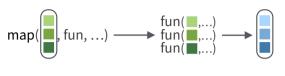
Apply functions with purrr:: cheat sheet



Apply Functions

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



map(.x, .f, ...) Apply a function to each element of a list or vector. *map*(x, is.logical)



map2(.x, ,y, .f, ...) Apply a function to pairs of elements from two lists, vectors. *map2*(*x*, *y*, *sum*)



pmap(.l, .f, ...) Apply a function to groups of elements from list of lists, vectors. pmap(list(x, y, z), sum, na.rm = TRUE)



invoke_map(.f, .x =
list(NULL), ..., env=NULL)
Run each function in a list.
Also invoke. ! <- list(var,
sd); invoke_map(!, x = 1:9)</pre>

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.

function	returns	
map	list	
map_chr	character vector	
map_dbl	double (numeric) vector	
map_dfc	data frame (column bind)	
map_dfr	data frame (row bind)	
map_int	integer vector	
map_lgl	logical vector	
walk	triggers side effects, returns	

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. $map(l, \sim 2 + .x)$ becomes map(l, function(x) 2 + x)



~ .x .y becomes **function(.x, .y) .x .y**, e.g. $map2(l, p, \sim .x +.y)$ becomes map2(l, p, function(l, p) l + p)

the input invisibly

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



С

d 🔣

Drop empty elements. compact(x)

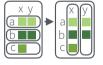
head_while(.x, .p, ...)

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

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)

SUMMARISE LISTS

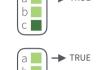


every(.x, .p, ...) Do all
elements pass a test?
every(x, is.character)

some(.x, .p, ...) Do some

elements pass a test?

has_element(x, "foo")



some(x, is.character)
has_element(.x, .y) Does a
list contain an element?



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)*

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")

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)*



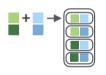
modify_if(.x, .p, .f, ...) Apply function to elements that pass a test. Also **map_if**. *modify_if*(x, is.numeric,~.+2)

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)

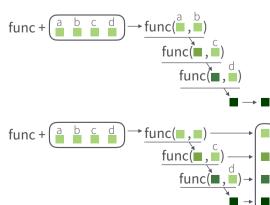


cross2(.x, .y, .filter = NULL) All combinations of .x and .y. Also cross, cross3, cross_df. cross2(1:3, 4:6)



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)

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)

Modify function behavior

compose() Compose multiple functions.

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

rerun() Rerun expression n times.

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

partial() Create a version of a function that has some args preset to values.

safely() Modify func to return list of results and errors. 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.

nested data frame

Species	data			
setosa	<tibble 4]="" [50="" x=""></tibble>			
versicolor	<tibble 4]="" [50="" x=""></tibble>			
virginica <tibble 4]="" [50="" x=""></tibble>				
n iris				

Use a nested data frame to:

- preserve relationships between observations and subsets of data
- manipulate many sub-tables

"cell" contents

1	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]]

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

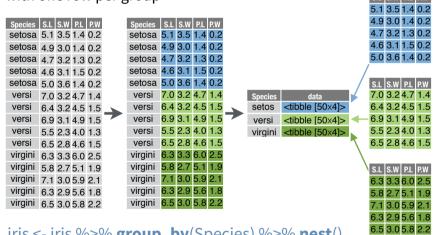
n iris\$data[[3]]

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



versi <tibble [50x4]>

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

tidyr::nest(data, ..., .key = data)

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

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

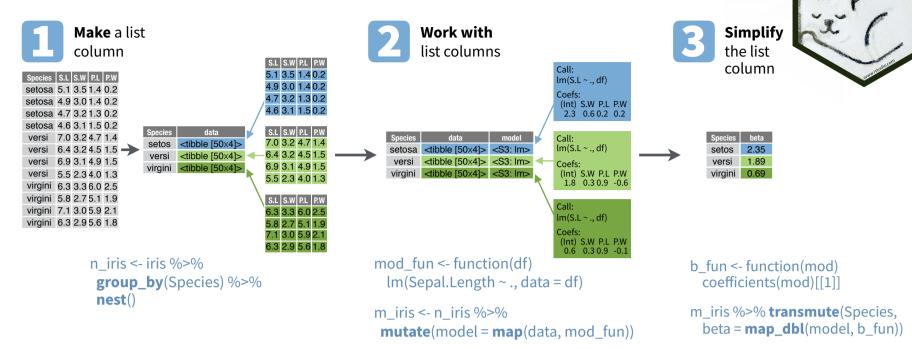
n iris %>% unnest()

tidyr::unnest(data, ..., .drop = NA, .id=NULL, .sep=NULL) Unnests a nested data frame.

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 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

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 - You can create list columns with functions in the tibble and dplyr packages, as well as tidyr's nest()

tibble::tribble(...)

Makes list column when needed

tribble(~max, ~seq, 3, 1:3, 3 <int [3]> 4 <int [4]> 4, 1:4, 5 <int [5]> 1:5)

tibble::tibble(...)

Saves list input as list columns tibble(max = c(3, 4, 5), seq = list(1:3, 1:4, 1:5))

tibble::enframe(x, name="name", value="value") Converts multi-level list to tibble with list cols enframe(list('3'=1:3, '4'=1:4, '5'=1:5), 'max', 'seq')

dplyr::mutate(.data, ...) Also transmute() Returns list col when result returns list.

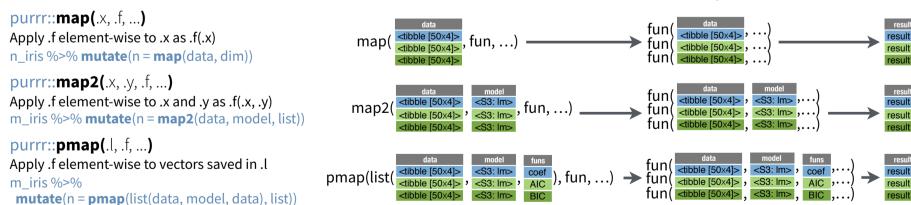
purrr

mtcars %>% mutate(seq = map(cyl, seq))

dplyr::summarise(.data, ...)

Returns list col when result is wrapped with list() mtcars %>% group_by(cyl) %>% summarise(q = list(quantile(mpg)))

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.



3. SIMPLIFY THE LIST COLUMN (into a regular column)

Use the purrr functions map lgl(). map_int(), map_dbl(), map_chr(), as well as tidyr's **unnest()** to reduce a list column into a regular column.

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

Apply .f element-wise to .x, return a logical vector n iris %>% transmute(n = map_lgl(data, is.matrix))

purrr::map_int(.x, .f, ...**)**

Apply .f element-wise to .x, return an integer vector n_iris %>% transmute(n = map_int(data, nrow))

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

Apply .f element-wise to .x, return a double vector n iris %>% transmute(n = map dbl(data, nrow))

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

Apply .f element-wise to .x, return a character vector n iris %>% transmute(n = map_chr(data, nrow))

