Data reshaping with tidyr and functionals with purrr

Programming for Statistical Science

Shawn Santo

Supplementary materials

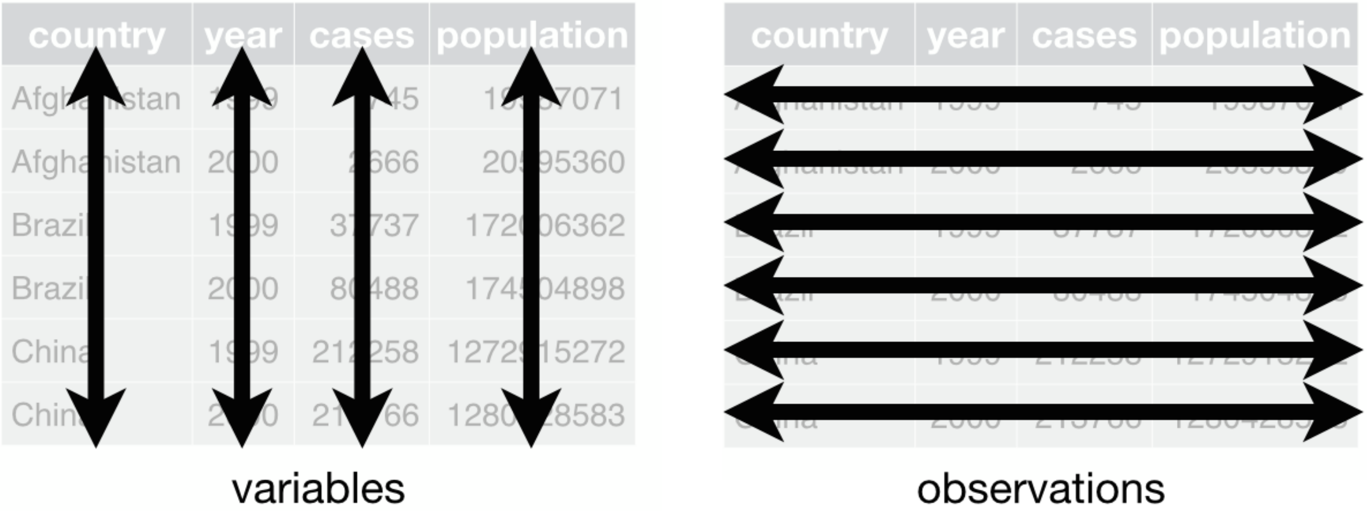
Full video lecture available in Zoom Cloud Recordings

Additional resources

- Sections 9.1 - 9.4, Advanced R
- Chapter 12, R for Data Science
- `tidyr` vignette
- See `vignette("pivot")` in package `tidyr`
- `purrr` tutorial
- `purrr` cheat sheet

tidyr

Tidy data



Source: R for Data Science, https://r4ds.had.co.nz

Getting started

```
library(tidyverse)
```

```
congress <- read_csv("http://www2.stat.duke.edu/~sms185/data/politics/cor
congress
```

```
#> # A tibble: 54 x 12
#>    year_start year_end total_senate dem_senate gop_senate other_senate
#>         <dbl>    <dbl>        <dbl>      <dbl>      <dbl>        <dbl>
#>  1       1913     1915           96         51         44            1
#>  2       1915     1917           96         56         39            1
#>  3       1917     1919           96         53         42            1
#>  4       1919     1921           96         47         48            1
#>  5       1921     1923           96         37         59           NA
#>  6       1923     1925           96         43         51            2
#>  7       1925     1927           96         40         54            1
#>  8       1927     1929           96         47         48            1
#>  9       1929     1931           96         39         56            1
#> 10       1931     1933           96         47         48            1
#> # … with 44 more rows, and 6 more variables: vacant_senate <dbl>,
#> #   total_house <dbl>, dem_house <dbl>, gop_house <dbl>, other_house <dbl>,
#> #   vacant_house <dbl>
```

Smaller data set

```
senate_1913 <- congress %>%
  select(year_start, year_end, contains("senate"), -total_senate) %>%
  arrange(year_start) %>%
  slice(1)

senate_1913
```

```
#> # A tibble: 1 x 6
#>   year_start year_end dem_senate gop_senate other_senate vacant_senate
#>        <dbl>    <dbl>      <dbl>      <dbl>        <dbl>         <dbl>
#> 1       1913     1915         51         44            1            NA
```

Wide to long

```
#> # A tibble: 1 x 6
#>   year_start year_end dem_senate gop_senate other_senate vacant_senate
#>        <dbl>    <dbl>      <dbl>      <dbl>        <dbl>         <dbl>
#> 1       1913     1915         51         44            1            NA
```

```
senate_1913_long <- senate_1913 %>%
  pivot_longer(cols = dem_senate:vacant_senate,
               names_to = "party", values_to = "seats")
senate_1913_long
```

```
#> # A tibble: 4 x 4
#>   year_start year_end party         seats
#>        <dbl>    <dbl> <chr>         <dbl>
#> 1       1913     1915 dem_senate       51
#> 2       1913     1915 gop_senate       44
#> 3       1913     1915 other_senate      1
#> 4       1913     1915 vacant_senate    NA
```

Long to wide

```
#> # A tibble: 4 x 4
#>   year_start year_end party          seats
#>        <dbl>    <dbl> <chr>          <dbl>
#> 1       1913     1915 dem_senate        51
#> 2       1913     1915 gop_senate        44
#> 3       1913     1915 other_senate       1
#> 4       1913     1915 vacant_senate     NA
```

```
senate_1913_long %>%
  pivot_wider(names_from = party, values_from = seats)
```

```
#> # A tibble: 1 x 6
#>   year_start year_end dem_senate gop_senate other_senate vacant_senate
#>        <dbl>    <dbl>      <dbl>      <dbl>        <dbl>         <dbl>
#> 1       1913     1915         51         44            1            NA
```

pivot_*()

Lengthen the data (increase the number of rows, decrease the number of columns)

```
pivot_longer(data, cols, names_to = "col_name", values_to = "col_values")
```

Widen the data (decrease the number of rows, increase the number of columns)

```
pivot_wider(names_from = name_of_var, values_to = var_with_values)
```

Exercise

Consider a tibble of data filtered from world_bank_pop. This dataset is included in package tidyr.

```
usa_pop <- world_bank_pop %>%
  filter(country == "USA")
```

Tidy usa_pop so it looks like the tibble below. See ?world_bank_pop for a description of the variables and their values.

```
#> # A tibble: 6 x 6
#>   country year  sp_urb_totl sp_urb_grow sp_pop_totl sp_pop_grow
#>   <chr>   <chr>       <dbl>       <dbl>       <dbl>       <dbl>
#> 1 USA     2000    223069137        1.51   282162411       1.11 
#> 2 USA     2001    225792302        1.21   284968955       0.990
#> 3 USA     2002    228400290        1.15   287625193       0.928
#> 4 USA     2003    230876596        1.08   290107933       0.859
#> 5 USA     2004    233532722        1.14   292805298       0.925
#> 6 USA     2005    236200507        1.14   295516599       0.922
```

Pivoting

Two older, but related, functions in `tidyr` that you may have encountered before are `gather()` and `spread()`.

- Function `gather()` is similar to function `pivot_longer()` in that it "lengthens" data, increasing the number of rows and decreasing the number of columns.
- Function `spread()` is similar to function `pivot_wider()` in that it makes a dataset wider by increasing the number of columns and decreasing the number of rows.

Check out the vignette for more examples on pivoting data frames.

Unite columns

```
#> # A tibble: 4 x 4
#>   year_start year_end party          seats
#>        <dbl>    <dbl> <chr>          <dbl>
#> 1       1913     1915 dem_senate        51
#> 2       1913     1915 gop_senate        44
#> 3       1913     1915 other_senate       1
#> 4       1913     1915 vacant_senate     NA
```

```
senate_1913_long %>%
  unite(col = "term", year_start:year_end, sep = "-")
```

```
#> # A tibble: 4 x 3
#>   term      party          seats
#>   <chr>     <chr>          <dbl>
#> 1 1913-1915 dem_senate        51
#> 2 1913-1915 gop_senate        44
#> 3 1913-1915 other_senate       1
#> 4 1913-1915 vacant_senate     NA
```

```
unite(data, col, ..., sep = "_", remove = TRUE, na.rm = FALSE)
```

Separate columns

```
#> # A tibble: 4 x 4
#>   year_start year_end party          seats
#>        <dbl>    <dbl> <chr>          <dbl>
#> 1       1913     1915 dem_senate        51
#> 2       1913     1915 gop_senate        44
#> 3       1913     1915 other_senate       1
#> 4       1913     1915 vacant_senate     NA
```

```
senate_1913_long %>%
  separate(col = party, into = c("party", "leg_branch"), sep = "_")
```

```
#> # A tibble: 4 x 5
#>   year_start year_end party  leg_branch seats
#>        <dbl>    <dbl> <chr>  <chr>      <dbl>
#> 1       1913     1915 dem    senate        51
#> 2       1913     1915 gop    senate        44
#> 3       1913     1915 other  senate         1
#> 4       1913     1915 vacant senate        NA
```

```
separate(data, col, into, sep = "[^[:alnum:]]+", remove = TRUE,
  convert = FALSE, extra = "warn", fill = "warn", ...)
```

Functionals

What is a functional?

A functional is a function that takes a function as an input and returns a vector as output.

```
fixed_point <- function(f, x0, tol = .0001, ...) {
  y <- f(x0, ...)
  x_new <- x0

  while (abs(y - x_new) > tol) {
    x_new <- y
    y <- f(x_new, ...)
  }

  return(x_new)
}
```

Argument `f` takes in a function name.

```
fixed_point(cos, 1)
```

```
#> [1] 0.7391302
```

```
fixed_point(sin, 0)
```

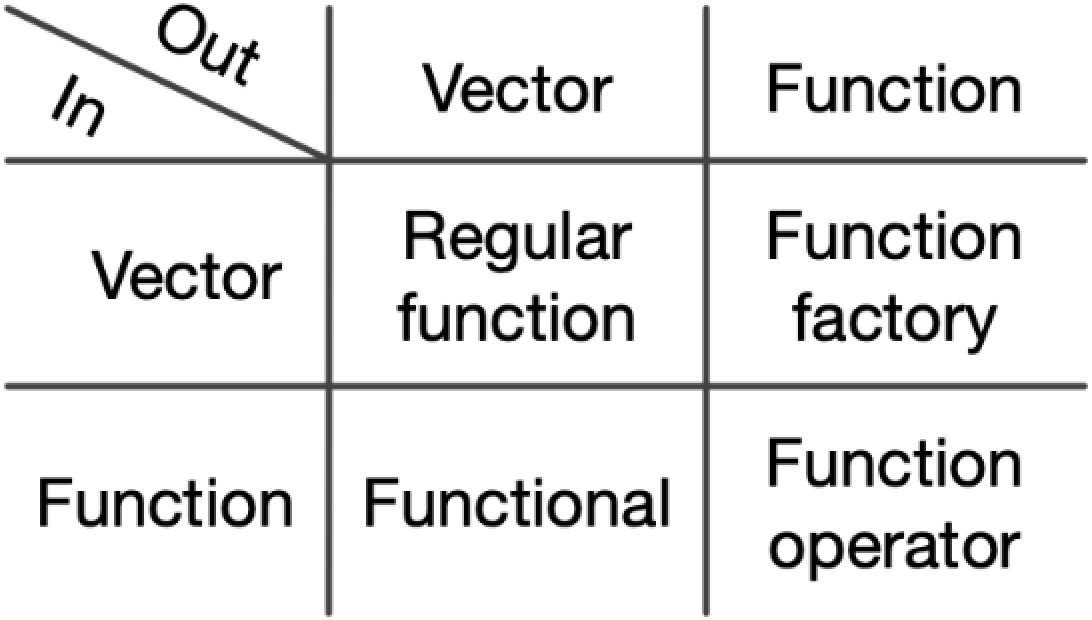
```
#> [1] 0
```

```
fixed_point(f = sqrt, x0 = .01, tol = .000000001)
```

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

Functional programming

A functional is one property of first-class functions and part of what makes a language a functional programming language.



Out In	Vector	Function
Vector	Regular function	Function factory
Function	Functional	Function operator

Apply functions

[a-z]pply() functions

The apply functions are a collection of tools for functional programming in R, they are variations of the `map` function found in many other languages.

lapply()

Usage: `lapply(X, FUN, ...)`

`lapply()` returns a list of the same length as `X`, each element of which is the result of applying `FUN` to the corresponding element of `X`.

```
lapply(1:8, sqrt) %>%
  str()
```

```
#> List of 8
#>  $ : num 1
#>  $ : num 1.41
#>  $ : num 1.73
#>  $ : num 2
#>  $ : num 2.24
#>  $ : num 2.45
#>  $ : num 2.65
#>  $ : num 2.83
```

```
lapply(1:8, function(x) (x+1)^2) %
  str()
```

```
#> List of 8
#>  $ : num 4
#>  $ : num 9
#>  $ : num 16
#>  $ : num 25
#>  $ : num 36
#>  $ : num 49
#>  $ : num 64
#>  $ : num 81
```

```r
lapply(1:8, function(x, pow) x ^ pow, 3) %>%
  str()
```

```
#> List of 8
#>  $ : num 1
#>  $ : num 8
#>  $ : num 27
#>  $ : num 64
#>  $ : num 125
#>  $ : num 216
#>  $ : num 343
#>  $ : num 512
```

```r
pow <- function(x, pow) x ^ pow
lapply(1:8, pow, x = 2) %>%
  str()
```

```
#> List of 8
#>  $ : num 2
#>  $ : num 4
#>  $ : num 8
#>  $ : num 16
#>  $ : num 32
#>  $ : num 64
#>  $ : num 128
#>  $ : num 256
```

`sapply()`

Usage: `sapply(X, FUN, ..., simplify = TRUE, USE.NAMES = TRUE)`

`sapply()` is a *user-friendly* version and wrapper of `lapply`, it is a *simplifying* version of lapply. Whenever possible it will return a vector, matrix, or an array.

```r
sapply(1:8, sqrt) %>%
  round(2)
```

```
#> [1] 1.00 1.41 1.73 2.00 2.24 2.45 2.65 2.83
```

```r
sapply(1:8, function(x) (x + 1)^2)
```

```
#> [1]  4  9 16 25 36 49 64 81
```

```r
sapply(1:8, function(x) c(x, x^2, x^3, x^4))
```

```
#>      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
#> [1,]    1    2    3    4    5    6    7    8
#> [2,]    1    4    9   16   25   36   49   64
#> [3,]    1    8   27   64  125  216  343  512
#> [4,]    1   16   81  256  625 1296 2401 4096
```

```r
sapply(1:8, function(x) list(x, x^2, x^3, x^4))
```

```
#>      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
#> [1,] 1    2    3    4    5    6    7    8
#> [2,] 1    4    9    16   25   36   49   64
#> [3,] 1    8    27   64   125  216  343  512
#> [4,] 1    16   81   256  625  1296 2401 4096
```

```
sapply(2:6, seq)
```

```
#> [[1]]
#> [1] 1 2
#> 
#> [[2]]
#> [1] 1 2 3
#> 
#> [[3]]
#> [1] 1 2 3 4
#> 
#> [[4]]
#> [1] 1 2 3 4 5
#> 
#> [[5]]
#> [1] 1 2 3 4 5 6
```

Why do we have a list?

```
sapply(2:6, seq, from = 1, length.out = 4)
```

```
#>          [,1]     [,2] [,3]     [,4]     [,5]
#> [1,] 1.000000 1.000000    1 1.000000 1.000000
#> [2,] 1.333333 1.666667    2 2.333333 2.666667
#> [3,] 1.666667 2.333333    3 3.666667 4.333333
#> [4,] 2.000000 3.000000    4 5.000000 6.000000
```

[ls]apply() and data frames

We can use these functions with data frames, the key is to remember that a data frame is just a fancy list.

```
df <-  data.frame(a = 1:6, b = letters[1:6], c = c(TRUE,FALSE))
lapply(df, class) %>% str()
```

```
#> List of 3
#>  $ a: chr "integer"
#>  $ b: chr "character"
#>  $ c: chr "logical"
```

```
sapply(df, class)
```

```
#>           a           b           c 
#>   "integer" "character"   "logical" 
```

More in the family

- `apply(X, MARGIN, FUN, ...)` - applies a function over the rows or columns of a data frame, matrix, or array
- `vapply(X, FUN, FUN.VALUE, ..., USE.NAMES = TRUE)` - is similar to `sapply()`, but has a enforced return type and size
- `mapply(FUN, ..., MoreArgs = NULL, SIMPLIFY = TRUE, USE.NAMES = TRUE)` - like `sapply()` but will iterate over multiple vectors at the same time.
- `rapply(object, f, classes = "ANY", deflt = NULL, how = c("unlist", "replace", "list"), ...)` - a recursive version of `lapply()`, behavior depends largely on the `how` argument
- `eapply(env, FUN, ..., all.names = FALSE, USE.NAMES = TRUE)` - apply a function over an environment.

Exercise

Using `sw_people` in package `repurrrsive`, extract the name of all characters using:

- a for loop,
- an apply function.

```r
library(repurrrsive)
str(sw_people[[1]])
```

```
#> List of 16
#>  $ name      : chr "Luke Skywalker"
#>  $ height    : chr "172"
#>  $ mass      : chr "77"
#>  $ hair_color: chr "blond"
#>  $ skin_color: chr "fair"
#>  $ eye_color : chr "blue"
#>  $ birth_year: chr "19BBY"
#>  $ gender    : chr "male"
#>  $ homeworld : chr "http://swapi.co/api/planets/1/"
#>  $ films     : chr [1:5] "http://swapi.co/api/films/6/" "http://swapi.co/api/films/3/" "http://swapi.co
#>  $ species   : chr "http://swapi.co/api/species/1/"
#>  $ vehicles  : chr [1:2] "http://swapi.co/api/vehicles/14/" "http://swapi.co/api/vehicles/30/"
#>  $ starships : chr [1:2] "http://swapi.co/api/starships/12/" "http://swapi.co/api/starships/22/"
#>  $ created   : chr "2014-12-09T13:50:51.644000Z"
#>  $ edited    : chr "2014-12-20T21:17:56.891000Z"
#>  $ url       : chr "http://swapi.co/api/people/1/"
```

Hint: The `[` and `[[` are functions.

`purrr`

Why `purrr`?

- Member of the `tidyverse` package
- Improves the functional programming tools in R
- The `map()` family of functions can be used to replace loops and `[a-z]pply()`
  - Consistent output
  - Easier to read and write

Map functions

Basic functions for looping over an object and returning a value (of a specific type) - replacement for `lapply()`/`sapply()`/`vapply()`.

Map variant	Description
`map()`	returns a list
`map_lgl()`	returns a logical vector
`map_int()`	returns a integer vector
`map_dbl()`	returns a double vector
`map_chr()`	returns a character vector
`map_df()` / `map_dfr()`	returns a data frame by row binding
`map_dfc()`	returns a data frame by column binding

All have leading arguments `.x` and `.f`.

map_*() is strict

```
x <- list(1L:5L, c(-2, .2, -20), c(pi, sqrt(2), 7))
```

```
map_dbl(x, mean)
```

```
#> [1]  3.000000 -7.266667  3.851935
```

```
map_chr(x, mean)
```

```
#> [1] "3.000000"  "-7.266667" "3.851935"
```

```
map_lgl(x, mean)
```

```
#> Error: Can't coerce element 1 from a double to a logical
```

```
map_int(x, mean)
```

```
#> Error: Can't coerce element 1 from a double to a integer
```

```
x <- list(1L:5L, c(-2, .2, -20), c(pi, sqrt(2), 7))
```

```
map_dbl(x, `[`, 1)
```

```
#> [1]  1.000000 -2.000000  3.141593
```

```
map_chr(x, `[`, 3)
```

```
#> [1] "3"          "-20.000000" "7.000000"
```

```
map_lgl(x, `[`, 1)
```

```
#> Error: Can't coerce element 1 from a integer to a logical
```

```
map_int(x, `[`, 1)
```

```
#> Error: Can't coerce element 2 from a double to a integer
```

Flexibility in `.f`

Argument `.f` in `map()` and `map_*()` can take a

- function,
- formula (one sided) / anonymous function, or a
- vector.
  - character vector
  - numeric vector
  - list

Examples

Using `map_*()`

```
map_dbl(1:5, ~ . ^ .)
```

```
#> [1]    1    4   27  256 3125
```

```
map_dbl(1:5, ~ .x ^ .x)
```

```
#> [1]    1    4   27  256 3125
```

```
map2_dbl(1:5, -1:-5,  ~ .y ^ .x)
```

```
#> [1]    -1     4   -27   256 -3125
```

```
pmap_dbl(data.frame(1:5, 1:5, 1:5),
         ~..1 + ..2 + ..3)
```

```
#> [1]  3  6  9 12 15
```

Using Base R

```
sapply(1:5, function(x) x ^ x)
```

```
#> [1]    1    4   27  256 3125
```

```
sapply(1:5, function(x) x ^ x)
```

```
#> [1]    1    4   27  256 3125
```

```
sapply(1:5, function(x, y) y ^ x, y = -1:-5)
  diag()
```

```
#> [1]    -1     4   -27   256 -3125
```

```
sapply(1:5, function(x, y, z) x + y + z, y =
  diag()
```

```
#> [1]  3  6  9 12 15
```

More examples

Consider `gh_users` from package `repurrrsive`.

```r
library(repurrrsive)
str(gh_users, max.level = 1)
```

```
#> List of 6
#>  $ :List of 30
#>  $ :List of 30
#>  $ :List of 30
#>  $ :List of 30
#>  $ :List of 30
#>  $ :List of 30
```

```r
str(gh_users[[1]], max.level = 1)
```

```
#> List of 30
#>  $ login              : chr "gaborcsardi"
#>  $ id                 : int 660288
#>  $ avatar_url         : chr "https://avatars.githubusercontent.com/u/660288?v=3"
#>  $ gravatar_id        : chr ""
#>  $ url                : chr "https://api.github.com/users/gaborcsardi"
#>  $ html_url           : chr "https://github.com/gaborcsardi"
#>  $ followers_url      : chr "https://api.github.com/users/gaborcsardi/followers"
#>  $ following_url      : chr "https://api.github.com/users/gaborcsardi/following{/other_user}"
#>  $ gists_url          : chr "https://api.github.com/users/gaborcsardi/gists{/gist_id}"
#>  $ starred_url        : chr "https://api.github.com/users/gaborcsardi/starred{/owner}{/repo}"
#>  $ subscriptions_url  : chr "https://api.github.com/users/gaborcsardi/subscriptions"
#>  $ organizations_url  : chr "https://api.github.com/users/gaborcsardi/orgs"
#>  $ repos_url          : chr "https://api.github.com/users/gaborcsardi/repos"
#>  $ events_url         : chr "https://api.github.com/users/gaborcsardi/events{/privacy}"
#>  $ received_events_url: chr "https://api.github.com/users/gaborcsardi/received_events"
#>  $ type               : chr "User"
#>  $ site_admin         : logi FALSE
#>  $ name               : chr "Gábor Csárdi"
#>  $ company            : chr "Mango Solutions, @MangoTheCat "
#>  $ blog               : chr "http://gaborcsardi.org"
#>  $ location           : chr "Chippenham, UK"
#>  $ email              : chr "csardi.gabor@gmail.com"
#>  $ hireable           : NULL
#>  $ bio                : NULL
#>  $ public_repos       : int 52
#>  $ public_gists       : int 6
#>  $ followers          : int 303
#>  $ following          : int 22
#>  $ created_at         : chr "2011-03-09T17:29:25Z"
#>  $ updated_at         : chr "2016-10-11T11:05:06Z"
```

```r
map_chr(gh_users, "login")
```

```
#> [1] "gaborcsardi" "jennybc"     "jtleek"      "juliasilge"  "leeper"
#> [6] "masalmon"
```

```r
map_chr(gh_users, 1)
```

```
#> [1] "gaborcsardi" "jennybc"     "jtleek"      "juliasilge"  "leeper"
#> [6] "masalmon"
```

```r
map_chr(gh_users, c(1, 2))
```

```
#> Error: Result 1 must be a single string, not NULL of length 0
```

```
map(gh_users, `[`, c(1, 2)) %>%
  str()
```

```
#> List of 6
#>  $ :List of 2
#>   ..$ login: chr "gaborcsardi"
#>   ..$ id   : int 660288
#>  $ :List of 2
#>   ..$ login: chr "jennybc"
#>   ..$ id   : int 599454
#>  $ :List of 2
#>   ..$ login: chr "jtleek"
#>   ..$ id   : int 1571674
#>  $ :List of 2
#>   ..$ login: chr "juliasilge"
#>   ..$ id   : int 12505835
#>  $ :List of 2
#>   ..$ login: chr "leeper"
#>   ..$ id   : int 3505428
#>  $ :List of 2
#>   ..$ login: chr "masalmon"
#>   ..$ id   : int 8360597
```

```
map(gh_users, `[[`, c(1, 2)) %>%
  str()
```

```
#> Error in .x[[...]]: subscript out of bounds
```

```
map_dbl(gh_users, list(28, 1))
```

```
#> [1]  22  34   6  10 230  38
```

```
map_dbl(gh_users, list("following", 1))
```

```
#> [1]  22  34   6  10 230  38
```

To make the above more clear:

```
my_list <- list(
  list(x = 1:10, y = 6, z = c(9, 0)),
  list(x = 1:10, y = 6, z = c(-3, 2))
)
map_chr(my_list, list("z", 2))
```

```
#> [1] "0.000000" "2.000000"
```

```
map_chr(my_list, list(3, 1))
```

```
#> [1] "9.000000"  "-3.000000"
```

```
map_df(gh_users, `[`, c(1, 2))
```

```
#> # A tibble: 6 x 2
#>   login             id
#>   <chr>          <int>
#> 1 gaborcsardi   660288
#> 2 jennybc       599454
#> 3 jtleek       1571674
#> 4 juliasilge  12505835
#> 5 leeper       3505428
#> 6 masalmon     8360597
```

```
map_df(gh_users, `[`, c("name", "type", "location"))
```

```
#> # A tibble: 6 x 3
#>   name                     type  location
#>   <chr>                    <chr> <chr>
#> 1 Gábor Csárdi             User  Chippenham, UK
#> 2 Jennifer (Jenny) Bryan User  Vancouver, BC, Canada
#> 3 Jeff L.                  User  Baltimore,MD
#> 4 Julia Silge              User  Salt Lake City, UT
#> 5 Thomas J. Leeper         User  London, United Kingdom
#> 6 Maëlle Salmon            User  Barcelona, Spain
```

`map()` variants

- `walk()` - returns nothing, call function exclusively for its side effects
- `modify()` - returns the same type as the input object, useful for data frames

```
df <- data_frame(x = 1:3, y = -1:-3)
modify(df, ~ .x ^ 3)
```

```
#> # A tibble: 3 x 2
#>       x     y
#>   <dbl> <dbl>
#> 1     1    -1
#> 2     8    -8
#> 3    27   -27
```

- `map2()` and `pmap()` to vary two and n inputs, respectively
- `imap()` iterate over indices and values

Exercise

Use `mtcars` and a single map or map variant to

- get the type of each variable,
- get the fourth row such that result is a character vector,
- compute the mean of each variable, and
- compute the mean and median for each variable such that the result is a data frame with the mean values in row 1 and the median values in row 2.

References

1. Grolemund, G., & Wickham, H. (2020). R for Data Science. https://r4ds.had.co.nz/

2. Wickham, H. (2020). Advanced R. https://adv-r.hadley.nz/