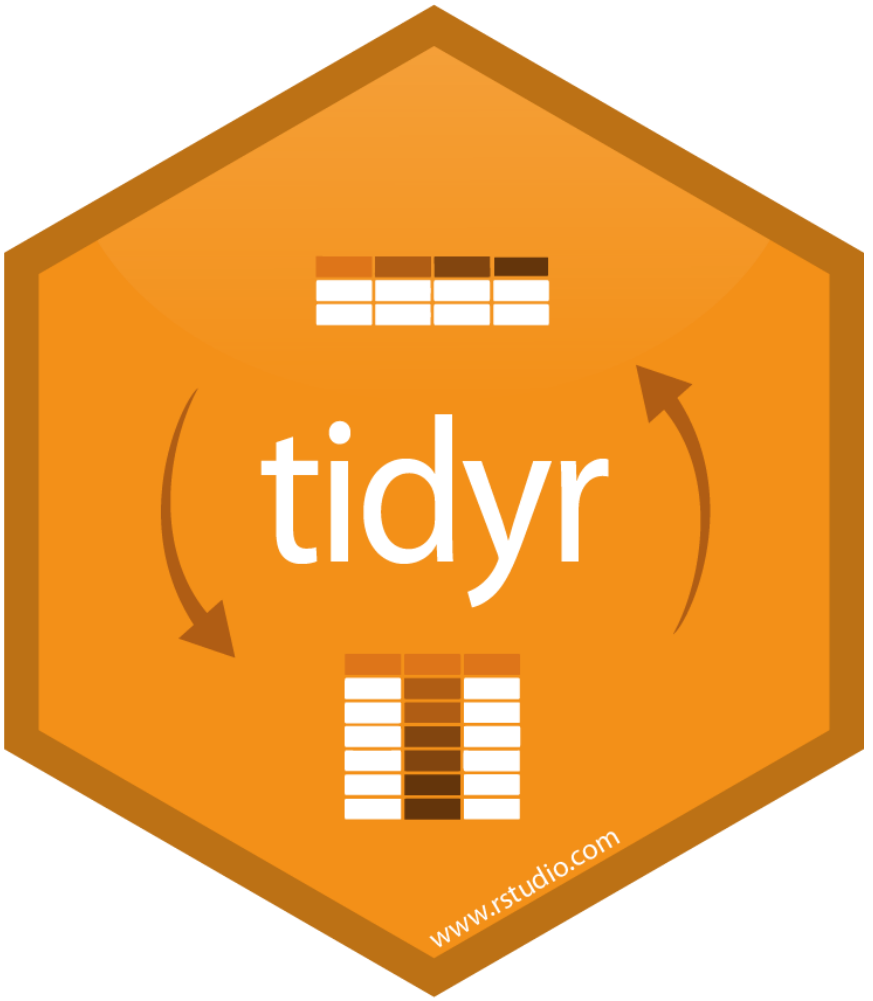
tidyr and purrr

Colin Rundel

2019-10-07


tidyr
www.rstudio.com

Example - Grades

Is the following data tidy?

```
(grades = tibble(
  name  = c("Alice", "Bob", "Carol", "Dave"),
  hw_1   = c(19, 18, 18, 19),
  hw_2   = c(19, 20, 20, 19),
  hw_3   = c(18, 18, 18, 18),
  hw_4   = c(20, 16, 17, 19),
  exam_1 = c(89, 77, 96, 86),
  exam_2 = c(95, 88, 99, 82)
))
```

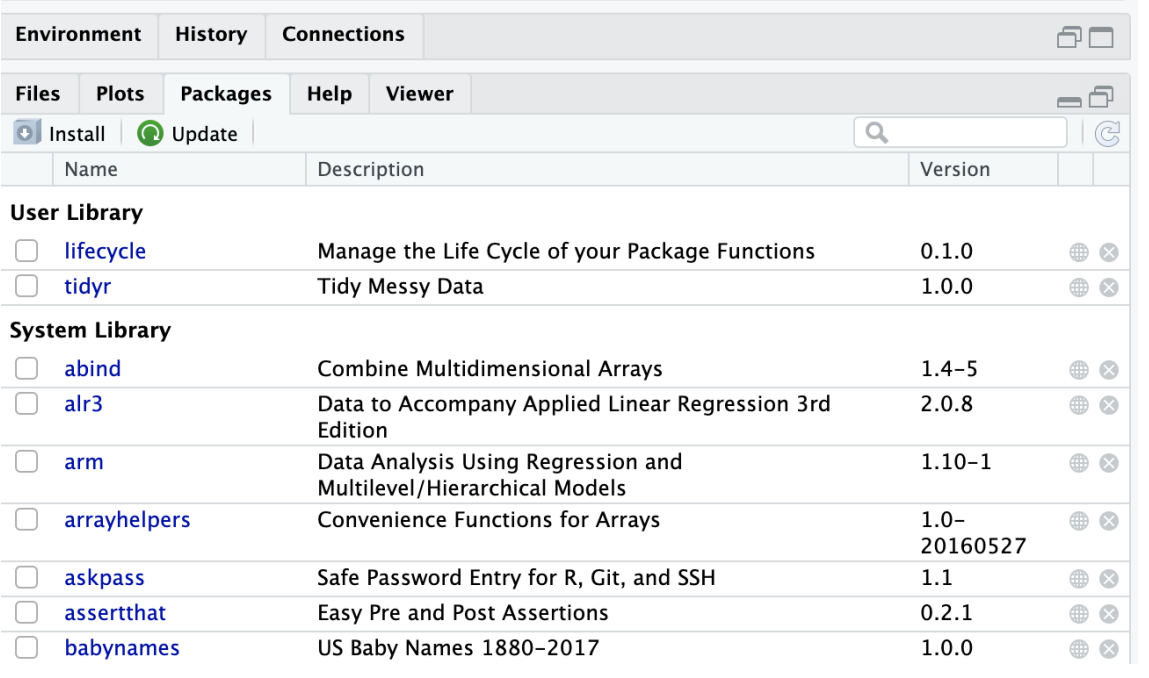
```
## # A tibble: 4 x 7
##   name   hw_1  hw_2  hw_3  hw_4 exam_1 exam_2
##   <chr> <dbl> <dbl> <dbl> <dbl>  <dbl>  <dbl>
## 1 Alice    19    19    18    20     89     95
## 2 Bob      18    20    18    16     77     88
## 3 Carol    18    20    18    17     96     99
## 4 Dave     19    19    18    19     86     82
```

This is an example of *wide* data, which is almost never *tidy*.

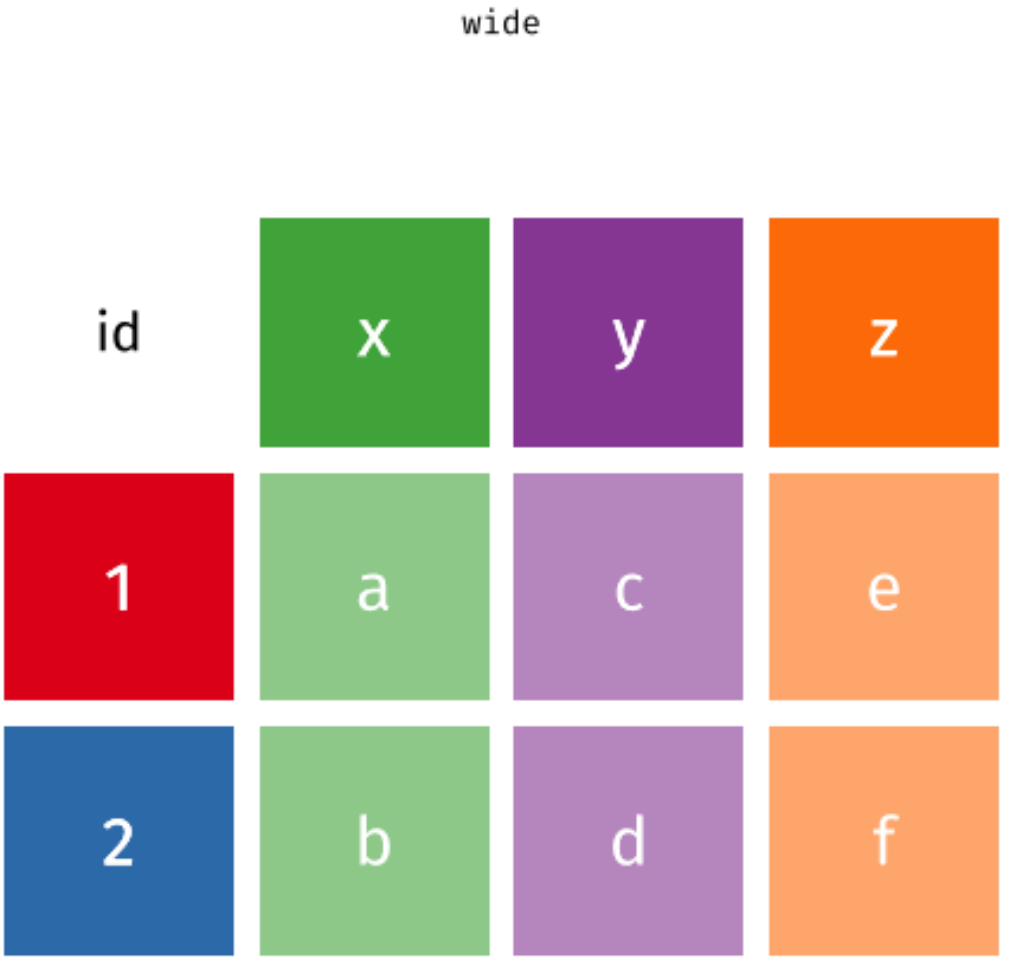
Updating `tidyr`

The current version of tidyr installed in Noteable is slightly out of date (v0.8.3 vs v1.0.0). To fix this run the following,

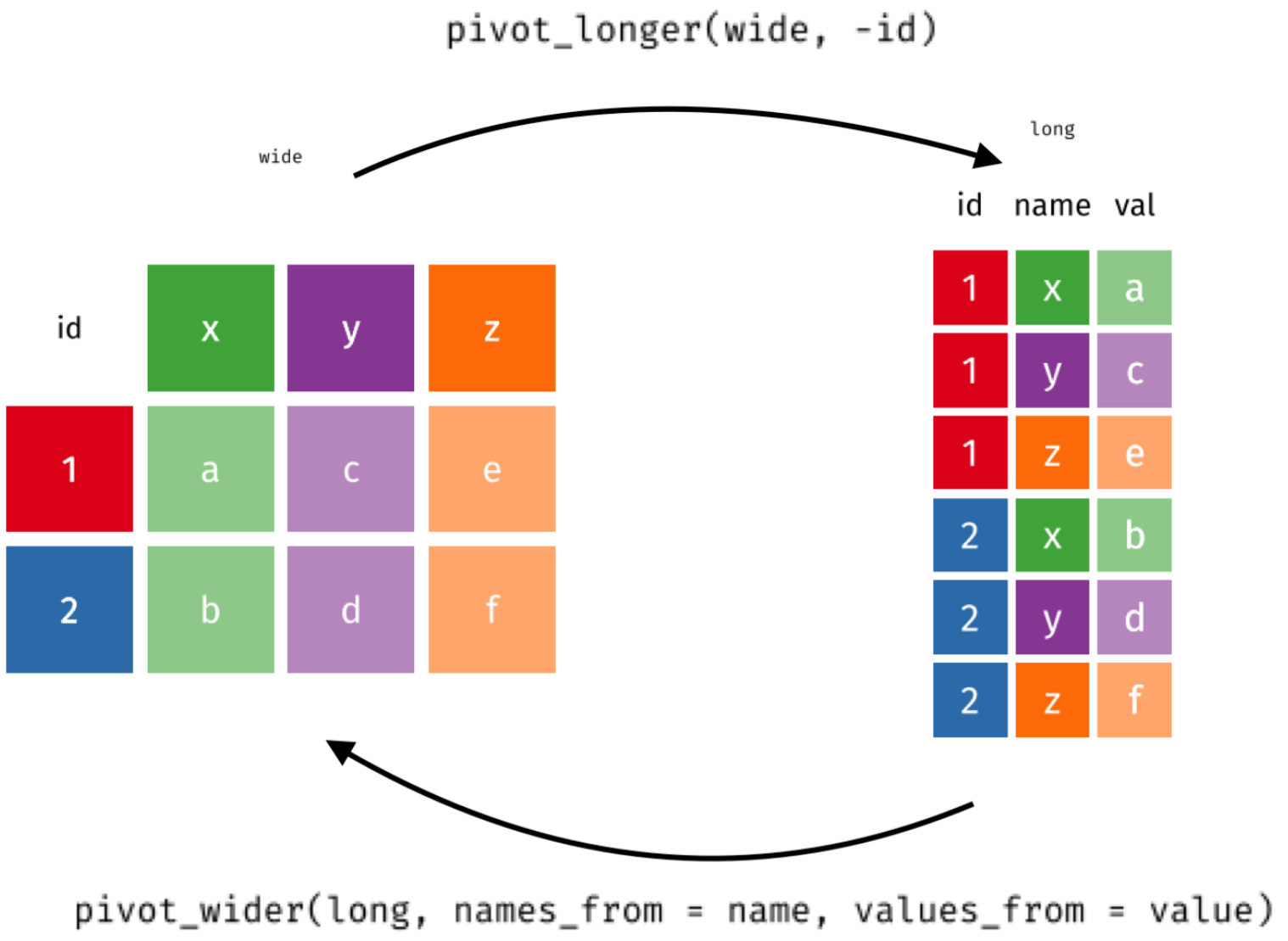
```
lib = Sys.getenv("R_LIBS_USER")
dir.create(lib, recursive=TRUE, showWarnings=FALSE)
install.packages("tidyr", lib=lib)
```



Wider <-> Longer



From Mara Averick's tidyexplain repo


pivot_longer(wide, -id)
wide
long
id
x
y
z
1
a
c
e
2
b
d
f
id
name
val
1
x
a
1
y
c
1
z
e
2
x
b
2
y
d
2
z
f
pivot_wider(long, names_from = name, values_from = value)

pivot_longer

```
pivot_longer(table, cols = -country, names_to = "year", values_to = "cases")
```

country	1999	2000
A	0.7K	2K
B	37K	80K
C	212K	213K

country	year	cases
A	1999	0.7K
B	1999	37K
C	1999	212K
A	2000	2K
B	2000	80K
C	2000	213K

pivot_wider

```
pivot_wider(table, id_cols = country:year, names_from = type, values_from = count)
```

country	year	type	count
A	1999	cases	0.7K
A	1999	pop	19M
A	2000	cases	2K
A	2000	pop	20M
B	1999	cases	37K
B	1999	pop	172M
B	2000	cases	80K
B	2000	pop	174M
C	1999	cases	212K
C	1999	pop	1T
C	2000	cases	213K
C	2000	pop	1T

country	year	cases	pop
A	1999	0.7K	19M
A	2000	2K	20M
B	1999	37K	172M
B	2000	80K	174M
C	1999	212K	1T
C	2000	213K	1T

Separate

```
separate(table, col = rate, sep = "/", into = c("cases", "pop"))
```

country	year	rate
A	1999	0.7K/19M
A	2000	2K/20M
B	1999	37K/172M
B	2000	80K/174M
C	1999	212K/1T
C	2000	213K/1T

country	year	cases	pop
A	1999	0.7K	19M
A	2000	2K	20M
B	1999	37K	172
B	2000	80K	174
C	1999	212K	1T
C	2000	213K	1T

Unite

```
unite(table, century, year, col = "year", sep = "")
```

country	century	year
Afghan	19	99
Afghan	20	0
Brazil	19	99
Brazil	20	0
China	19	99
China	20	0

country	year
Afghan	1999
Afghan	2000
Brazil	1999
Brazil	2000
China	1999
China	2000

Example 1 - Summarizing Grades

Is the following data tidy?

```
(grades = tibble(
  name  = c("Alice", "Bob", "Carol", "Dave"),
  hw_1   = c(19, 18, 18, 19),
  hw_2   = c(19, 20, 20, 19),
  hw_3   = c(18, 18, 18, 18),
  hw_4   = c(20, 16, 17, 19),
  exam_1 = c(89, 77, 96, 86),
  exam_2 = c(95, 88, 99, 82)
))
```

```
## # A tibble: 4 x 7
##   name   hw_1  hw_2  hw_3  hw_4 exam_1 exam_2
##   <chr> <dbl> <dbl> <dbl> <dbl>  <dbl>  <dbl>
## 1 Alice    19    19    18    20     89     95
## 2 Bob      18    20    18    16     77     88
## 3 Carol    18    20    18    17     96     99
## 4 Dave     19    19    18    19     86     82
```

How would we calculate a final score based on the following formula,

$$\text{score} = 0.6 \frac{\sum \text{hw}_i}{80} + 0.4 \frac{\sum \text{exam}_j}{200}$$

Semi-tidy approach

```
grades %>%
  mutate(
    hw_avg = (hw_1+hw_2+hw_3+hw_4)/4,
    exam_avg = (exam_1+exam_2)/2
  ) %>%
  mutate(
    overall = 0.4*(exam_avg/100) + 0.6*(hw_avg/20)
  )
```

```
## # A tibble: 4 x 10
##   name   hw_1  hw_2  hw_3  hw_4 exam_1 exam_2 hw_avg exam_avg overall
##   <chr> <dbl> <dbl> <dbl> <dbl>  <dbl>  <dbl>  <dbl>    <dbl>   <dbl>
## 1 Alice    19    19    18    20     89     95   19       92     0.938
## 2 Bob      18    20    18    16     77     88   18       82.5   0.87
## 3 Carol    18    20    18    17     96     99   18.2     97.5   0.938
## 4 Dave     19    19    18    19     86     82   18.8     84     0.899
```

What is problematic about this approach?

Wide -> Long (pivot_longer)

```
tidyr::pivot_longer(grades, cols = hw_1:exam_2,
                    names_to = "assignment",
                    values_to = "score")
```

```
## # A tibble: 24 x 3
##    name  assignment score
##    <chr> <chr>      <dbl>
##  1 Alice hw_1          19
##  2 Alice hw_2          19
##  3 Alice hw_3          18
##  4 Alice hw_4          20
##  5 Alice exam_1        89
##  6 Alice exam_2        95
##  7 Bob   hw_1          18
##  8 Bob   hw_2          20
##  9 Bob   hw_3          18
## 10 Bob   hw_4          16
## # … with 14 more rows
```

```
tidyr::pivot_longer(grades, cols = hw_1:exam_2,
                    names_to = c("type", "id"), names_sep = "_",
                    values_to = "score")
```

```
## # A tibble: 24 x 4
##    name  type  id    score
##    <chr> <chr> <chr> <dbl>
##  1 Alice hw    1        19
##  2 Alice hw    2        19
##  3 Alice hw    3        18
##  4 Alice hw    4        20
##  5 Alice exam  1        89
##  6 Alice exam  2        95
##  7 Bob   hw    1        18
##  8 Bob   hw    2        20
##  9 Bob   hw    3        18
## 10 Bob   hw    4        16
## # … with 14 more rows
```

Tidy approach?

```
grades %>%
  tidyr::pivot_longer(
    cols = hw_1:exam_2,
    names_to = c("type", "id"), names_sep = "_",
    values_to = "score"
  ) %>%
  group_by(name, type) %>%
  summarize(total = sum(score))
```

```
## # A tibble: 8 x 3
## # Groups:   name [4]
##   name  type  total
##   <chr> <chr> <dbl>
## 1 Alice exam    184
## 2 Alice hw       76
## 3 Bob   exam    165
## 4 Bob   hw       72
## 5 Carol exam    195
## 6 Carol hw       73
## 7 Dave  exam    168
## 8 Dave  hw       75
```

Long -> Wide (pivot_wider)

```
grades %>%
  tidyr::pivot_longer(
    cols = hw_1:exam_2,
    names_to = c("type", "id"), names_sep = "_",
    values_to = "score"
  ) %>%
  group_by(name, type) %>%
  summarize(total = sum(score)) %>%
  tidyr::pivot_wider(
    names_from = type, values_from = total
  )
```

```
## # A tibble: 4 x 3
## # Groups:   name [4]
##   name   exam    hw
##   <chr> <dbl> <dbl>
## 1 Alice   184    76
## 2 Bob     165    72
## 3 Carol   195    73
## 4 Dave    168    75
```

Apply functions

Apply 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

```
??apply
---

##
## Help files with alias or concept or title matching ‘apply’ using fuzzy
## matching:
##
## base::apply             Apply Functions Over Array Margins
## base::.subset           Internal Objects in Package 'base'
## base::by                Apply a Function to a Data Frame Split by Factors
## base::eapply            Apply a Function Over Values in an Environment
## base::lapply            Apply a Function over a List or Vector
## base::mapply            Apply a Function to Multiple List or Vector Arguments
## base::rapply            Recursively Apply a Function to a List
## base::tapply            Apply a Function Over a Ragged Array
```

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
```

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

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

```
lapply(1:8, function(x, pow) x^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.

```
sapply(1:8, sqrt)
```

```
## [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751 2.828427
```

```
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
```

[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 "factor"
##  $ c: chr "logical"
```

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

```
##         a         b         c
## "integer"  "factor" "logical"
```

other less common applies

- `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 1

Using the `sw_people` data set in the `repurrrsive` package, extract the names of all of the characters using:

- a for loop
- one of the apply functions

Start by examining the structure of the data using RStudio's viewer,

```
library(repurrrsive)
View(sw_people)
```


purrr
www.rstudio.com

Map functions

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

- `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_dfr()` - returns a data frame by row binding.
- `map_dfc()` - returns a data frame by column binding.
- `walk()` - returns nothing, call function exclusively for its side effects

Type Consistency

R is a weakly / dynamically typed language which means there is no simple way to define a function which enforces the argument or return types. This flexibility can be useful at times, but often it makes it hard to reason about your code and requires more verbose code to handle edge cases.

```
x = list(rnorm(1e3),rnorm(1e3),rnorm(1e3))
```

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

```
## [1] -0.009105024  0.035028661 -0.027726877
```

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

```
## [1] "-0.009105" "0.035029"  "-0.027727"
```

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

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

Shortcut - Anonymous Functions

An anonymous function is one that is never given a name (assigned to a variable)

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

```
## [1]     1     8    81  1024 15625
```

purrr lets us write anonymous functions using one sided formulas where the argument is given by `.` or `.x` for `map` and related functions.

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

```
## [1]     1     8    81  1024 15625
```

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

```
## [1]     1     8    81  1024 15625
```

Shortcut - Anonymous Functions - `map2`

Functions with the `map2` prefix work the same as the `map` functions but they iterate over two objects instead of one. Arguments in an anonymous function are instead given by `.x` and `.y` (or `..1` and `..2`) respectively.

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

```
## [1]     1     8    81  1024 15625
```

```
map2_dbl(1:5, 1:5, ~ ..1^(..2+1))
```

```
## [1]     1     8    81  1024 15625
```

```
map2_chr(letters[1:5], LETTERS[1:5], paste0)
```

```
## [1] "aA" "bB" "cC" "dD" "eE"
```

Purrr shortcut - Lookups

Very often we want to extract only certain (named) values from a list, `purrr` provides a shortcut for this operation when you provide either a character or numeric value instead of a function to apply.

```
x = list(list(a=1L,b=2L,c=list(d=3L,e=4L)),
         list(a=5L,b=6L,c=list(d=7L,e=8L,f=9L)))
```

```
map_int(x, "a")
```

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

```
map_dbl(x, c("c","e"))
```

```
## [1] 4 8
```

```
map_chr(x, list(3,"d"))
```

```
## [1] "3" "7"
```

```
map_df(x, 3)
```

```
## # A tibble: 2 x 3
##       d     e     f
##   <int> <int> <int>
## 1     3     4    NA
## 2     7     8     9
```

```
map_dfc(x, 3)
```

```
## # A tibble: 1 x 5
##       d     e    d1    e1     f
##   <int> <int> <int> <int> <int>
## 1     3     4     7     8     9
```

```
x = list(list(a=1L,b=2L,c=list(d=3L,e=4L)),
         list(a=5L,b=6L,c=list(d=7L,e=8L,f=9L)))
```

```
map(x, list(3,"f"))
```

```
## [[1]]
## NULL
##
## [[2]]
## [1] 9
```

```
map_int(x, list(3,"f"))
```

```
## Result 1 must be a single integer, not NULL of length 0
```

```
map_int(x, list(3,"f"), .default=NA)
```

```
## [1] NA  9
```

Exercise 2

Using the `sw_people` data set again, generate a tidy data frame (tibble) containing as many details as possible.

list columns

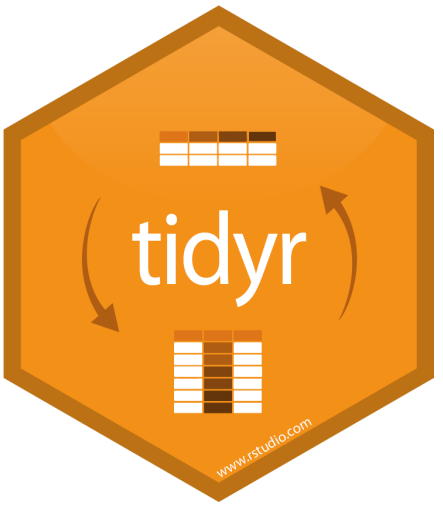
```
d = tibble(
  name = purrr::map_chr(sw_people, "name"),
  starships = purrr::map(sw_people, "starships")
)

d
```

```
## # A tibble: 87 x 2
##    name               starships
##    <chr>              <list>
##  1 Luke Skywalker     <chr [2]>
##  2 C-3PO              <NULL>
##  3 R2-D2              <NULL>
##  4 Darth Vader        <chr [1]>
##  5 Leia Organa        <NULL>
##  6 Owen Lars          <NULL>
##  7 Beru Whitesun lars <NULL>
##  8 R5-D4              <NULL>
##  9 Biggs Darklighter  <chr [1]>
## 10 Obi-Wan Kenobi     <chr [5]>
## # … with 77 more rows
```

```
d %>%
  mutate(
    n_starships = purrr::map_int(starships, length)
  )
```

```
## # A tibble: 87 x 3
##    name               starships n_starships
##    <chr>              <list>          <int>
##  1 Luke Skywalker     <chr [2]>           2
##  2 C-3PO              <NULL>              0
##  3 R2-D2              <NULL>              0
##  4 Darth Vader        <chr [1]>           1
##  5 Leia Organa        <NULL>              0
##  6 Owen Lars          <NULL>              0
##  7 Beru Whitesun lars <NULL>              0
##  8 R5-D4              <NULL>              0
##  9 Biggs Darklighter  <chr [1]>           1
## 10 Obi-Wan Kenobi     <chr [5]>           5
## # … with 77 more rows
```



and

Tidy data from nested lists

The recent version of `tidyr` have added several functions that are designed to aide in the tidying of heirachical data. Since they are part of `tidyr` all of the following functions work with data frames.
From `tidyr`

> `hoist()`, `unnest_longer()`, and `unnest_wider()` provide tools for rectangling, collapsing deeply nested lists into regular columns.

```
(d = tibble(people=sw_people))
```

```
## # A tibble: 87 x 1
##    people
##    <list>
##  1 <named list [16]>
##  2 <named list [14]>
##  3 <named list [14]>
##  4 <named list [15]>
##  5 <named list [15]>
##  6 <named list [14]>
##  7 <named list [14]>
##  8 <named list [14]>
##  9 <named list [15]>
## 10 <named list [16]>
## # … with 77 more rows
```

```
unnest_wider(d, people)
```

```
## # A tibble: 87 x 16
##    name  height mass  hair_color skin_color eye_color birth_year gender
##    <chr> <chr>  <chr> <chr>      <chr>      <chr>     <chr>      <chr>
##  1 Luke… 172    77    blond      fair       blue      19BBY      male
##  2 C-3PO 167    75    n/a        gold       yellow    112BBY     n/a
##  3 R2-D2 96     32    n/a        white, bl… red       33BBY      n/a
##  4 Dart… 202    136   none       white      yellow    41.9BBY    male
##  5 Leia… 150    49    brown      light      brown     19BBY      female
##  6 Owen… 178    120   brown, gr… light      blue      52BBY      male
##  7 Beru… 165    75    brown      light      blue      47BBY      female
##  8 R5-D4 97     32    n/a        white, red red       unknown    n/a
##  9 Bigg… 183    84    black      light      brown     24BBY      male
## 10 Obi-… 182    77    auburn, w… fair       blue-gray 57BBY      male
## # … with 77 more rows, and 8 more variables: homeworld <chr>, films <list>,
## #   species <chr>, vehicles <list>, starships <list>, created <chr>,
## #   edited <chr>, url <chr>
```

```
unnest_longer(d, people)
```

```
## # A tibble: 1,244 x 2
##    people    people_id
##    <list>    <chr>
##  1 <chr [1]> name
##  2 <chr [1]> height
##  3 <chr [1]> mass
##  4 <chr [1]> hair_color
##  5 <chr [1]> skin_color
##  6 <chr [1]> eye_color
##  7 <chr [1]> birth_year
##  8 <chr [1]> gender
##  9 <chr [1]> homeworld
## 10 <chr [5]> films
## # … with 1,234 more rows
```

```
unnest_wider(d, people) %>%
  select(name, starships) %>%
  unnest_longer(starships, )
```

```
## # A tibble: 98 x 2
##    name               starships
##    <chr>              <chr>
##  1 Luke Skywalker     http://swapi.co/api/starships/12/
##  2 Luke Skywalker     http://swapi.co/api/starships/22/
##  3 C-3PO              NA
##  4 R2-D2              NA
##  5 Darth Vader        http://swapi.co/api/starships/13/
##  6 Leia Organa        NA
##  7 Owen Lars          NA
##  8 Beru Whitesun lars NA
##  9 R5-D4              NA
## 10 Biggs Darklighter  http://swapi.co/api/starships/12/
## # … with 88 more rows
```

Acknowledgments

Acknowledgments

Above materials are derived in part from the following sources:

- Hadley Wickham - Adv-R Functionals
- Hadley Wickham - R for Data Science
- Neil Saunders - A brief introduction to "apply" in R
- Jenny Bryan - Purrr Tutorial
- R Language Definition