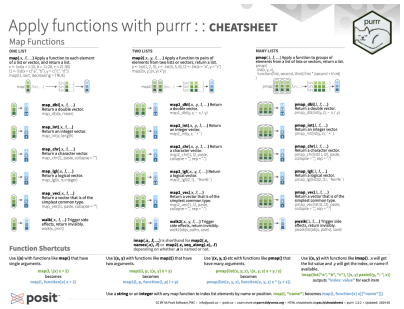
Apply functions with purrr :: Cheatsheet

purrr enhances R's functional programming (FP) toolkit by providing a complete and consistent set of tools for working with functions and vectors. If you've never heard of FP before, the best place to start is the family of `map()` functions which allow you to replace many for loops with code that is both more succinct and easier to read. The best place to learn about the `map()` functions is the iterationchapter in R for Data Science.

Download PDF



Translations (PDF)

- Korean
- Portuguese
- Russian
- Spanish
- Ukrainian
- Vietnamese

```
library(purrr)
```

Map Functions

```
x <- list(a = 1:10, b = 11:20, c = 21:30)
y <- list(1, 2, 3)
z <- list(4, 5, 6)
l1 <- list(x = c("a", "b"), y = c("c", "d"))
l2 <- list(x = "a", y = "z")
```

- `map(.x, .f, ...)`: Apply a function to each element of a list of vector, and return a list.

```
map(l1, sort, decreasing = TRUE)
```

- `map2(.x, .y, .f, ...)`: Apply a function pairs of elements from two lists or vectors, return a list.

```
map2(x, y, \(x, y) x*y)
```

`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.

- `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))
```

		One list	Two lists	Many lists
Logical	Returns a logical vector.	`map_lgl(x, is.integer)`	`map2_lgl(l2, l1, `%in%`)`	`pmap_lgl(list(l2, l1), `%in%`)`
Integer	Returns an integer vector.	`map_int(x, length)`	`map2_int(y, z, `+`)`	`pmap_int(list(y, z), `+`)`
Double	Returns a double vector.	`map_dbl(x, mean)`	`map2_dbl(y, z, ~ .x / .y)`	`pmap_dbl(list(y, z), ~ .x / .y)`
Character	Returns a character vector.	`map_chr(l1, paste, collapse = "")`	`map2_chr(l1, l2, paste, collapse = ",", sep = ":")`	`pmap_chr(list(l1, l2), paste, collapse = ",", sep = ":")`
Vector	Returns a vector that is of the simplest common type.	`map_vec(l1, paste, collapse = "")`	`map2_vec(l1, l2, paste, collapse = ",", sep = ":")`	`pmap_chr(list(l1, l2), paste, collapse = ",", sep = ":")`
No output	Calls `.f` for its side-effect.	`walk(x, print)`	`walk2(objs, paths, save)`	`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` with 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 `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 of vector. Also `reduce2()`.

```
a <- list(1, 2, 3, 4)
reduce(a, sum)
```

- `accumulate(.x, .f, ..., .init)`: Reduce a list, but also return intermediate results in a list. Also `accumulate2()`.

```
a <- list(1, 2, 3, 4)
accumulate(a, sum)
```

Vectors

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

```
compact(x)
```

- `keep_at()`: Keep/discard elements based by name or position.

```
keep_at(x, "a")
keep_at(x, 2)
```

- `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

A predicate function returns a single `TRUE` or `FALSE` and purrr provides

- `keep(.x, .p, ...)` retains elements where the predicate is `TRUE`; `discard(.x, .p, ...)` drops elements where the predicate is `TRUE`.

```
keep(x, is.numeric)
discard(x, is.numeric)
```

- `head_while(.x, .p, ...)` keeps the first elements until one fails the predicate. Also `tail_while()`.

```
head_while(x, is.character)
```

- `detect(.x, .f, ..., dir = c("forward", "backward"), .right = 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, ..., .deault = 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)
```

Reshape

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

```
list_flatten(x)
```

- `list_transpose(x)`: Transposes the index order in a multi-level list.

```
list_transpose(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()`: Combines elements into a vector by concatenating them together.

```
list_c(x1)
```

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

```
list_rbind(x2)
```

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

```
list_cbind(x2)
```

List-Columns

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()` and `transmute()`. Because each element is a list, use **map functions** within a column function to manipulate each element.

- `map()`, `map2()`, or `pmap()` return lists and will **create new list-columns**. In this example, `transmute()` is a column function, `map2()` is a list function which returns a list, and `vehicles` and `starships` are list-columns.

```
dplyr::starwars |>
  dplyr::mutate(ships = map2(vehicles, starships, append))
```

- Suffixed map functions like `map_int()` return an atomic data type and will **simplify list-columns into regular columns**. In this example, `mutate()` is a column function, `map_int()` is a list function which returns a column vector, and `films` is a list column.

```
dplyr::starwars |>
  dplyr::mutate(n_films = map_int(films, length))
```

Learn more at purrr.tidyverse.org.

Updated: 2024-05.

```
packageVersion("purrr")
```

```
[1] '1.0.2'
```
