INTRO TO FUNCTIONAL PROGRAMMING IN R

A workshop with

Christian Testa & Nicole A. Swartwood

Outline

What is functional programming?
- Why is it useful?

 apply family of functions
- description
- code demo

 {purrr} package
- description
- code demo

what functional programming?

Programming is an abstraction of reality.

Different programming paradigms:

Functional programming

- Focus on the evaluation of functions
- Variables and functions
- Declarative programming model

Object oriented programming

- Based on the concept of objects
- Objects and methods
- Imperative programming model

functional programming in R

- R incorporates aspects of both object oriented and functional programming.

- First class functions
  - You can do anything with functions that you can do with vectors: you can assign them to variables, store them in lists, pass them as arguments to other functions, create them inside functions, and even return them as the result of a function.

- Functional methods in lieu of loops
  - Apply functions
  - {purr} package map functions

why is functional programming useful in R?

Functional programming in R can be:

- more concise
- easier to read and debug
- faster (in certain situations)
- more elegant

apply functions

- A family of functions that allow the user to *apply* a function to elements of a vector, list, or dataframes.
- Avoids the explicit use of loops.
- Included in base R.
- Now considered **legacy** functionality and should **not** be used for **new** code – still useful for maintaining code bases.

apply functions

function	usage
`apply( )`	apply a given function across a dimension of an array; typically columns or rows of a matrix
`lapply( )`	***list apply -*** apply a given function to every element of a list and obtain a list as a result
`sapply( )`	***simplified lapply –*** apply a given function to every element of a list and returns the simplest possible form of the result

apply()

```
# general format of apply function
apply(X = dataStructure, MARGIN = margin, FUN = function)

# example usage of apply function
exampleMatrix <- matrix(1:50, nrow = 10, ncol =5)
# calculate across rows
apply(X = exampleMatrix, MARGIN = 1, FUN = mean)
# returns a vector of length 10 with means of each row

# calculate across columns
apply(X = exampleMatrix, MARGIN = 2, FUN = mean)
# returns a vector of length 5 with means of each column
```

lapply( )

```
# general format of lapply function
lapply(dataList, function)
dataList %>% lapply(function)

# example usage of apply function
exampleList <- list(height = sample(58:72, 10),
                    weight = sample(110:300, 10))
lapply(exampleList, median)
exampleList %>% lapply(median)

# returns a list of length 2 containing median of height in
first element and median of weight in the second element
```

sapply( )

```
# general format of lapply function
sapply(dataList, function)
dataList %>% sapply(function)

# example usage of apply function
exampleList <- list(height = sample(58:72, 10),
                    weight = sample(110:300, 10))
sapply(exampleList, median)
exampleList %>% sapply(median)

# returns a vector of length 2 containing median of height in
first element and median of weight in the second element
```

Additional apply( ) functions

function	usage
`tapply( )`	apply a given function to subsets of a data structure
`mapply( )`	***multi-variate sapply*** - passes multiple arguments to a given function
`rapply( )`	***recursive lapply*** - recursively apply a given function to a list
`vapply( )`	apply a given function and set the expected output type

These functions provide an answer to a more niche environments and attempt to solve some shortcomings of the original apply functions.

live demonstration – apply functions

{purrr} package

- purrr enhances R's functional programming (FP) toolkit by providing a complete and consistent set of tools for working with functions and vectors/lists.
- core of the functional programming in {purrr} package is the family of map() functions which replace many instances of for loops.

improvements in purr::map( ) vs. apply ( )

- Always returns a list or lists.
- Consistent input/output across different functions.
- Easier to write and read code.

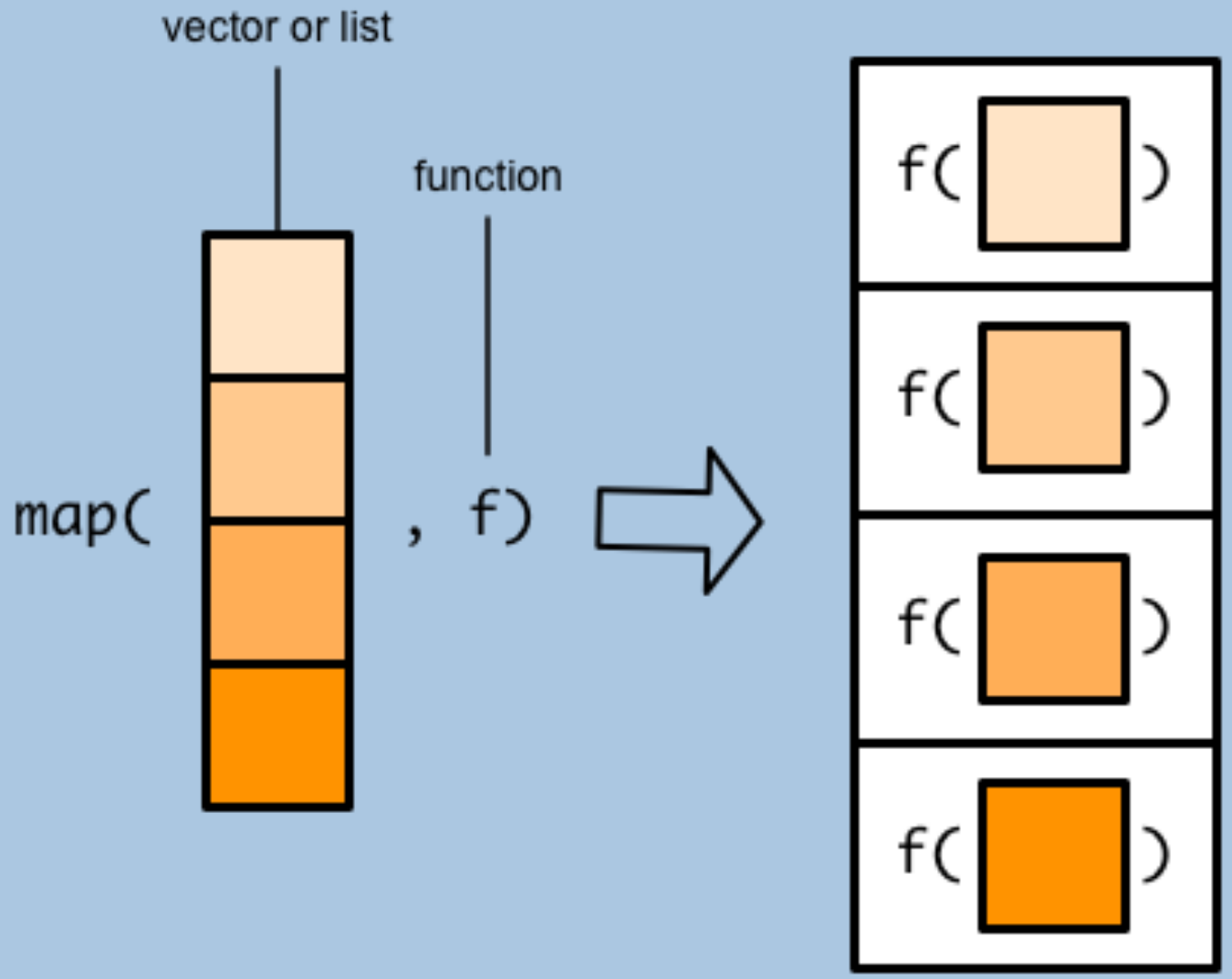
from R for Data Science:

- The goal of using purrr functions instead of for loops is to allow you to break common list manipulation challenges into independent pieces:
  1. How can you solve the problem for a single element of the list? Once you've solved that problem, purrr takes care of generalizing your solution to every element in the list.
  2. If you're solving a complex problem, how can you break it down into bite-sized pieces that allow you to advance one small step towards a solution? With purrr, you get lots of small pieces that you can compose together with the pipe.

{purr} map family of functions

function	usage
`map( )`	applies a function to **every** element of a list/vector
`map2( )`	applies a function to **every pair** of elements from two lists/vectors
`pmap( )`	***parallel map*** **-** applies a function to a group of elements in a list/vectors

- There are additional, more advanced map functions that allow you to apply a list of functions or apply functions to certain elements of the data structure.



map ( )

image from *Functional Programming,* by Sara Altman, Bill Behrman, Hadley Wickham

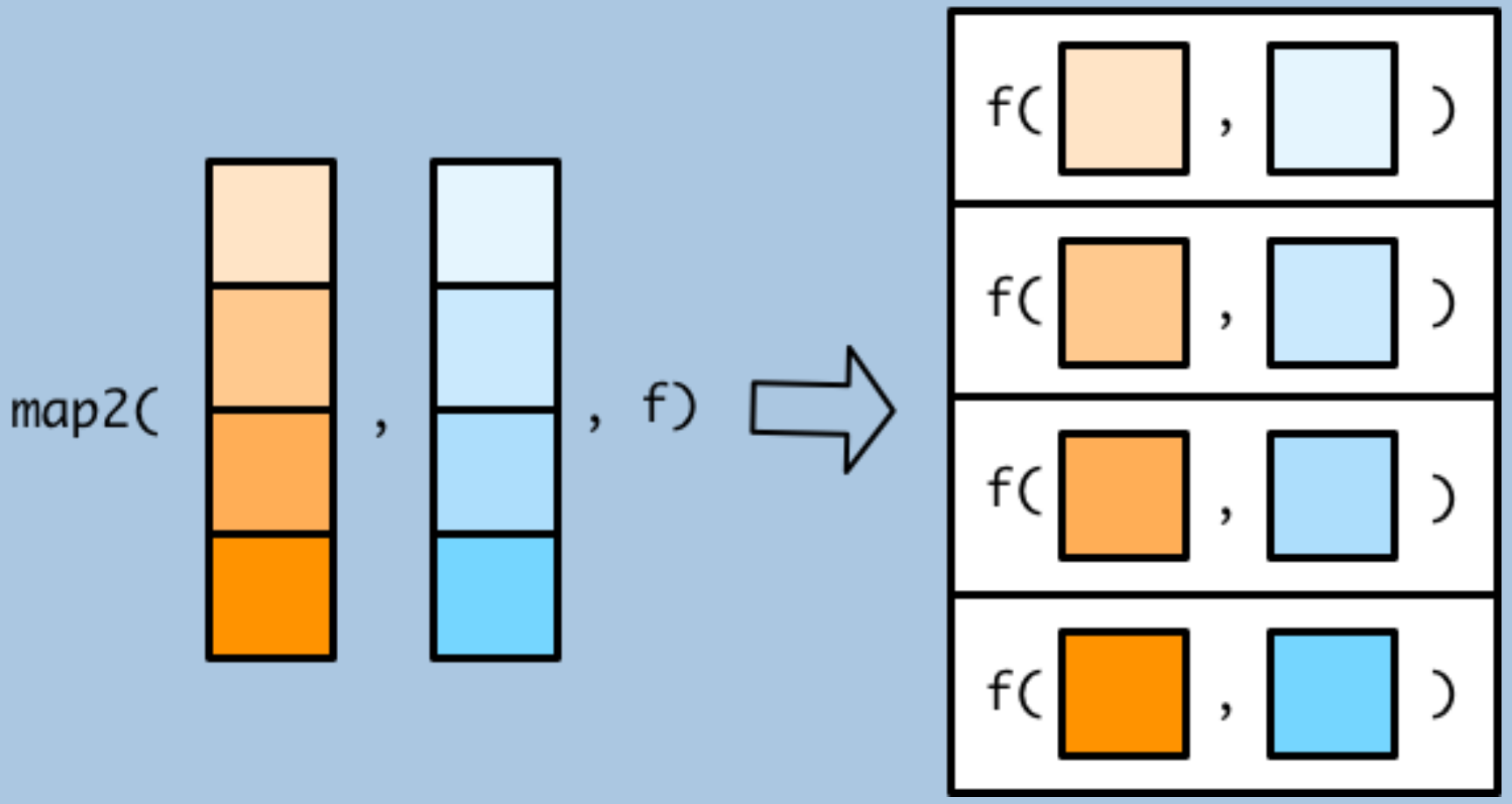
map()

```
# general format of map function
map(data_structure, function)
data_structure %>% map(function) # with pipes

# example usage of map function
exampleList <-list(sample1=sample(1:100, 10),
                   sample2=sample(1:100, 30))
map(exampleList, mean)

# example usage of map function with pipes
exampleList %>% map(mean)

# returns a list of length 2 mean of each element
  of the input list (each sample).
```

map()

	input	output
`map( )`	vector/list	vector/list
`map_lgl( )`	vector/list	logical vector/list
`map_int( )`	vector/list	integer vector/list
`map_dbl( )`	vector/list	double vector/list
`map_chr( )`	vector/list	character vector/list

These functions take a vector as input, applies a user-passed function to each element, then returns a new vector of the same length and with the same names as the input vector.

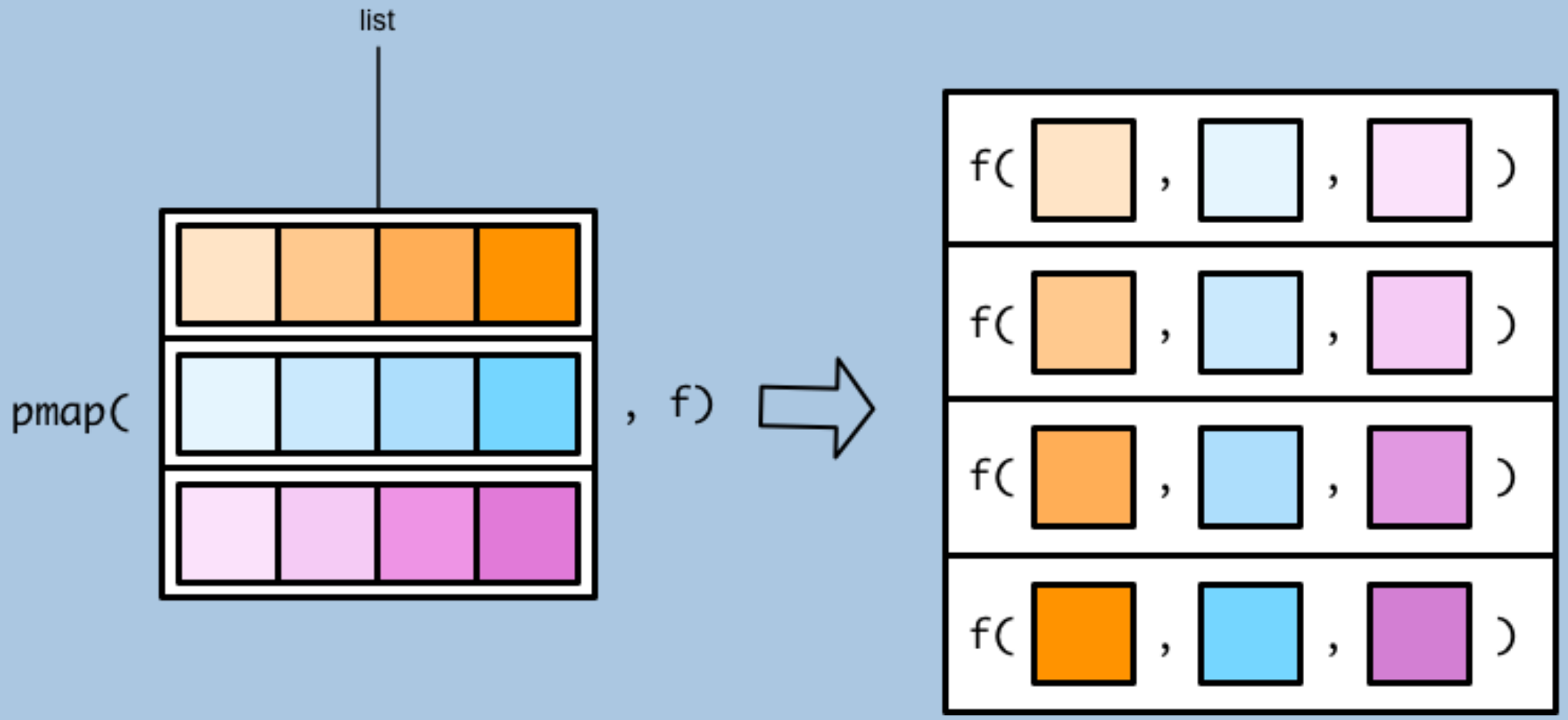


map2 ( )

image from *Functional Programming,* by Sara Altman, Bill Behrman, Hadley Wickham

map2()

```
# general format of map2 function
map2(data_structure1, data_structure2, function)

# example usage of map2 function
exampleVector1 <- sample(1:100, 12)
exampleVector2 <- sample(1:100, 12)
map2_dbl(exampleVector1, exampleVector2, min)

# returns a vector of length 12 with the min of each
element pair, i.e. the first element of the vector
is min(exampleVector1[1], exampleVector2[1])
```



pmap ( )

image from *Functional Programming,* by Sara Altman, Bill Behrman, Hadley Wickham

pmap ( )

```
# general format of pmap function
pmap(data_structure, function)

# example usage of pmap function
exampleDF <-data.frame( sample1 = sample(1:100, 10),
                        sample2 = sample(1:100, 10)),
                        sample3 = sample(1:100, 10))
pmap(exampleDF, sum)

# example usage of pmap function with pipes
exampleDF %>% pmap(sum)

# returns a list of length 10 with the sum of each first element,
  each second element, etc.
```

live demonstration – {purrr} package

additional resources

Apply functions with purrr : : **CHEAT SHEET**

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_dfc(.x, .f, ...**)**
Return a data frame created by column-binding.
map_dfc(l1, rep, 3)

map_dfr(.x, .f, ..., .id = NULL**)**
Return a data frame created by row-binding.
map_dfr(x, summary)

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)

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_dfc(.x, .y, .f, ...**)**
Return a data frame created by column-binding.
map2_dfc(l1, l2,
~ as.data.frame(c(.x, .y)))

map2_dfr(.x, .y, .f, ..., .id = NULL**)** Return a data frame created by row-binding.
map2_dfr(l1, l2,
~ as.data.frame(c(.x, .y)))

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

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), ~ ..1 * (..2 + ..3))

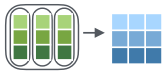
pmap_dbl(.l, .f, …**)**
Return a double vector.
pmap_dbl(list(y, z), ~ .x / .y)

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_dfc(.l, .f, ...**)** Return a data frame created by column-binding.
pmap_dfc(list(l1, l2),
~ as.data.frame(c(.x, .y)))



pmap_dfr(.l, .f, ..., .id = NULL**)** Return a data frame created by row-binding.
pmap_dfr(list(l1, l2),
~ as.data.frame(c(.x, .y)))

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

LISTS AND INDEXES

imap(.x, .f, ...**)** Apply .f to each element and its index, return a list.
imap(y, ~ paste0(.y, ": ", .x))

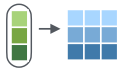
imap_dbl(.x, .f, …**)**
Return a double vector.
imap_dbl(y, ~ .y)

imap_int(.x, .f, ...**)**
Return an integer vector.
imap_int(y, ~ .y)

imap_chr(.x, .f, …**)**
Return a character vector.
imap_chr(y, ~ paste0(.y, ": ", .x))

imap_lgl(.x, .f, …**)**
Return a logical vector.
imap_lgl(l1, ~ is.character(.y))

imap_dfc(.x, .f, ...**)**
Return a data frame created by column-binding.
imap_dfc(l2,
~ as.data.frame(c(.x, .y)))



imap_dfr(.x, .f, ..., .id = NULL**)**
Return a data frame created by row-binding.
imap_dfr(l2,
~ as.data.frame(c(.x, .y)))

iwalk(.x, .f, ...**)** Trigger side effects, return invisibly.
iwalk(z, ~ print(paste0(.y, ": ", .x)))

Function Shortcuts

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

map(l, ~ . + 2)
becomes
map(l, function(x) x + 2))

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

map2(l, p, ~ .x +.y)
becomes
map2(l, p, function(l, p) l + p)

Use **~ ..1 ..2 ..3** etc with functions like **pmap()** that have many arguments.

pmap(list(a, b, c), ~ ..3 + ..1 - ..2)
becomes
pmap(list(a, b, c), function(a, b, c) c + a -b)

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), ~ 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"]])**

posit™

CC BY SA Posit Software, PBC • info@posit.co • posit.co • Learn more at **purrr.tidyverse.org** • purrr 1.0.1 • Updated: 2023-05

https://github.com/rstudio/cheatsheets/blob/main/purrr.pdf

Work with Lists

Filter

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

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

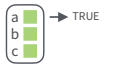
head_while(.x, .p, …**)**
Return head elements until one does not pass.
Also **tail_while()**.
head_while(x, is.character)

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)

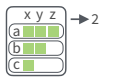
every(.x, .p, …**)**
Do all elements pass a test?
every(x, is.character)

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)

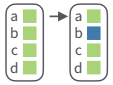
has_element(.x, .y**)**
Does a list contain an element?
has_element(x, "foo")



pluck_depth(x**)**
Return depth (number of levels of indexes).
pluck_depth(x)

Index

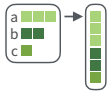
pluck(.x, ..., .default=NULL**)**
Select an element by name or index. Also **attr_getter()** and **chuck()**.
pluck(x, "b")
x |> pluck("b")



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

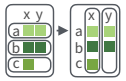
modify_in(.x, .where, .f**)**
Apply a function to a value at a selected location.
modify_in(x, "b", abs)
x |> modify_in("b", abs)

Reshape



flatten(.x**)** Remove a level of indexes from a list.
Also **flatten_chr()** etc.
flatten(x)

array_tree(array, margin = NULL**)** Turn array into list.
Also **array_branch()**.
z <- array(1:12, c(2,2,2))
array_tree(x, margin = 3)



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

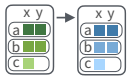
set_names(x, nm = x**)**
Set the names of a vector/list directly or with a function.
set_names(x, c("p", "q", "r"))
set_names(x, tolower)

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)

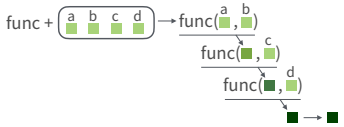
modify_if(.x, .p, .f, ...**)** Apply a function to elements that pass a test. Also **map_if()**.
modify_if(x, is.numeric,~.+2)



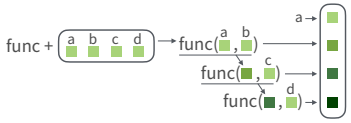
modify_depth(.x, .depth, .f, ...**)** Apply function to each element at a given level of a list. Also **map_depth()**.
modify_depth(x, 1, ~.+ 2)

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)



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()** 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**.

```
starwars |>
  transmute(ships = map2(vehicles, starships, append))
```

(list function, return list; list-columns; column function)

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

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

(list function, return int; column function; list-column)

CC BY SA Posit Software, PBC • info@posit.co • posit.co • Learn more at **purrr.tidyverse.org** • purrr 1.0.1 • Updated: 2023-05

Functional Programming

Sara Altman, Bill Behrman, Hadley Wickham

2021-09-09

Welcome

This book is a practical introduction to functional programming using the tidyverse.

An evolving book

This book is not intended to be static. Starting in April 2019, we use this book to teach functional programming in the Stanford Data Challenge Lab (DCL) course. The DCL functions as a testing ground for educational materials, as our students give us routine feedback on what they read and do in the course. We use this feedback to constantly improve our materials, including this book. The source for the book is also available on GitHub where we welcome suggestions for improvements.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

Functional programming text available at https://dcl-prog.stanford.edu

Stat 8054 Lecture Notes: R as a Functional Programming Language

Charles J. Geyer

January 24, 2023

1 License

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License (http://creativecommons.org/licenses/by-sa/4.0/).

2 R

- The version of R used to make this document is 4.2.1.
- The version of the `rmarkdown` package used to make this document is 2.19.
- The version of the `magrittr` package used to make this document is 2.0.3.
- The version of the `CatDataAnalysis` package used to make this document is 0.1.5.
- The version of the `glmbb` package used to make this document is 0.5.1.

3 Reading

- A blog post at R Bloggers: Functional programming in R.
- Three chapters in *Advanced R:*
  - Functional programming
  - Functionals
  - Function Operators
- A question in the R FAQ: What are the differences between R and S?.
- Some sections of my 3701 handout on the basics of R:
  - 4 Functions
  - 6 More on Functions
  - 7 Still More on Functions
  - 7.5 A Long Example (Maximum Likelihood Estimation), especially subsections 7.5.4 and 7.5.5 and 7.5.6
- Some sections of the book *The R Language Definition*, which is one of the R manuals that can be found at CRAN and also in your own R

https://www.stat.umn.edu/geyer/8054/notes/functional.html

slides created by N. Swartwood 12 July 2023