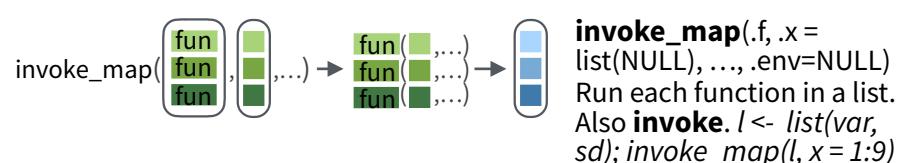
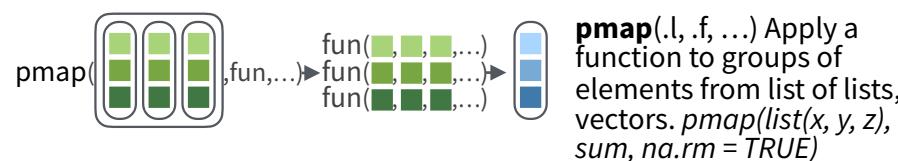
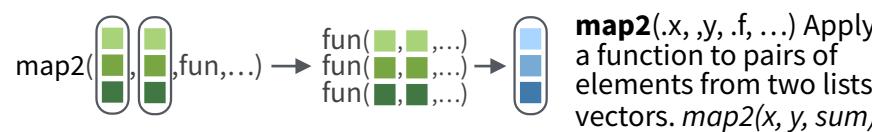
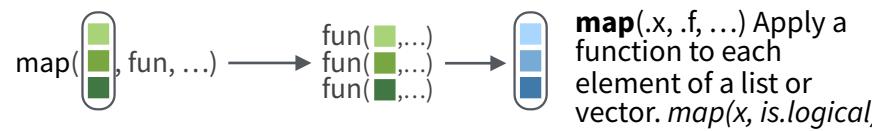


Apply functions with purrr :: CHEAT SHEET



Apply Functions

Map functions apply a function iteratively to each element of a list or vector.



lmap(.x, .f, ...) Apply function to each list-element of a list or vector.
imap(.x, .f, ...) Apply .f to each element of a list or vector and its index.

OUTPUT

map(), **map2()**, **pmap()**, **imap** and **invoke_map** each return a list. Use a suffixed version to return the results as a specific type of flat vector, e.g. **map2_chr**, **pmap_lgl**, etc.

Use **walk**, **walk2**, and **pwalk** to trigger side effects. Each return its input invisibly.

SHORTCUTS - within a purrr function:

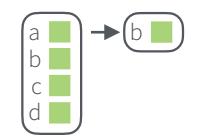
"**name**" becomes **function(x) x\$name**. e.g. `map(l, "a")` extracts \$a from each element of l

~.x becomes **function(x) x**. e.g. `map(l, ~.x + y)` becomes `map(l, p, function(l, p) l + p)`

~.1 .. 2 etc becomes **function(..1, ..2, etc) ..1 .. 2** etc e.g. `pmap(list(a, b, c), ~..3 + ..1 - ..2)` becomes `pmap(list(a, b, c), function(a, b, c) c + a - b)`

Work with Lists

FILTER LISTS



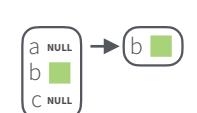
pluck(.x, ..., .default=NULL) Select an element by name or index, `pluck(x, "b")`, or its attribute with **attr_getter**. `pluck(x, "b", attr_getter("n"))`



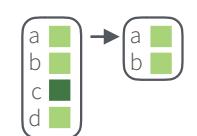
keep(.x, .p, ...) Select elements that pass a logical test. `keep(x, is.na)`



discard(.x, .p, ...) Select elements that do not pass a logical test. `discard(x, is.na)`

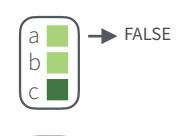


compact(.x, .p = identity) Drop empty elements. `compact(x)`

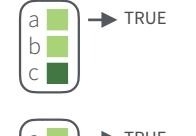


head_while(.x, .p, ...) Return head elements until one does not pass. Also **tail_while**. `head_while(x, is.character)`

SUMMARISE LISTS



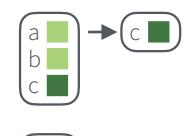
every(.x, .p, ...) Do all element pass a test? `every(x, is.character)`



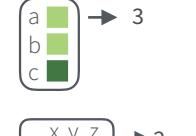
some(.x, .p, ...) Do some elements pass a test? `some(x, is.character)`



has_element(.x, .y) Does a list contain an element? `has_element(x, "foo")`



detect(.x, .f, ..., .right=FALSE, .p) Find first element to pass. `detect(x, is.character)`

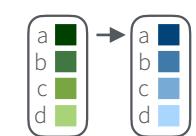


detect_index(.x, .f, ..., .right = FALSE, .p) Find index of first element to pass. `detect_index(x, is.character)`

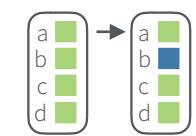


vec_depth(x) Return depth (number of levels of indexes). `vec_depth(x)`

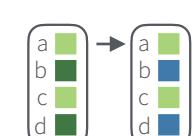
TRANSFORM LISTS



modify(.x, .f, ...) Apply function to each element. Also **map**, **map_chr**, **map_dbl**, **map_dfc**, **map_dfr**, **map_int**, **map_lgl**. `modify(x, ~.+ 2)`



modify_at(.x, .at, .f, ...) Apply function to elements by name or index. Also **map_at**. `modify_at(x, "b", ~.+ 2)`

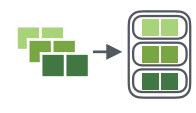


modify_if(.x, .p, .f, ...) Apply function to elements that pass a test. Also **map_if**. `modify_if(x, is.numeric, ~.+ 2)`



modify_depth(.x, .depth, .f, ...) Apply function to each element at a given level of a list. `modify_depth(x, 1, ~.+ 2)`

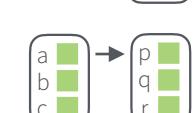
WORK WITH LISTS



array_tree(array, margin = NULL) Turn array into list. Also **array_branch**. `array_tree(x, margin = 3)`

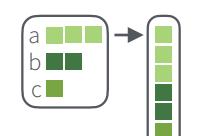


cross2(.x, .y, .filter = NULL) All combinations of .x and .y. Also **cross**, **cross3**, **cross_df**. `cross2(1:3, 4:6)`

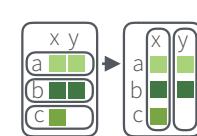


set_names(x, nm = x) Set the names of a vector/list directly or with a function. `set_names(x, c("p", "q", "r"))`
`set_names(x, tolower)`

RESHAPE LISTS

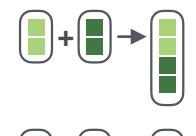


flatten(.x) Remove a level of indexes from a list. Also **flatten_chr**, **flatten_dbl**, **flatten_dfc**, **flatten_dfr**, **flatten_int**, **flatten_lgl**. `flatten(x)`

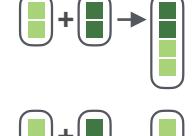


transpose(.l, .names = NULL) Transposes the index order in a multi-level list. `transpose(x)`

JOIN (TO) LISTS



append(x, values, after = length(x)) Add to end of list. `append(x, list(d = 1))`

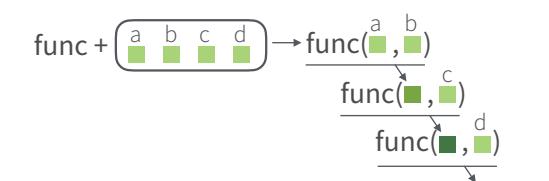


prepend(x, values, before = 1) Add to start of list. `prepend(x, list(d = 1))`

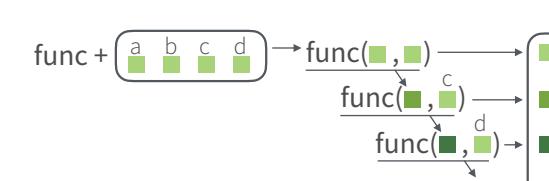


splice(...) Combine objects into a list, storing S3 objects as sub-lists. `splice(x, y, "foo")`

Reduce Lists



reduce(.x, .f, ..., .init) Apply function recursively to each element of a list or vector. Also **reduce_right**, **reduce2**, **reduce2_right**. `reduce(x, sum)`



accumulate(.x, .f, ..., .init) Reduce, but also return intermediate results. Also **accumulate_right**. `accumulate(x, sum)`

compose() Compose multiple functions.

lift() Change the type of input a function takes. Also **lift_dl**, **lift_lv**, **lift_vd**, **lift_vl**.

rerun() Rerun expression n times.

negate() Negate a predicate function (a pipe friendly !)

partial() Partially apply a function, filling in some args.

safely() Modify func to return list of results whenever an error occurs (instead of error).

quietly() Modify function to return list of results, output, messages, warnings.

possibly() Modify function to return default value whenever an error occurs (instead of error).



Nested Data

A **nested data frame** stores individual tables within the cells of a larger, organizing table.

"cell" contents			
Sepal.L	Sepal.W	Petal.L	Petal.W
5.1	3.5	1.4	0.2
4.9	3.0	1.4	0.2
4.7	3.2	1.3	0.2
4.6	3.1	1.5	0.2
5.0	3.6	1.4	0.2

`n_iris$data[[1]]`

nested data frame		Species	data
Species		setosa	<tibble [50 x 4]>
setosa		versicolor	<tibble [50 x 4]>
		virginica	<tibble [50 x 4]>

`n_iris`

Sepal.L	Sepal.W	Petal.L	Petal.W
7.0	3.2	4.7	1.4
6.4	3.2	4.5	1.5
6.9	3.1	4.9	1.5
5.5	2.3	4.0	1.3
6.5	2.8	4.6	1.5

`n_iris$data[[2]]`

Sepal.L	Sepal.W	Petal.L	Petal.W
6.3	3.3	6.0	2.5
5.8	2.7	5.1	1.9
7.1	3.0	5.9	2.1
6.3	2.9	5.6	1.8
6.5	3.0	5.8	2.2

`n_iris$data[[3]]`

Use a nested data frame to:

- preserve relationships between observations and subsets of data
- manipulate many sub-tables at once with the **purrr** functions `map()`, `map2()`, or `pmap()`.

Use a two step process to create a nested data frame:

1. Group the data frame into groups with `dplyr::group_by()`
2. Use `nest()` to create a nested data frame with one row per group

<code>Species S.L S.W P.L P.W</code>	<code>Species S.L S.W P.L P.W</code>
setosa 5.1 3.5 1.4 0.2	setosa 5.1 3.5 1.4 0.2
setosa 4.9 3.0 1.4 0.2	setosa 4.9 3.0 1.4 0.2
setosa 4.7 3.2 1.3 0.2	setosa 4.7 3.2 1.3 0.2
setosa 4.6 3.1 1.5 0.2	setosa 4.6 3.1 1.5 0.2
setosa 5.0 3.6 1.4 0.2	setosa 5.0 3.6 1.4 0.2
versi 7.0 3.2 4.7 1.4	versi 7.0 3.2 4.7 1.4
versi 6.4 3.2 4.5 1.5	versi 6.4 3.2 4.5 1.5
versi 6.9 3.1 4.9 1.5	versi 6.9 3.1 4.9 1.5
versi 5.5 2.3 4.0 1.3	versi 5.5 2.3 4.0 1.3
versi 6.5 2.8 4.6 1.5	versi 6.5 2.8 4.6 1.5
virgini 6.3 3.3 6.0 2.5	virgini 6.3 3.3 6.0 2.5
virgini 5.8 2.7 5.1 1.9	virgini 5.8 2.7 5.1 1.9
virgini 7.1 3.0 5.9 2.1	virgini 7.1 3.0 5.9 2.1
virgini 6.3 2.9 5.6 1.8	virgini 6.3 2.9 5.6 1.8
virgini 6.5 3.0 5.8 2.2	virgini 6.5 3.0 5.8 2.2

`n_iris <- iris %>% group_by(Species) %>% nest()`

`tidy::nest(data, ..., .key = data)`

For grouped data, moves groups into cells as data frames.

Unnest a nested data frame with `unnest()`:

`n_iris %>% unnest()`

`tidy::unnest(data, ..., .drop = NA, .id=NULL, .sep=NULL)`

Unnests a nested data frame.

List Column Workflow

Nested data frames use a **list column**, a list that is stored as a column vector of a data frame. A typical **workflow** for list columns:

1 Make a list column

Species	S.L	S.W	P.L	P.W
setosa	5.1	3.5	1.4	0.2
setosa	4.9	3.0	1.4	0.2
setosa	4.7	3.2	1.3	0.2
setosa	4.6	3.1	1.5	0.2
versi	7.0	3.2	4.7	1.4
versi	6.4	3.2	4.5	1.5
versi	6.9	3.1	4.9	1.5
versi	5.5	2.3	4.0	1.3
versi	6.5	2.8	4.6	1.5
virgini	6.3	3.3	6.0	2.5
virgini	5.8	2.7	5.1	1.9
virgini	7.1	3.0	5.9	2.1
virgini	6.3	2.9	5.6	1.8
virgini	6.5	3.0	5.8	2.2

```
n_iris <- iris %>%  
  group_by(Species) %>%  
  nest()
```

2 Work with list columns

Species	data	model
setosa	<tibble [50x4]>	<S3: lm>
versi	<tibble [50x4]>	<S3: lm>
virgini	<tibble [50x4]>	<S3: lm>

```
mod_fun <- function(df)  
  lm(Sepal.Length ~ ., data = df)
```

```
m_iris <- n_iris %>%  
  mutate(model = map(data, mod_fun))
```

3 Simplify the list column

Species	beta
setos	2.35
versi	1.89
virgini	0.69

```
b_fun <- function(mod)  
  coefficients(mod)[[1]]
```

```
m_iris %>% transmute(Species,  
  beta = map_dbl(model, b_fun))
```

tibble::tribble(...)

Makes list column when needed

```
tribble(~max, ~seq,  
       3, 1:3,  
       4, 1:4,  
       5, 1:5)
```

max	seq
3	<int [3]>
4	<int [4]>
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))
```

2. WORK WITH LIST COLUMNS

- Use the purrr functions `map()`, `map2()`, and `pmap()` to apply a function that returns a result element-wise to the cells of a list column. `walk()`, `walk2()`, and `pwalk()` work the same way, but return a side effect.

purrr::map(.x, .f, ...)

Apply `.f` element-wise to `.x` as `.f(x)`

```
n_iris %>% mutate(n = map(data, dim))
```

purrr::map2(.x, .y, .f, ...)

Apply `.f` element-wise to `.x` and `.y` as `.f(x, y)`

```
m_iris %>% mutate(n = map2(data, model, list))
```

purrr::pmap(.l, .f, ...)

Apply `.f` element-wise to vectors saved in `.l`

```
m_iris %>%  
  mutate(n = pmap(list(data, model, data), list))
```

map(data, fun, ...)

```
map(<tibble [50x4]>, fun, ...)
```

fun(data, ...)

```
fun(<tibble [50x4]>, ...)
```

result

```
result 1
```

result 2

```
result 2
```

result 3

map2(data, model, fun, ...)

```
map2(<tibble [50x4]>, <S3: lm>, fun, ...)
```

fun(data, model, ...)

```
fun(<tibble [50x4]>, <S3: lm>, ...)
```

result

```
result 1
```

result 2

```
result 2
```

result 3

pmap(list(data, model, fun, ...))

```
pmap(list(<tibble [50x4]>, <S3: lm>, <S3: lm>, fun, ...))
```

fun(data, model, fun, ...)