

Apply functions with purrr : : CHEATSHEET



Map Functions

ONE LIST

map(.x, .f, ...) Apply a function to each element of a list or vector, and return a list.
`x <- list(a = 1:10, b = 11:20, c = 21:30)`
`l1 <- list(x = c("a", "b"), y = c("c", "d"))`
`map(l1, sort, decreasing = TRUE)`



map_dbl(.x, .f, ...)
 Return a double vector.
`map_dbl(x, mean)`



map_int(.x, .f, ...)
 Return an integer vector.
`map_int(x, length)`



map_chr(.x, .f, ...)
 Return a character vector.
`map_chr(l1, paste, collapse = "")`



map_lgl(.x, .f, ...)
 Return a logical vector.
`map_lgl(x, is.integer)`



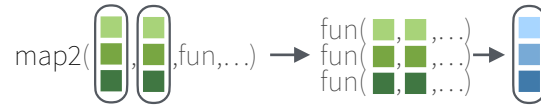
map_vec(.x, .f, ...)
 Return a vector that is of the simplest common type.
`map_vec(l1, paste, collapse = "")`



walk(.x, .f, ...) Trigger side effects, return invisibly.
`walk(x, print)`

TWO LISTS

map2(.x, .y, .f, ...) Apply a function to pairs of elements from two lists or vectors, return a list.
`y <- list(1, 2, 3); z <- list(4, 5, 6); l2 <- list(x = "a", y = "z")`
`map2(x, y, \(x, y) x*y)`



map2_dbl(.x, .y, .f, ...) Return a double vector.
`map2_dbl(y, z, ~.x / .y)`



map2_int(.x, .y, .f, ...) Return an integer vector.
`map2_int(y, z, `+`)`



map2_chr(.x, .y, .f, ...) Return a character vector.
`map2_chr(l1, l2, paste, collapse = ",", sep = ":")`



map2_lgl(.x, .y, .f, ...) Return a logical vector.
`map2_lgl(l2, l1, `%%in%`)`



map2_vec(.x, .f, ...) Return a vector that is of the simplest common type.
`map2_vec(l1, l2, paste, collapse = ",", sep = ":")`



walk2(.x, .y, .f, ...) Trigger side effects, return invisibly.
`walk2(objs, paths, save)`

imap(.x, .f, ...) is shorthand for **map2(.x, names(.x), .f)** or **map2(.x, seq_along(.x), .f)** depending on whether **.x** is named or not.

MANY LISTS

pmap(.l, .f, ...) Apply a function to groups of elements from a list of lists or vectors, return a list.
`pmap(list(x, y, z), function(first, second, third) first * (second + third))`



pmap_dbl(.l, .f, ...) Return a double vector.
`pmap_dbl(list(y, z), ~.x / .y)`



pmap_int(.l, .f, ...) Return an integer vector.
`pmap_int(list(y, z), `+`)`



pmap_chr(.l, .f, ...) Return a character vector.
`pmap_chr(list(l1, l2), paste, collapse = ",", sep = ":")`



pmap_lgl(.l, .f, ...) Return a logical vector.
`pmap_lgl(list(l2, l1), `%%in%`)`



pmap_vec(.l, .f, ...) Return a vector that is of the simplest common type.
`pmap_vec(list(l1, l2), paste, collapse = ",", sep = ":")`



pwalk(.l, .f, ...) Trigger side effects, return invisibly.
`pwalk(list(objs, paths), save)`

Function Shortcuts

Use **\(x)** with functions like **map()** that have single arguments.

map(l, \(x) x + 2)
 becomes
map(l, function(x) x + 2)

Use **\(x, y)** with functions like **map2()** that have two arguments.

map2(l, p, \(x, y) x + y)
 becomes
map2(l, p, function(l, p) l + p)

Use **\(x, y, z)** etc with functions like **pmap()** that have many arguments.

pmap(list(x, y, z), \(x, y, z) x + y / z)
 becomes
pmap(list(x, y, z), function(x, y, z) x * (y + z))

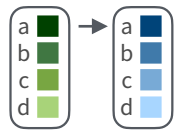
Use **\(x, y)** with functions like **imap()**. **.x** will get the list value and **.y** will get the index, or name if available.

imap(list("a", "b", "c"), \(x, y) paste0(y, ":", x))
 outputs **"index: value"** for each item

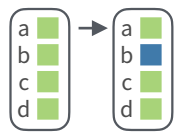
Use a **string** or an **integer** with any map function to index list elements by name or position. **map(l, "name")** becomes **map(l, function(x) x[["name"]])**



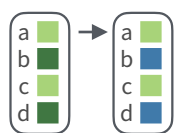
Modify



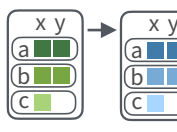
modify(.x, .f, ...) Apply a function to each element. Also **modify2()**, and **imodify()**.
`modify(x, ~.+ 2)`



modify_at(.x, .at, .f, ...) Apply a function to selected elements. Also **map_at()**.
`modify_at(x, "b", ~.+ 2)`



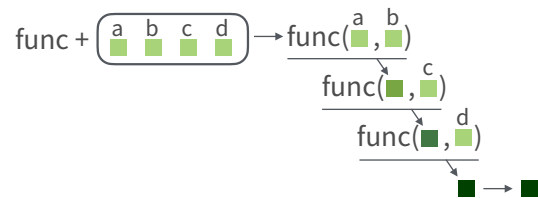
modify_if(.x, .p, .f, ...) Apply a function to elements that pass a test. Also **map_if()**.
`modify_if(x, is.numeric, ~.+ 2)`



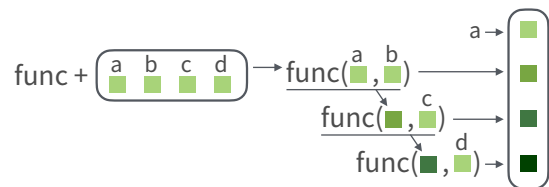
modify_depth(.x, .depth, .f, ...) Apply function to each element at a given level of a list. Also **map_depth()**.
`modify_depth(x, 1, ~.+ 2)`

Reduce

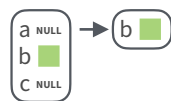
reduce(.x, .f, ..., .init, .dir = c("forward", "backward")) Apply function recursively to each element of a list or vector. Also **reduce2()**.
`reduce(x, sum)`



accumulate(.x, .f, ..., .init) Reduce a list, but also return intermediate results. Also **accumulate2()**.
`accumulate(x, sum)`



Vectors



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



keep_at(x, at) Keep/discard elements based by name or position. Conversely, **discard_at()**.
`keep_at(x, "a")`

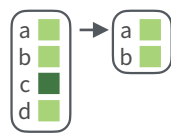


set_names(x, nm = x) Set the names of a vector/list directly or with a function.
`set_names(x, c("p", "q", "r"))`
`set_names(x, tolower)`

Predicate functions



keep(.x, .p, ...) Keep elements that pass a logical test. Conversely, **discard()**.
`keep(x, is.numeric)`



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



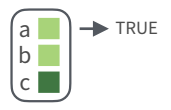
detect(.x, .f, ..., .dir = c("forward", "backward"), .right = NULL, .default = NULL) Find first element to pass.
`detect(x, is.character)`



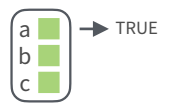
detect_index(.x, .f, ..., .dir = c("forward", "backward"), .right = NULL) Find index of first element to pass.
`detect_index(x, is.character)`



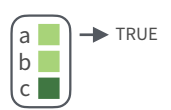
every(.x, .p, ...) Do all elements pass a test?
`every(x, is.character)`



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

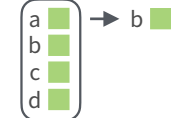


none(.x, .p, ...) Do no elements pass a test?
`none(x, is.character)`

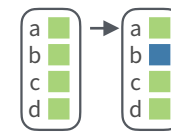


has_element(.x, .y) Does a list contain an element?
`has_element(x, "foo")`

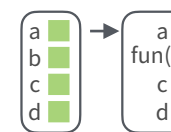
Pluck



pluck(.x, ..., .default=NULL) Select an element by name or index. Also **attr_getter()** and **chuck()**.
`pluck(x, "b")`
`x |> pluck("b")`

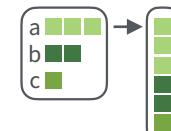


assign_in(x, where, value) Assign a value to a location using pluck selection.
`assign_in(x, "b", 5)`
`x |> assign_in("b", 5)`



modify_in(.x, .where, .f) Apply a function to a value at a selected location.
`modify_in(x, "b", abs)`
`x |> modify_in("b", abs)`

Reshape



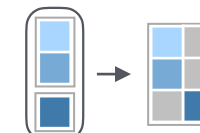
list_flatten(.x) Remove a level of indexes from a list.
`list_flatten(x)`

Concatenate

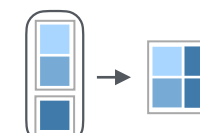
```
x1 <- list(a = 1, b = 2, c = 3)
x2 <- list(
  a = data.frame(x = 1:2),
  b = data.frame(y = "a")
)
```



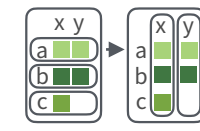
list_c(x) Combines elements into a vector by concatenating them together.
`list_c(x1)`



list_rbind(x) Combines elements into a data frame by row-binding them together.
`list_rbind(x2)`

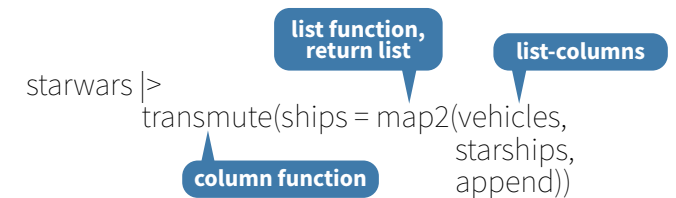


list_cbind(x) Combines elements into a data frame by column-binding them together.
`list_cbind(x2)`

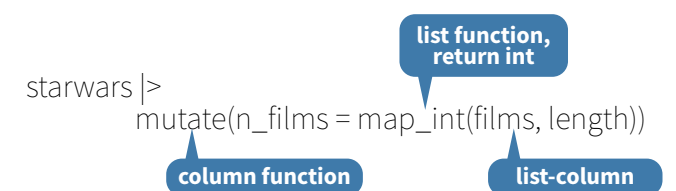


list_transpose(.l, .names = NULL) Transposes the index order in a multi-level list.
`list_transpose(x)`

map(), map2(), or pmap() return lists and will create new list-columns.



Suffixed map functions like **map_int()** return an atomic data type and will **simplify list-columns into regular columns**.



List-Columns

max	seq
3	<int [3]>
4	<int [4]>
5	<int [5]>

List-columns are columns of a data frame where each element is a list or vector instead of an atomic value. Columns can also be lists of data frames. See **tidyr** for more about nested data and list columns.

WORK WITH LIST-COLUMNS

Manipulate list-columns like any other kind of column, using **dplyr** functions like **mutate()**. Because each element is a list, use **map functions** within a column function to manipulate each element.