names in a character vector. All names must be present, otherwise an out-of-bounds error is thrown.
- `any_of()`: Same as `all_of()`, except that no error is thrown for names that don't exist.

Or using a predicate function:

- `where()`: Applies a function to all variables and selects those for which the function returns TRUE.

`relocate()`

`relocate()` moves columns to new position.

```
names(depress)[1:5]
## [1] "id"           "class"        "grade"        "elite"        "intervention"
depress |>
  select(1:5) |>
  relocate(grade, .before = class) |>
  names()
## [1] "id"           "grade"        "class"        "elite"        "intervention"
```

Make New Variables

Apply vectorized functions to columns. Vectorized functions take vectors as input and return vectors of the same length as output.

`mutate()`

`mutate()` computes new variable(s) or column(s).

```
depress |>
  mutate(depression_mean_t1 = rowMeans(select(depress,starts_with("depr1i"))),
         depression_mean_t2 = rowMeans(select(depress,starts_with("depr2i"))),
         depression_mean_t3 = rowMeans(select(depress,starts_with("depr3i")))) |>
  select(depression_mean_t1, depression_mean_t2, depression_mean_t3) |>
  head()
##   depression_mean_t1 depression_mean_t2 depression_mean_t3
## 1               1.55               1.80               2.00
## 2               1.55               1.85               2.25
## 3               1.90               1.65               1.65
## 4               1.35               1.65               1.60
## 5               1.30               2.25               2.10
## 6               1.95               1.60               1.65
```

Vectorized Functions

`mutate()` applies vectorized functions to columns to create new columns. Vectorized functions take vectors as input and return vectors of the same length as output.

Offset

- `dplyr::lag()`: offset elements by 1
- `dplyr::lead()`: offset elements by -1

Cumulative Aggregate

- `dplyr::cumall()`: cumulative all()
- `dply::cumany()`: cumulative any()
- `cummax()`: cumulative max()
- `dplyr::cummean()`: cumulative mean()
- `cummin()`: cumulative min()
- `cumprod()`: cumulative prod()
- `cumsum()`: cumulative sum()

Ranking

- `dplyr::cume_dist()`: proportion of all values `<=`
- `dplyr::dense_rank()`: rank with ties = min, no gaps
- `dplyr::min_rank()`: rank with ties = min
- `dplyr::ntile()`: bins into n bins
- `dplyr::percent_rank()`: min_rank() scaled to [0,1]

- `dplyr::row_number()`: rank with ties = "first"

Math

- `+`, `-`, `*`, `/`, `^`,`%/%`, `%%`: arithmetic ops
- `log()`, `log2()`, `log10()`: logs
- `<`, `<=`, `>`, `>=`, `!=`, `==`: logical comparisons
- `dplyr::between()`: x >= left & x <= right
- `dplyr::near()`: safe `==` for floating point numbers

Miscellaneous

- `dplyr::case_when()`: multi-case `if_else()`. `case_when()` could be used to recode variables.

```
depress |>
  mutate(cope1i1p_new = case_when(
    cope1i1p == 1 ~ 5,
    cope1i1p == 2 ~ 4,
    cope1i1p == 3 ~ 3,
    cope1i1p == 4 ~ 2,
    cope1i1p == 5 ~ NA,
    .default = 999
  ),
  cope1i1p_new2 = case_when(
    cope1i1p_new == 1 ~ 5,
    cope1i1p_new == 2 ~ 4,
    cope1i1p_new == 3 ~ 3,
    cope1i1p_new == 4 ~ 2,
    cope1i1p_new == 5 ~ 1,
    is.na(cope1i1p_new) ~ 999
  )) |>
  select(cope1i1p, cope1i1p_new, cope1i1p_new2) |>
  head()
##   cope1i1p cope1i1p_new cope1i1p_new2
## 1        5           NA           999
## 2        4            2             4
## 3        4            2             4
## 4        3            3             3
## 5        3            3             3
## 6        3            3             3
```

- `dplyr::coalesce()`: first non-NA values by element across a set of vectors
- `dplyr::if_else()`: element-wise if() + else()

- `dplyr::na_if()`: replace specific values with `NA`
- `pmax()`: element-wise `max()`
- `pmin()`: element-wise `min()`

`rename()`

`rename()` renames columns. Use `rename_with()` to rename with a string function.

Manipulate Multiple Variables at Once

`across()`

`across()` summarizes or mutate multiple columns in the same way.

```
# summarize 1 variable
depress |>
  select(starts_with("dm")) |>
  summarize(mean(dm1))
##   mean(dm1)
## 1  1.907447
depress |>
  select(starts_with("dm")) |>
  summarize(across(everything(), mean))
##        dm1      dm2      dm3
## 1 1.907447 2.017021 2.012234
```

`c_across()`

`c_across()` computes across columns in row-wise data.

```
depress |>
  rowwise() |>
  mutate(anx2 = mean(c_across(ends_with("anx")))) |>
  select(anx, anx2) |>
  slice_head(n = 6)
## # A tibble: 94 x 2
## # Rowwise:
##      anx  anx2
##    <dbl> <dbl>
##  1  1     1
##  2  2.67  2.67
```

```
##  3  1     1
##  4  2.33  2.33
##  5  1     1
##  6  1     1
##  7  1.33  1.33
##  8  1     1
##  9  1     1
## 10  1.67  1.67
## # i 84 more rows
```

Row Names

Tidy data does not use rownames, which store a variable outside of the columns. To work with the rownames, first move them into a column.

- `tibble::rownames_to_column()`: Move row names into col
- `tibble::columns_to_rownames()`: Move col into row names.
- `tibble::has_rownames()`
- `tibble::remove_rownames()`

Combine Tables

```
x <- tribble(
   ~A,  ~B, ~C,
  "a", "t",  1,
  "b", "u",  2,
  "c", "v",  3
)

y <- tribble(
   ~A,  ~B, ~D,
  "a", "t",  3,
  "b", "u",  2,
  "d", "w",  1
)
```

Combine Variables

`bind_cols()` returns tables placed side by side as a single table. Column lengths must be equal. Columns will NOT be matched by id (to do that look at Relational Data below), so be sure to check that both tables are ordered the way you want before binding.

Combine Cases

`bind_rows(..., .id = NULL)`: Returns tables one on top of the other as a single table. Set `.id` to a column name to add a column of the original table names.

Relational Data

Use a "Mutating Join" to join one table to columns from another, matching values with the rows that the correspond to. Each join retains a different combination of values from the tables.

- `left_join(x, y, by = NULL)`: Join matching values from y to x.
- `right_join(x, y, by = NULL)`: Join matching values from x to y.
- `inner_join(x, y, by = NULL)`: Join data. retain only rows with matches.
- `full_join(x, y, by = NULL)`: Join data. Retain all values, all rows.

Use a "Filtering Join" to filter one table against the rows of another.

- `semi_join(x, y, by = NULL)`: Return rows of x that have a match in y. Use to see what will be included in a join.
- `anti_join(x, y, by = NULL)`: Return rows of x that do not have a match in y. Use to see what will not be included in a join.

Use a "Nest Join" to inner join one table to another into a nested data frame.

`nest_join(x, y, by = NULL)`: Join data, nesting matches from y in a single new data frame column.

Column Matching for Joins

Use `by = join_by(col1, col2, …)` to specify one or more common columns to match on.

`left_join(x, y, by = join_by(A))` `left_join(x, y, by = join_by(A, B))`

Use a logical statement, `by = join_by(col1 == col2)`, to match on columns that have different names in each table.

```
left_join(x, y, by = join_by(C == D))
```

Use suffix to specify the suffix to give to unmatched columns that have the same name in both tables.

```
left_join(x, y, by = join_by(C == D), suffix = c("1", "2"))
```

Set Operations

- `intersect(x, y, ...)`: Rows that appear in both x and y.
- `setdiff(x, y, ...)`: Rows that appear in x but not y.
- `union(x, y, ...)`: Rows that appear in x or y, duplicates removed. union_all() retains duplicates.
- `setequal()`: test whether two data sets contain the exact same rows (in any order).

`tidyr`

Learn

- Package Web, https://tidyr.tidyverse.org/
- Manual, `help("tidyr")`
- Cheat Sheet, https://posit.co/resources/cheatsheets/, https://rstudio.github.io/cheatsheets/html/tidyr.html

Reshape Data

Pivot data to reorganize values into a new layout.

`pivot_longer()`

`pivot_longer()` lengthens data by collapsing several columns into two.

For demonstrating purpose, `tidyr` has internal example tables including `table4a` which looks like the following:

```
table4a
## # A tibble: 3 x 3
##   country     `1999` `2000`
##   <chr>        <dbl>  <dbl>
## 1 Afghanistan    745   2666
## 2 Brazil       37737  80488
## 3 China       212258 213766
names(table4a)
## [1] "country" "1999"    "2000"
```

Column names move to a new `names_to` column and values to a new `values_to` column. The output of `pivot_longer()` will look like the following:

```
pivot_longer(table4a, cols = 2:3, names_to = "year", values_to = "cases")
## # A tibble: 6 x 3
##   country     year   cases
##   <chr>       <chr>  <dbl>
## 1 Afghanistan 1999     745
## 2 Afghanistan 2000    2666
## 3 Brazil      1999   37737
## 4 Brazil      2000   80488
## 5 China       1999  212258
## 6 China       2000  213766
```

`pivot_wider()`

`pivot_wider()` is the inverse of `pivot_longer()`, `pivot_wider()` Widens data by expanding two columns into several.

The initial `table2` looks like the following:

```
table2
## # A tibble: 12 x 4
##    country      year type            count
##    <chr>       <dbl> <chr>           <dbl>
##  1 Afghanistan  1999 cases             745
##  2 Afghanistan  1999 population   19987071
##  3 Afghanistan  2000 cases            2666
##  4 Afghanistan  2000 population   20595360
##  5 Brazil       1999 cases           37737
##  6 Brazil       1999 population  172006362
##  7 Brazil       2000 cases           80488
##  8 Brazil       2000 population  174504898
##  9 China        1999 cases          212258
## 10 China        1999 population 1272915272
## 11 China        2000 cases          213766
## 12 China        2000 population 1280428583
```

One column provides the new column names, the other the values. The output of `pivot_wider()` will look like the following:

```
pivot_wider(table2, names_from = type, values_from = count)
## # A tibble: 6 x 4
##   country      year  cases population
##   <chr>       <dbl>  <dbl>      <dbl>
```

```
## 1 Afghanistan  1999    745   19987071
## 2 Afghanistan  2000   2666   20595360
## 3 Brazil       1999  37737  172006362
## 4 Brazil       2000  80488  174504898
## 5 China        1999 212258 1272915272
## 6 China        2000 213766 1280428583
```

Split Cells

Use these functions to split or combine cells into individual, isolated values.

`unite()`

`unite()` collapses cells across several columns into a single column.

The initial table5 looks like the following:

```
table5
## # A tibble: 6 x 4
##   country     century year  rate
##   <chr>       <chr>   <chr> <chr>
## 1 Afghanistan 19      99    745/19987071
## 2 Afghanistan 20      00    2666/20595360
## 3 Brazil      19      99    37737/172006362
## 4 Brazil      20      00    80488/174504898
## 5 China       19      99    212258/1272915272
## 6 China       20      00    213766/1280428583
```

The output of `unite()` will look like the following:

```
unite(table5, century, year, col = "year", sep = "")
## # A tibble: 6 x 3
##   country     year  rate
##   <chr>       <chr> <chr>
## 1 Afghanistan 1999  745/19987071
## 2 Afghanistan 2000  2666/20595360
## 3 Brazil      1999  37737/172006362
## 4 Brazil      2000  80488/174504898
## 5 China       1999  212258/1272915272
## 6 China       2000  213766/1280428583
```

`separate_wider_delim()`

`separate_wider_delim()` separates each cell in a column into several columns.

Also `extract()`.

The initial table3 looks like the following:

```
table3
## # A tibble: 6 x 3
##   country      year rate
##   <chr>       <dbl> <chr>
## 1 Afghanistan  1999 745/19987071
## 2 Afghanistan  2000 2666/20595360
## 3 Brazil       1999 37737/172006362
## 4 Brazil       2000 80488/174504898
## 5 China        1999 212258/12729152 72
## 6 China        2000 213766/1280428583
```

The output of `separate_wider_delim()` will look like the following:

```
separate_wider_delim(table3, rate, delim = "/", names = c("cases", "pop"))
## # A tibble: 6 x 4
##   country      year cases  pop
##   <chr>       <dbl> <chr>  <chr>
## 1 Afghanistan  1999 745    19987071
## 2 Afghanistan  2000 2666   20595360
## 3 Brazil       1999 37737  172006362
## 4 Brazil       2000 80488  174504898
## 5 China        1999 212258 12729152 72
## 6 China        2000 213766 1280428583
```

`separate_longer_delim()` separates each cell in a column into several rows.

The initial table3 looks like the following:

```
table3
## # A tibble: 6 x 3
##   country      year rate
##   <chr>       <dbl> <chr>
## 1 Afghanistan  1999 745/19987071
## 2 Afghanistan  2000 2666/20595360
## 3 Brazil       1999 37737/172006362
```

```
## 4 Brazil       2000 80488/174504898
## 5 China        1999 212258/1272915272
## 6 China        2000 213766/1280428583
```

The output of `separate_longer_delim()` will look like the following:

```
separate_longer_delim(table3, rate, delim = "/")
## # A tibble: 12 x 3
##    country      year rate
##    <chr>       <dbl> <chr>
##  1 Afghanistan  1999 745
##  2 Afghanistan  1999 19987071
##  3 Afghanistan  2000 2666
##  4 Afghanistan  2000 20595360
##  5 Brazil       1999 37737
##  6 Brazil       1999 172006362
##  7 Brazil       2000 80488
##  8 Brazil       2000 174504898
##  9 China        1999 212258
## 10 China        1999 1272915272
## 11 China        2000 213766
## 12 China        2000 1280428583
```

Expand Tables

Create new combinations of variables or identify implicit missing values (combinations of variables not present in the data).

`expand()`

`expand()` creates a new tibble with all possible combinations of the values of the variables listed in … Drop other variables.

```
expand(mtcars, cyl, gear, carb)
## # A tibble: 54 x 3
##      cyl  gear  carb
##    <dbl> <dbl> <dbl>
##  1     4     3     1
##  2     4     3     2
##  3     4     3     3
```

```
##  4     4     3     4
##  5     4     3     6
##  6     4     3     8
##  7     4     4     1
##  8     4     4     2
##  9     4     4     3
## 10     4     4     4
## # i 44 more rows
```

`complete()`

`complete()` adds missing possible combinations of values of variables listed in … Fill remaining variables with `NA`.

```
complete(mtcars, cyl, gear, carb)
## # A tibble: 74 x 11
##      cyl  gear  carb   mpg  disp    hp  drat    wt  qsec    vs    am
##    <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
##  1     4     3     1  21.5 120.     97  3.7   2.46  20.0     1     0
##  2     4     3     2  NA    NA      NA NA    NA     NA      NA    NA
##  3     4     3     3  NA    NA      NA NA    NA     NA      NA    NA
##  4     4     3     4  NA    NA      NA NA    NA     NA      NA    NA
##  5     4     3     6  NA    NA      NA NA    NA     NA      NA    NA
##  6     4     3     8  NA    NA      NA NA    NA     NA      NA    NA
##  7     4     4     1  22.8 108      93  3.85  2.32  18.6     1     1
##  8     4     4     1  32.4  78.7    66  4.08  2.2   19.5     1     1
##  9     4     4     1  33.9  71.1    65  4.22  1.84  19.9     1     1
## 10     4     4     1  27.3  79      66  4.08  1.94  18.9     1     1
## # i 64 more rows
```

Handle Missing Values

Drop or replace explicit missing values (`NA`).

```
x <- tribble(
  ~x1, ~x2,
  "A",   1,
  "B",  NA,
  "C",  NA,
  "D",   3,
```

```
  "E",  NA
)
```

drop_na()

`drop_na()` drops rows containing `NA`s in … columns.

```
drop_na(x, x2)
## # A tibble: 2 x 2
##   x1       x2
##   <chr> <dbl>
## 1 A         1
## 2 D         3
```

fill()

`fill()` fills in `NA`s in … columns using the next or previous value (is this meaningful?).

```
fill(x, x2)
## # A tibble: 5 x 2
##   x1       x2
##   <chr> <dbl>
## 1 A         1
## 2 B         1
## 3 C         1
## 4 D         3
## 5 E         3
```

replace_na()

`replace_na()` specifies a value to replace `NA` in selected columns.

```
replace_na(x, list(x2 = 2))
## # A tibble: 5 x 2
##   x1       x2
##   <chr> <dbl>
## 1 A         1
## 2 B         2
## 3 C         2
## 4 D         3
## 5 E         2
```

Nested Data

A nested data frame stores individual tables as a list-column of data frames within a larger organizing data frame. List-columns can also be lists of vectors or lists of varying data types. Use a nested data frame to:

- Preserve relationships between observations and subsets of data. Preserve the type of the variables being nested (factors and datetimes aren't coerced to character).
- Manipulate many sub-tables are once with `purrr` functions like `map()`, `map2()`, or `pmap()` or with `dplyr::rowwise()` grouping.

Create Nested Data

`nest()` moves groups of cells into a list-column of a data frame. Use alone or with `dplyr::group_by()`.

Group the data frame with `group_by()` and use `nest()` to move the groups into a list-column.

```
n_storms <- storms |>
  group_by(name) |>
  nest()
```

Use `nest(new_col = c(x,y))` to specify the columns to group using `dplyr::select()` syntax.

```
n_storms <- storms |>
  nest(data = c(year:long))
```

Index list-columns with `[[]]`.

```
n_storms$data[1]
## [[1]]
## # A tibble: 4 x 6
##    year month   day  hour   lat  long
##   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
## 1  1975     6    27     0  27.5   -79
## 2  1975     6    27     6  28.5   -79
## 3  1975     6    27    12  29.5   -79
## 4  1975     6    27    18  30.5   -79
n_storms$data[[1]]
```

```
## # A tibble: 4 x 6
##    year month   day  hour   lat  long
##   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
## 1  1975     6    27     0  27.5   -79
## 2  1975     6    27     6  28.5   -79
## 3  1975     6    27    12  29.5   -79
## 4  1975     6    27    18  30.5   -79
```

Create Tibbles With List-Columns

`tibble::tribble()`

`tibble::tribble()` makes list-columns when needed.

```
tribble(
  ~max, ~seq,
     3,  1:3,
     4,  1:4,
     5,  1:5
)
## # A tibble: 3 x 2
##     max seq
##   <dbl> <list>
## 1     3 <int [3]>
## 2     4 <int [4]>
## 3     5 <int [5]>
```

`tibble::tibble()` saves list input as list-columns.

```
tibble(
  max = c(3,4,5),
  seq = list(1:3, 1:4, 1:5)
)
## # A tibble: 3 x 2
##     max seq
##   <dbl> <list>
## 1     3 <int [3]>
## 2     4 <int [4]>
## 3     5 <int [5]>
```

`tibble::enframe()` converts multi-level list to a tibble with list-cols.

```
enframe(list("3" = 1:3, "4" = 1:4, "5" = 1:5), "max", "seq")
## # A tibble: 3 x 2
##   max   seq
##   <chr> <list>
## 1 3     <int [3]>
## 2 4     <int [4]>
## 3 5     <int [5]>
```

Output List-Columns From Other Functions

`dplyr::mutate()`, `transmute()`, and `summarise()` will output list-columns if they return a list.

```
mtcars |>
  group_by(cyl) |>
  summarise(q = list(quantile(mpg)))
## # A tibble: 3 x 2
##     cyl q
##   <dbl> <list>
## 1     4 <dbl [5]>
## 2     6 <dbl [5]>
## 3     8 <dbl [5]>
```

Reshape Nested Data

`unnest()`

`unnest()` flatten nested columns back to regular columns. The inverse of `nest()`.

```
n_storms |> unnest(data)
## # A tibble: 19,537 x 13
##    name  status              category  wind pressure tropicalstorm_force_diame~1
##    <chr> <fct>                  <dbl> <int>    <int>                        <int>
##  1 Amy   tropical depression       NA    25     1013                           NA
##  2 Amy   tropical depression       NA    25     1013                           NA
##  3 Amy   tropical depression       NA    25     1013                           NA
##  4 Amy   tropical depression       NA    25     1013                           NA
##  5 Amy   tropical depression       NA    25     1012                           NA
##  6 Amy   tropical depression       NA    25     1012                           NA
```

```
##  7 Amy   tropical depression       NA    25     1011                          NA
##  8 Amy   tropical depression       NA    30     1006                          NA
##  9 Amy   tropical storm            NA    35     1004                          NA
## 10 Amy   tropical storm            NA    40     1002                          NA
## # i 19,527 more rows
## # i abbreviated name: 1: tropicalstorm_force_diameter
## # i 7 more variables: hurricane_force_diameter <int>, year <dbl>, month <dbl>,
## #   day <int>, hour <dbl>, lat <dbl>, long <dbl>
```

unnest_longer()

`unnest_longer()` turns each element of a list-column into a row.

```
starwars |>
  select(name, films) |>
  unnest_longer(films)
## # A tibble: 173 x 2
##    name           films
##    <chr>          <chr>
##  1 Luke Skywalker A New Hope
##  2 Luke Skywalker The Empire Strikes Back
##  3 Luke Skywalker Return of the Jedi
##  4 Luke Skywalker Revenge of the Sith
##  5 Luke Skywalker The Force Awakens
##  6 C-3PO          A New Hope
##  7 C-3PO          The Empire Strikes Back
##  8 C-3PO          Return of the Jedi
##  9 C-3PO          The Phantom Menace
## 10 C-3PO          Attack of the Clones
## # i 163 more rows
```

unnest_wider()

`unnest_wider()` turns each element of a list-column into a regular column.

```
starwars |>
  select(name, films) |>
  unnest_wider(films, names_sep = "_")
## # A tibble: 87 x 8
##    name               films_1    films_2 films_3 films_4 films_5 films_6 films_7
##    <chr>              <chr>      <chr>   <chr>   <chr>   <chr>   <chr>   <chr>
```

```
##  1 Luke Skywalker     A New Hope The Em~ Return~ Reveng~ The Fo~ <NA>     <NA>
##  2 C-3PO              A New Hope The Em~ Return~ The Ph~ Attack~ Reveng~ <NA>
##  3 R2-D2              A New Hope The Em~ Return~ The Ph~ Attack~ Reveng~ The Fo~
##  4 Darth Vader        A New Hope The Em~ Return~ Reveng~ <NA>     <NA>     <NA>
##  5 Leia Organa        A New Hope The Em~ Return~ Reveng~ The Fo~ <NA>     <NA>
##  6 Owen Lars          A New Hope Attack~ Reveng~ <NA>     <NA>     <NA>     <NA>
##  7 Beru Whitesun Lars A New Hope Attack~ Reveng~ <NA>     <NA>     <NA>     <NA>
##  8 R5-D4              A New Hope <NA>    <NA>     <NA>     <NA>     <NA>     <NA>
##  9 Biggs Darklighter  A New Hope <NA>    <NA>     <NA>     <NA>     <NA>     <NA>
## 10 Obi-Wan Kenobi     A New Hope The Em~ Return~ The Ph~ Attack~ Reveng~ <NA>
## # i 77 more rows
```

`hoist()`

`hoist()` selectively pulls list components out into their own top-level columns. Uses `purrr::pluck()` syntax for selecting from lists.

```
starwars |>
  select(name, films) |>
  hoist(films, first_film = 1, second_film = 2)
## # A tibble: 87 x 4
##    name               first_film second_film             films
##    <chr>              <chr>      <chr>                   <list>
##  1 Luke Skywalker     A New Hope The Empire Strikes Back <chr [3]>
##  2 C-3PO              A New Hope The Empire Strikes Back <chr [4]>
##  3 R2-D2              A New Hope The Empire Strikes Back <chr [5]>
##  4 Darth Vader        A New Hope The Empire Strikes Back <chr [2]>
##  5 Leia Organa        A New Hope The Empire Strikes Back <chr [3]>
##  6 Owen Lars          A New Hope Attack of the Clones    <chr [1]>
##  7 Beru Whitesun Lars A New Hope Attack of the Clones    <chr [1]>
##  8 R5-D4              A New Hope <NA>                    <chr [0]>
##  9 Biggs Darklighter  A New Hope <NA>                    <chr [0]>
## 10 Obi-Wan Kenobi     A New Hope The Empire Strikes Back <chr [4]>
## # i 77 more rows
```

Transform Nested Data

A vectorized function takes a vector, transforms each element in parallel, and returns a vector of the same length. By themselves vectorized functions cannot work with lists, such as list-columns.

`dplyr::rowwise()` groups data so that each row is one group, and within the groups, elements of list-columns appear directly (accessed with `[[`), not as lists of length one. When you use `rowwise()`, `dplyr` functions will seem to apply functions to list-columns in a vectorized fashion.

Apply a function to a list-column and create a new list-column. In this example, `dim()` returns two values per row and so is wrapped with `list()` to tell `mutate()` to create a list-column.

```
n_storms |>
  rowwise() |>
  mutate(n = list(dim(data))) # dim() returns two values per row, wrap with list to tell muta
## # A tibble: 14,075 x 9
## # Rowwise:
##    name  status              category  wind pressure tropicalstorm_force_diame~1
##    <chr> <fct>                  <dbl> <int>    <int>                       <int>
##  1 Amy   tropical depression       NA    25     1013                          NA
##  2 Amy   tropical depression       NA    25     1012                          NA
##  3 Amy   tropical depression       NA    25     1011                          NA
##  4 Amy   tropical depression       NA    30     1006                          NA
##  5 Amy   tropical storm            NA    35     1004                          NA
##  6 Amy   tropical storm            NA    40     1002                          NA
##  7 Amy   tropical storm            NA    45     1000                          NA
##  8 Amy   tropical storm            NA    50      998                          NA
##  9 Amy   tropical storm            NA    55      998                          NA
## 10 Amy   tropical storm            NA    60      987                          NA
## # i 14,065 more rows
## # i abbreviated name: 1: tropicalstorm_force_diameter
## # i 3 more variables: hurricane_force_diameter <int>, data <list>, n <list>
```

Apply a function to a list-column and create a regular column. In this example, `nrow()` returns one integer per row.

```
n_storms |>
  rowwise() |>
  mutate(n = nrow(data)) # nrow() returns one integer per row
## # A tibble: 14,075 x 9
## # Rowwise:
##    name  status              category  wind pressure tropicalstorm_force_diame~1
##    <chr> <fct>                  <dbl> <int>    <int>                       <int>
##  1 Amy   tropical depression       NA    25     1013                          NA
##  2 Amy   tropical depression       NA    25     1012                          NA
##  3 Amy   tropical depression       NA    25     1011                          NA
##  4 Amy   tropical depression       NA    30     1006                          NA
```

```
##  5 Amy   tropical storm           NA    35     1004                          NA
##  6 Amy   tropical storm           NA    40     1002                          NA
##  7 Amy   tropical storm           NA    45     1000                          NA
##  8 Amy   tropical storm           NA    50      998                          NA
##  9 Amy   tropical storm           NA    55      998                          NA
## 10 Amy   tropical storm           NA    60      987                          NA
## # i 14,065 more rows
## # i abbreviated name: 1: tropicalstorm_force_diameter
## # i 3 more variables: hurricane_force_diameter <int>, data <list>, n <int>
```

Collapse multiple list-columns into a single list-column. In this example, `append()` returns a list for each row, so col type must be list.

```
starwars |>
  rowwise() |>
  mutate(transport = list(append(vehicles, starships))) # append() returns a list for each r
## # A tibble: 87 x 15
## # Rowwise:
##    name     height  mass hair_color skin_color eye_color birth_year sex   gender
##    <chr>     <int> <dbl> <chr>      <chr>      <chr>          <dbl> <chr> <chr>
##  1 Luke Sk~    172    77 blond      fair       blue            19   male  mascu~
##  2 C-3PO       167    75 <NA>       gold       yellow         112   none  mascu~
##  3 R2-D2        96    32 <NA>       white, bl~ red             33   none  mascu~
##  4 Darth V~    202   136 none       white      yellow          41.9 male  mascu~
##  5 Leia Or~    150    49 brown      light      brown           19   fema~ femin~
##  6 Owen La~    178   120 brown, gr~ light      blue            52   male  mascu~
##  7 Beru Wh~    165    75 brown      light      blue            47   fema~ femin~
##  8 R5-D4        97    32 <NA>       white, red red             NA   none  mascu~
##  9 Biggs D~    183    84 black      light      brown           24   male  mascu~
## 10 Obi-Wan~    182    77 auburn, w~ fair       blue-gray       57   male  mascu~
## # i 77 more rows
## # i 6 more variables: homeworld <chr>, species <chr>, films <list>,
## #   vehicles <list>, starships <list>, transport <list>
```

Apply a function to multiple list-columns. In this example, `length()` returns one integer per row.

```
starwars |>
  rowwise() |>
  mutate(n_transports = length(c(vehicles, starships)))
## # A tibble: 87 x 15
```

```
## # Rowwise:
##    name     height  mass hair_color skin_color eye_color birth_year sex   gender
##    <chr>     <int> <dbl> <chr>      <chr>      <chr>          <dbl> <chr> <chr>
##  1 Luke Sk~    172    77 blond      fair       blue            19   male  mascu~
##  2 C-3PO       167    75 <NA>       gold       yellow         112   none  mascu~
##  3 R2-D2        96    32 <NA>       white, bl~ red             33   none  mascu~
##  4 Darth V~    202   136 none       white      yellow          41.9 male  mascu~
##  5 Leia Or~    150    49 brown      light      brown           19   fema~ femin~
##  6 Owen La~    178   120 brown, gr~ light      blue            52   male  mascu~
##  7 Beru Wh~    165    75 brown      light      blue            47   fema~ femin~
##  8 R5-D4        97    32 <NA>       white, red red             NA   none  mascu~
##  9 Biggs D~    183    84 black      light      brown           24   male  mascu~
## 10 Obi-Wan~    182    77 auburn, w~ fair       blue-gray       57   male  mascu~
## # i 77 more rows
## # i 6 more variables: homeworld <chr>, species <chr>, films <list>,
## #   vehicles <list>, starships <list>, n_transports <int>
  # length() returns one integer per row
```

See `purrr` package for more list functions.

Enjoy!