

Functional programming & purrr

Lecture 09

Dr. Colin Rundel

Rectangling

Star Wars & repurrrsive

`repurrrsive` is a package that contains a number of interesting example data sets that are stored in a hierarchical format. Many come from web-based APIs which provide results as JSON.

```
1 str(repurrrsive::sw_people)
```

List of 87

\$: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/api/films/2/" "http://swapi.co/api/films/10/" "http://swapi.co/api/films/1/"
..$ 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/"
```

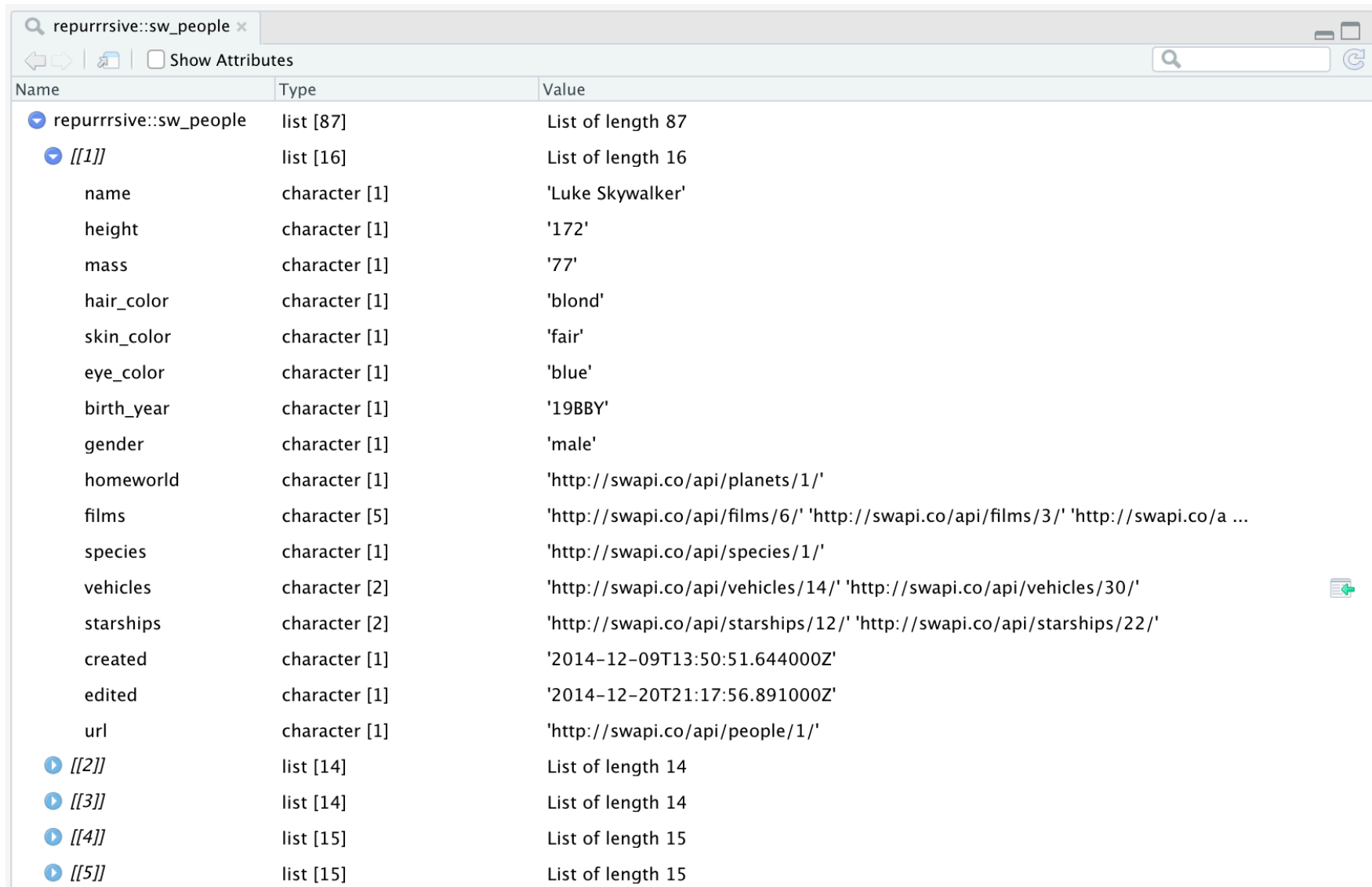
\$:List of 14

```
..$ name      : chr "C-3P0"
..$ height    : chr "167"
```

```
..$ mass      : chr "75"  
$ hair_color: chr "n/a"
```

RStudio data viewer

```
1 View(repurrrsive::sw_people)
```



Name	Type	Value
repurrrsive::sw_people	list [87]	List of length 87
[[1]]	list [16]	List of length 16
name	character [1]	'Luke Skywalker'
height	character [1]	'172'
mass	character [1]	'77'
hair_color	character [1]	'blond'
skin_color	character [1]	'fair'
eye_color	character [1]	'blue'
birth_year	character [1]	'19BBY'
gender	character [1]	'male'
homeworld	character [1]	'http://swapi.co/api/planets/1/'
films	character [5]	'http://swapi.co/api/films/6/' 'http://swapi.co/api/films/3/' 'http://swapi.co/a ...
species	character [1]	'http://swapi.co/api/species/1/'
vehicles	character [2]	'http://swapi.co/api/vehicles/14/' 'http://swapi.co/api/vehicles/30/'
starships	character [2]	'http://swapi.co/api/starships/12/' 'http://swapi.co/api/starships/22/'
created	character [1]	'2014-12-09T13:50:51.644000Z'
edited	character [1]	'2014-12-20T21:17:56.891000Z'
url	character [1]	'http://swapi.co/api/people/1/'
[[2]]	list [14]	List of length 14
[[3]]	list [14]	List of length 14
[[4]]	list [15]	List of length 15
[[5]]	list [15]	List of length 15

List columns

We can make `sw_people` into a data frame by treating the original list as a single column in a data frame.

```
1 (sw_df = tibble::tibble(  
2   people = repurrrsive::sw_people  
3 ))
```

```
# A tibble: 87 × 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]>  
# i 77 more rows
```

```
1 as.data.frame(sw_df) |> head()
```

```
1 Luke Skywalker, 172, 77, blond, fair, bl  
2  
3  
4  
5  
6
```

Unnesting

```
1 sw_df |>
2   unnest_wider(people)
```

```
# A tibble: 87 × 16
```

	name	height	mass	hair_color	skin_color	eye_color	birth_year
	<chr>	<chr>	<chr>	<chr>	<chr>	<chr>	<chr>
1	Luke Skywa...	172	77	blond	fair	blue	19BBY
2	C-3P0	167	75	n/a	gold	yellow	112BBY
3	R2-D2	96	32	n/a	white, bl...	red	33BBY
4	Darth Vader	202	136	none	white	yellow	41.9BBY
5	Leia Organa	150	49	brown	light	brown	19BBY
6	Owen Lars	178	120	brown, gr...	light	blue	52BBY
7	Beru White...	165	75	brown	light	blue	47BBY
8	R5-D4	97	32	n/a	white, red	red	unknown
9	Biggs Dark...	183	84	black	light	brown	24BBY
10	Obi-Wan Ke...	182	77	auburn, w...	fair	blue-gray	57BBY

```
# i 77 more rows
```

```
# i 9 more variables: gender <chr>, homeworld <chr>, films <list>,  
# species <chr>, vehicles <list>, starships <list>, created <chr>,  
# edited <chr>, url <chr>
```

Unnesting - column types

```
1 sw_df |>  
2   unnest_wider(people) |>  
3   pull(height)
```

[1]	"172"	"167"	"96"	"202"	"150"	"178"
[7]	"165"	"97"	"183"	"182"	"188"	"180"
[13]	"228"	"180"	"173"	"175"	"170"	"180"
[19]	"66"	"170"	"183"	"200"	"190"	"177"
[25]	"175"	"180"	"150"	"unknown"	"88"	"160"
[31]	"193"	"191"	"170"	"196"	"224"	"206"
[37]	"183"	"137"	"112"	"183"	"163"	"175"
[43]	"180"	"178"	"94"	"122"	"163"	"188"
[49]	"198"	"196"	"171"	"184"	"188"	"264"
[55]	"188"	"196"	"185"	"157"	"183"	"183"
[61]	"170"	"166"	"165"	"193"	"191"	"183"
[67]	"168"	"198"	"229"	"213"	"167"	"79"
[73]	"96"	"193"	"191"	"178"	"216"	"234"
[79]	"188"	"178"	"206"	"unknown"	"unknown"	"unknown"
[85]	"unknown"	"unknown"	"165"			

More list columns

Unnest Longer

```
1 unnest_wider(sw_df, people) |>
2   select(name, starships) |>
3   unnest_longer(starships)
```

A tibble: 31 × 2

	name <chr>	starships <chr>
1	Luke Skywalker	http://swapi.co/api/starships/12/
2	Luke Skywalker	http://swapi.co/api/starships/22/
3	Darth Vader	http://swapi.co/api/starships/13/
4	Biggs Darklighter	http://swapi.co/api/starships/12/
5	Obi-Wan Kenobi	http://swapi.co/api/starships/48/
6	Obi-Wan Kenobi	http://swapi.co/api/starships/59/
7	Obi-Wan Kenobi	http://swapi.co/api/starships/64/
8	Obi-Wan Kenobi	http://swapi.co/api/starships/65/
9	Obi-Wan Kenobi	http://swapi.co/api/starships/74/
10	Anakin Skywalker	http://swapi.co/api/starships/59/

i 21 more rows

General advice

- If there is a consistent set of entries (usually named) in the list column, use `unnest_wider()`
- If there are different numbers of entries (often unnamed) in the list column, use `unnest_longer()`
- Never use just `unnest()` - it can be inconsistent depending on input data
- Think about if you need all the data or not - `unnest_*` are not always the best choice (more on the `hoist()` shortly)

Functional Programming

Functions as objects

We have mentioned in passing that in R functions are treated as 1st class objects (like vectors), meaning they can be assigned names, stored in lists, passed as arguments, etc.

```
1 f = function(x) {  
2   x*x  
3 }  
4 f(2)
```

```
[1] 4
```

```
1 g = f  
2 g(2)
```

```
[1] 4
```

```
1 l = list(f = f, g = g)  
2 l$f(3)
```

```
[1] 9
```

```
1 l[[2]](4)
```

```
[1] 16
```

```
1 l[1](3)
```

```
Error: attempt to apply non-function
```

Functions as arguments

We can pass in functions as arguments to other functions,

```
1 do_calc = function(v, func) {  
2   func(v)  
3 }
```

```
1 do_calc(1:3, sum)
```

```
[1] 6
```

```
1 do_calc(1:3, mean)
```

```
[1] 2
```

```
1 do_calc(1:3, sd)
```

```
[1] 1
```

Anonymous functions

These are short functions that are created without ever assigning a name,

```
1 function(x) {x+1}
```

```
function (x)
{
  x + 1
}
```

```
1 (function(y) {y-1})(10)
```

```
[1] 9
```

this can be particularly helpful for implementing certain types of tasks,

```
1 integrate(function(x) x, 0, 1)
```

```
0.5 with absolute error < 5.6e-15
```

```
1 integrate(function(x) x^2-2*x+1, 0, 1)
```

```
0.3333333 with absolute error < 3.7e-15
```

Base R - anonymous function shorthand

Along with the base pipe (`|>`), R v4.1.0 introduced a shortcut for anonymous functions using `\()`,

```
1 (\(x) {1+x})(1:5)
```

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

```
1 (function(x) {1+x})(1:5)
```

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

```
1 (\(x) x^2)(10)
```

```
[1] 100
```

```
1 (function(x) x^2)(10)
```

```
[1] 100
```

```
1 integrate(\(x) sin(x)^2, 0, 1)
```

```
0.2726756 with absolute error < 3e-15
```

```
1 integrate(function(x) sin(x)^2, 0, 1)
```

```
0.2726756 with absolute error < 3e-15
```

We can use this with the base pipe to avoid using `_`,

```
1 data.frame(x = runif(10), y = runif(10)) |>  
2   (\(d) lm(y~x, data = d))()
```

```
Call:  
lm(formula = y ~ x, data = d)
```

```
Coefficients:  
(Intercept)          x  
    0.66606      -0.09099
```


apply (base R)

Apply functions

The apply functions are a collection of tools for functional programming in base R, they are variations of the `map` function found in many other languages and apply a function over the elements of an input (vector).

```
1 ??base::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 (Aliases: lapply, sa
## 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
```

for loops vs *apply functions

Of course, someone has to write loops. It doesn't have to be you.

— Jenny Bryan

From dplyr's Row-wise operations

R's apply functions are not any faster than `for` loops, but they are often more concise and easier to read.

The chief advantage of using the apply functions is that they generally take care of the bookkeeping of collecting results and assembling them into a useful format, and doing so in a way that avoids common pitfalls (e.g. appending to an existing object).

::: {.aside} Vectorization will always be much much faster than either `for` loops or `*apply` functions. :::

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`.

```
1 lapply(1:8, sqrt) |>  
2   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
```

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

List of 8

```
$ : num 4  
$ : num 9  
$ : num 16  
$ : num 25  
$ : num 36  
$ : num 49  
$ : num 64  
$ : num 81
```

Argument matching

```
1 lapply(  
2   1:8, function(x, pow) x^pow, pow=3  
3 ) |>  
4   str()
```

List of 8

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

```
1 lapply(  
2   1:8, function(x, pow) x^pow, x=2  
3 ) |>  
4   str()
```

List of 8

```
$ : num 2  
$ : num 4  
$ : num 8  
$ : num 16  
$ : num 32  
$ : num 64  
$ : num 128  
$ : num 256
```

supply

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

`supply` 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.

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

```
[1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751  
[8] 2.828427
```

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

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

```
1 supply(1:8, function(x) c(x, x^2, x^3))
```

	[,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

Length mismatch?

```
1 sapply(1:6, seq) |> str()
```

List of 6

```
$ : int 1  
$ : int [1:2] 1 2  
$ : int [1:3] 1 2 3  
$ : int [1:4] 1 2 3 4  
$ : int [1:5] 1 2 3 4 5  
$ : int [1:6] 1 2 3 4 5 6
```

```
1 lapply(1:6, seq) |> str()
```

List of 6

```
$ : int 1  
$ : int [1:2] 1 2  
$ : int [1:3] 1 2 3  
$ : int [1:4] 1 2 3 4  
$ : int [1:5] 1 2 3 4 5  
$ : int [1:6] 1 2 3 4 5 6
```

Type mismatch?

```
1 l = list(a = 1:3, b = 4:6, c = 7:9, d = list(10, 11, "A"))
```

```
1 apply(l, function(x) x[1]) |> str()
```

List of 4

```
$ a: int 1  
$ b: int 4  
$ c: int 7  
$ d: num 10
```

```
1 apply(l, function(x) x[[1]]) |> str()
```

Named num [1:4] 1 4 7 10

```
- attr(*, "names")= chr [1:4] "a" "b" "c" "d"
```

```
1 apply(l, function(x) x[[3]]) |> str()
```

Named chr [1:4] "3" "6" "9" "A"

```
- attr(*, "names")= chr [1:4] "a" "b" "c" "d"
```


*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.

```
1 df = data.frame(  
2   a = 1:6,  
3   b = letters[1:6],  
4   c = c(TRUE, FALSE)  
5 )
```

```
1 lapply(df, class) |> str()
```

List of 3

```
$ a: chr "integer"  
$ b: chr "character"  
$ c: chr "logical"
```

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

a	b	c
"integer"	"character"	"logical"

A more useful example

Some sources of data (e.g. some US government agencies) will encode missing values with `-999`, if want to replace these with `NA`s `lapply` is not a bad choice.

```
1 d = tibble::tribble(  
2   ~patient_id, ~age, ~bp, ~o2,  
3     1,    32,   110,   97,  
4     2,    27,   100,   95,  
5     3,    56,   125, -999,  
6     4,    19, -999, -999,  
7     5,    65, -999,   99  
8 )
```

```
1 fix_missing = function(x) {  
2   x[x == -999] = NA  
3   x  
4 }  
5 lapply(d, fix_missing) |> str()
```

List of 4

```
$ patient_id: num [1:5] 1 2 3 4 5  
$ age       : num [1:5] 32 27 56 19 65  
$ bp        : num [1:5] 110 100 125 NA NA  
$ o2        : num [1:5] 97 95 NA NA 99
```

```
1 lapply(d, fix_missing) |>  
2   as_tibble()
```

```
# A tibble: 5 × 4  
  patient_id age    bp    o2  
    <dbl> <dbl> <dbl> <dbl>  
1         1   32   110   97  
2         2   27   100   95  
3         3   56   125  NA  
4         4   19    NA  NA  
5         5   65    NA   99
```

dplyr alternative

dplyr is also a viable option here using the `across()` helper,

```
1 d |>
2   mutate(
3     across(
4       bp:o2,
5       fix_missing
6     )
7   )
```

```
# A tibble: 5 × 4
  patient_id  age    bp    o2
  <dbl> <dbl> <dbl> <dbl>
1         1   32  110   97
2         2   27  100   95
3         3   56  125  NA
4         4   19   NA   NA
5         5   65   NA   99
```

```
1 d |>
2   mutate(
3     across(
4       where(is.numeric),
5       fix_missing
6     )
7   )
```

```
# A tibble: 5 × 4
  patient_id  age    bp    o2
  <dbl> <dbl> <dbl> <dbl>
1         1   32  110   97
2         2   27  100   95
3         3   56  125  NA
4         4   19   NA   NA
5         5   65   NA   99
```

other less common apply functions

- `apply()` - applies a function over the rows or columns of a matrix or array (data frames also work but are bad idea)
- `vapply()` - is similar to `sapply`, but has a enforced return type and size
- `mapply()` - like `sapply` but will iterate over multiple vectors at the same time.
- `rapply()` - a recursive version of `lapply`, behavior depends largely on the `how` argument
- `eapply()` - apply a function over an environment.



Map functions

Basic functions for looping over objects and returning a value (of a specific type) - type consistent replacements for `lapply/sapply/vapply`.

- `map()` - returns a list, equivalent to `lapply()`
- `map_lgl()` - returns a logical vector.
- `map_int()` - returns a integer vector.
- `map_dbl()` - returns a double vector.
- `map_chr()` - returns a character vector.
- `walk()` - returns nothing, used for side effects

Type Consistency

R is a weakly / dynamically typed language which means there is no syntactic way to define a function which enforces 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.

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

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

```
[1] 0.01926096 0.00210620 -0.02243359
```

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

```
Warning: Automatic coercion from double to character
1.0.0.
```

```
i Please use an explicit call to `as.character()`
instead.
```

```
[1] "0.019261" "0.002106" "-0.022434"
```

```
1 map(x, mean) |> str()
```

```
List of 3
```

```
$ : num 0.0193
$ : num 0.00211
$ : num -0.0224
```

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

```
Error in `map_int()`:
```

```
i In index: 1.
```

```
Caused by error:
```

```
! Can't coerce from a number to an integer.
```

```
1 lapply(x, mean) |> str()
```

```
List of 3
```

```
$ : num 0.0193
$ : num 0.00211
$ : num -0.0224
```

Working with Data Frames

purrr offers the functions `map_dfr` and `map_dfc` (which were superseded as of v1.0.0) - these allow for the construction of a data frame by row or by column respectively.

```
1 d = tibble::tribble(  
2   ~patient_id, ~age, ~bp, ~o2,  
3     1,    32,   110,   97,  
4     2,    27,   100,   95,  
5     3,    56,   125, -999,  
6     4,    19, -999, -999,  
7     5,    65, -999,   99  
8 )
```

```
1 fix_missing = function(x) {  
2   x[x == -999] = NA  
3   x  
4 }
```

```
1 purrr::map_dfc(d, fix_missing)
```

A tibble: 5 × 4

	patient_id	age	bp	o2
	<dbl>	<dbl>	<dbl>	<dbl>
1	1	32	110	97
2	2	27	100	95
3	3	56	125	NA
4	4	19	NA	NA
5	5	65	NA	99

```
1 purrr::map(d, fix_missing) |>  
2   bind_cols()
```

A tibble: 5 × 4

	patient_id	age	bp	o2
	<dbl>	<dbl>	<dbl>	<dbl>
1	1	32	110	97
2	2	27	100	95
3	3	56	125	NA
4	4	19	NA	NA
5	5	65	NA	99

Building by row

```
1 map(sw_people, function(x) x[1:5]) |> bind_rows()
```

```
# A tibble: 87 × 5
```

	name <chr>	height <chr>	mass <chr>	hair_color <chr>	skin_color <chr>
1	Luke Skywalker	172	77	blond	fair
2	C-3P0	167	75	n/a	gold
3	R2-D2	96	32	n/a	white, blue
4	Darth Vader	202	136	none	white
5	Leia Organa	150	49	brown	light
6	Owen Lars	178	120	brown, grey	light
7	Beru Whitesun lars	165	75	brown	light
8	R5-D4	97	32	n/a	white, red
9	Biggs Darklighter	183	84	black	light
10	Obi-Wan Kenobi	182	77	auburn, white	fair

```
# i 77 more rows
```

```
1 map(sw_people, function(x) x) |> bind_rows()
```

```
Error in `vctrs::data_frame()`:  
! Can't recycle `name` (size 5) to match `vehicles` (size 2).
```

purrr style anonymous functions

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

```
1 map_dbl(1:5, function(x) x/(x+1))
```

```
[1] 0.5000000 0.6666667 0.7500000 0.8000000 0.8333333
```

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

```
[1] 0.5000000 0.6666667 0.7500000 0.8000000 0.8333333
```

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

```
[1] 0.5000000 0.6666667 0.7500000 0.8000000 0.8333333
```

Generally, the latter option is preferred to avoid confusion with magrittr.

Multiargument anonymous functions

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

```
1 map2_dbl(1:5, 1:5, function(x,y) x / (y+1))
```

```
[1] 0.5000000 0.6666667 0.7500000 0.8000000 0.8333333
```

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

```
[1] 0.5000000 0.6666667 0.7500000 0.8000000 0.8333333
```

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

```
[1] 0.5000000 0.6666667 0.7500000 0.8000000 0.8333333
```

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

```
[1] "Aa" "Bb" "Cc" "Dd" "Ee"
```

Indexed maps

purrr also has a collection of `imap` prefixed functions which are short hand for mapping over an object and the indexes of that object (i.e. `seq_along(obj)`).

```
1 imap(  
2   letters[1:5],  
3   ~cat("index: ", .y, ", value: ", .x, "\n", sep="")  
4 )
```

```
index: 1, value: a  
index: 2, value: b  
index: 3, value: c  
index: 4, value: d  
index: 5, value: e
```

Lookups

Very often we want to extract only certain values by name or position from a list, `purrr` provides a shorthand for this operation - instead of a function you can provide either a character or numeric vector, those values will be used to sequentially subset the elements being iterated.

```
1 purrr::map_chr(sw_people, "name") |> head()
```

```
[1] "Luke Skywalker" "C-3P0"      "R2-D2"  
[4] "Darth Vader"    "Leia Organa" "Owen Lars"
```

```
1 purrr::map_chr(sw_people, 1) |> head()
```

```
[1] "Luke Skywalker" "C-3P0"      "R2-D2"  
[4] "Darth Vader"    "Leia Organa" "Owen Lars"
```

```
1 purrr::map_chr(sw_people, list("films", 1)) |> head(n=10)
```

```
[1] "http://swapi.co/api/films/6/" "http://swapi.co/api/films/5/"  
[3] "http://swapi.co/api/films/5/" "http://swapi.co/api/films/6/"  
[5] "http://swapi.co/api/films/6/" "http://swapi.co/api/films/5/"  
[7] "http://swapi.co/api/films/5/" "http://swapi.co/api/films/1/"  
[9] "http://swapi.co/api/films/1/" "http://swapi.co/api/films/5/"
```

Length coercion?

```
1 purrr::map_chr(sw_people, list("starships", 1))
```

```
Error in `purrr::map_chr()` :  
i In index: 2.  
Caused by error:  
! Result must be length 1, not 0.
```

```
1 sw_people[[2]]$name
```

```
[1] "C-3P0"
```

```
1 sw_people[[2]]$starships
```

```
NULL
```

```
1 purrr::map_chr(sw_people, list("starships", 1), .default = NA) |> head(4)
```

```
[1] "http://swapi.co/api/starships/12/"  
[2] NA  
[3] NA  
[4] "http://swapi.co/api/starships/13/"
```

```
1 purrr::map(sw_people, list("starships", 1)) |> head() |> str()
```

```
List of 6  
 $ : chr "http://swapi.co/api/starships/12/"  
 $ : NULL  
 $ : NULL  
 $ : chr "http://swapi.co/api/starships/13/"  
 $ : NULL  
 $ : NULL
```

list columns

```
1 (chars = tibble(  
2   name = purrr::map_chr(  
3     sw_people, "name"  
4   ),  
5   starships = purrr::map(  
6     sw_people, "starships"  
7   )  
8 ))
```

A tibble: 87 × 2

	name	starships
	<chr>	<list>
1	Luke Skywalker	<chr [2]>
2	C-3P0	<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]>

i 77 more rows

```
1 chars |>  
2   mutate(  
3     n_starships = map_int(  
4       starships, length  
5     )  
6   )
```

A tibble: 87 × 3

	name	starships	n_starships
	<chr>	<list>	<int>
1	Luke Skywalker	<chr [2]>	2
2	C-3P0	<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

i 77 more rows

Complex heirarchical data

Often we may encounter complex data structures where our goal is not to rectangle every value (which may not even be possible) but rather to rectangle a small subset of the data.

```
1 str(repurrrsive::discog, max.level = 3)
```

List of 155

\$:List of 5

..\$ instance_id : int 354823933

..\$ date_added : chr "2019-02-16T17:48:59-08:00"

..\$ basic_information:List of 11

.. ..\$ labels :List of 1

.. ..\$ year : int 2015

.. ..\$ master_url : NULL

.. ..\$ artists :List of 1

.. ..\$ id : int 7496378

.. ..\$ thumb : chr "https://img.discogs.com/vEVegHrMNTsP6xG_K60uFXz4h_U=/fit-in/150x150/f:

.. ..\$ title : chr "Demo"

.. ..\$ formats :List of 1

.. ..\$ cover_image : chr "https://img.discogs.com/EmbMh7vsElksjRgoXLFSuY1sjRQ=/fit-in/500x499/f:

.. ..\$ resource_url: chr "https://api.discogs.com/releases/7496378"

.. ..\$ master_id : int 0

..\$ id : int 7496378

..\$ rating : int 0

\$:List of 5

..\$ instance_id : int 354092601

..\$ date_added : chr "2019-02-13T14:13:11-08:00"

Partial vs complete rectangling

In the case of `discog` we may want to rectangle the `id`, `year`, `title`, `artist`, and `label` fields but leave the rest of the data as is.

In cases like this using tidyr's `unnest_wider()` and `unnest_long()` is not ideal as they will attempt to rectangle the entire data set when we only want a subset. There is no need to do the expensive work of unnesting columns we will never use.

purrr's `map_*()` functions and tidyr's `hoist()` function are useful for targeting specific columns to rectangle.

purrr

```
1 tibble(disc = repurrrsive::discog) |>
2   mutate(
3     id      = purrr::map_int(disc, "id"),
4     year    = purrr::map_int(disc, c("basic_information", "year")),
5     title   = purrr::map_chr(disc, c("basic_information", "title")),
6     artist  = purrr::map_chr(disc, list("basic_information", "artists", 1, "name")),
7     label   = purrr::map_chr(disc, list("basic_information", "labels", 1, "name"))
8   )
```

A tibble: 155 × 6

	disc		id	year	title	artist	label
	<list>		<int>	<int>	<chr>	<chr>	<chr>
1	<named list [5]>		7496378	2015	Demo	Mollot	Tobi...
2	<named list [5]>		4490852	2013	Observant Com El Mon ...	Una B...	La V...
3	<named list [5]>		9827276	2017	I	S.H.I...	La V...
4	<named list [5]>		9769203	2017	Oído Absoluto	Rata ...	La V...
5	<named list [5]>		7237138	2015	A Cat's Cause, No Dog...	Ivy (...)	Kato...
6	<named list [5]>		13117042	2019	Tashme	Tashme	High...
7	<named list [5]>		7113575	2014	Demo	Desgr...	Mind...
8	<named list [5]>		10540713	2015	Let The Miracles Begin	Phant...	Not ...
9	<named list [5]>		11260950	2017	Sub Space	Sub S...	Not ...
10	<named list [5]>		11726853	2017	Demo	Small...	Pres...

i 145 more rows

hoist()

```
1 tibble(disc = repurrrsive::discog) %>%
2   hoist(
3     disc,
4     id = "id",
5     year = c("basic_information", "year"),
6     title = c("basic_information", "title"),
7     artist = list("basic_information", "artists", 1, "name"),
8     label = list("basic_information", "labels", 1, "name")
9   )
```

A tibble: 155 × 6

	id	year	title	artist	label	disc
	<int>	<int>	<chr>	<chr>	<chr>	<list>
1	7496378	2015	Demo	Mollot	Tobi...	<named list>
2	4490852	2013	Observant Com El Mon Es D...	Una B...	La V...	<named list>
3	9827276	2017	I	S.H.I...	La V...	<named list>
4	9769203	2017	Oído Absoluto	Rata ...	La V...	<named list>
5	7237138	2015	A Cat's Cause, No Dogs Pr...	Ivy (...)	Kato...	<named list>
6	13117042	2019	Tashme	Tashme	High...	<named list>
7	7113575	2014	Demo	Desgr...	Mind...	<named list>
8	10540713	2015	Let The Miracles Begin	Phant...	Not ...	<named list>
9	11260950	2017	Sub Space	Sub S...	Not ...	<named list>
10	11726853	2017	Demo	Small...	Pres...	<named list>

i 145 more rows

Example

List columns and approximating pi