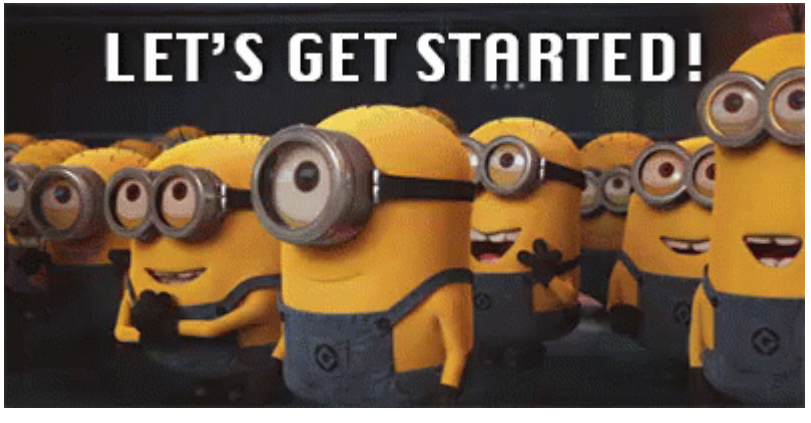
# Advanced R - Hadley Wickham Ch 9 Functionals

## R Ladies Netherlands Booookclub

Martine Jansen (@nnie_nl), July 21, 2020

# Welcome!

- This is joint effort between RLadies Nijmegen, Rotterdam, 's-Hertogenbosch (Den Bosch), Amsterdam and Utrecht
- We meet every 2 weeks to go through a chapter
- Use the HackMD to present yourself, ask questions and see your breakout room
- We split in breakout rooms after the presentation, and we return to the main jitsi link after xx min
- There are still possibilities to present a chapter :) Sign up at https://rladiesnl.github.io/book_club/
- https://advanced-r-solutions.rbind.io/ has some anwers and we could PR the ones missing
- The R4DS book club repo has a Q&A section.https://github.com/r4ds/bookclub-Advanced_R



# Thanks to:


- Hadley Wickham for writing AdvancedR
- Allison Horst for the palmerpenguins package
- R Core Team (2020). R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/.
- Lionel Henry and Hadley Wickham (2020). purrr: Functional Programming Tools. R package version 0.3.4. https://CRAN.R-project.org/package=purrr
- Gorman KB, Williams TD, Fraser WR (2014) Ecological Sexual Dimorphism and Environmental Variability within a Community of Antarctic Penguins (Genus Pygoscelis). PLoS ONE 9(3): e90081. https://doi.org/10.1371/journal.pone.0090081
- Yihui Xie (2020). xaringan: Presentation Ninja. R package version 0.16. https://CRAN.R-project.org/package=xaringan
- Garrick Aden-Buie (2020). xaringanExtra: Extras And Extensions for Xaringan Slides. R package version 0.0.17. https://github.com/gadenbuie/xaringanExtra

and all the authors of R packages used in this presentation

# Functional programming - Chp 9, 10 and 11

R has **first-class** functions. This means the language supports:

- assigning functions to variables
- store funtions in a list
- pass functions as arguments to other functions
- create functions inside functions
- return functions as the result of a function

A functions is **pure**, if:

- the output only depends on the input (same input, same output)
- the function has no side effects (so no actions beside the output value)

R is not strictly a functional programming language,
but it is very well possible to program in a **functional style**:

- decompose a problem in subproblems
- solve each subproblem with a set of functions.

# Functional programming - Chp 9, 10 and 11

| In | Out | |
|---|---|---|
| | Vector | Function |
| Vector | Regular functions<br>Chp 6 | Function factories<br>Chp 10 |
| Function | **Functionals**<br>**Chp 9** | Function operators<br>Chp 11 |

# Functionals, **function**(function) = vector

```
let_us <- function(f, inputs) f(inputs)
```

```
let_us(sum, 1:4)
```

```
[1] 10
```

```
let_us(median, 1:4)
```

```
[1] 2.5
```

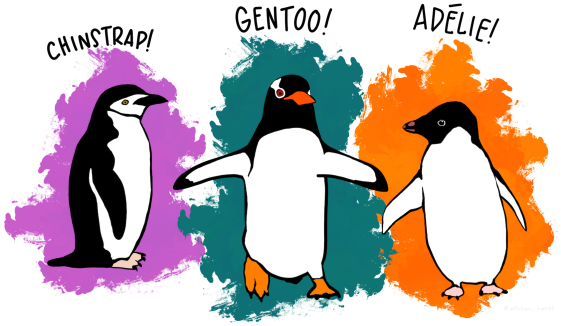
```
let_us(range, 1:4)
```

```
[1] 1 4
```

```
let_us(sqrt, 1:4)
```

```
[1] 1.000000 1.414214 1.732051 2.000000
```

# About the demo data: palmerpenguins::penguins

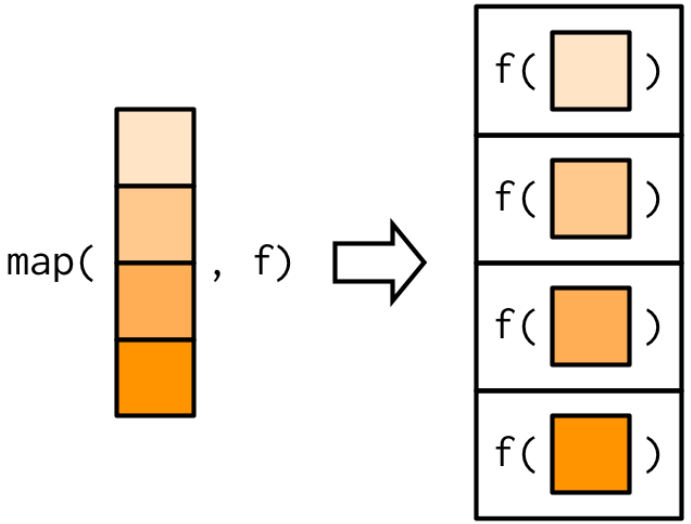



Artwork by @allison_horst

```
str(penguins)
```

```
tibble [344 x 7] (S3: tbl_df/tbl/data.frame)
 $ species          : Factor w/ 3 levels "Adelie","Chinstrap",..: 1 1 1 1 1 1 1 1 1 1 ...
 $ island           : Factor w/ 3 levels "Biscoe","Dream",..: 3 3 3 3 3 3 3 3 3 3 ...
 $ bill_length_mm   : num [1:344] 39.1 39.5 40.3 NA 36.7 39.3 38.9 39.2 34.1 42 ...
 $ bill_depth_mm    : num [1:344] 18.7 17.4 18 NA 19.3 20.6 17.8 19.6 18.1 20.2 ...
 $ flipper_length_mm: int [1:344] 181 186 195 NA 193 190 181 195 193 190 ...
 $ body_mass_g      : int [1:344] 3750 3800 3250 NA 3450 3650 3625 4675 3475 4250 ...
 $ sex              : Factor w/ 2 levels "female","male": 2 1 1 NA 1 2 1 2 NA NA ...
```

# purrr::map( )




input slot 1: vector
input slot 2: function
output: list

```
str(purrr::map)
```

```
function (.x, .f, ...)
```

```
map(penguins, typeof)
```

```
$species
[1] "integer"

$island
[1] "integer"

$bill_length_mm
[1] "double"

$bill_depth_mm
[1] "double"

$flipper_length_mm
[1] "integer"

$body_mass_g
[1] "integer"

$sex
[1] "integer"
```

# purrr::map( ) - atomic functions 1/3

list as output

```
map(penguins, typeof)
```

```
$species
[1] "integer"

$island
[1] "integer"

$bill_length_mm
[1] "double"

$bill_depth_mm
[1] "double"

$flipper_length_mm
[1] "integer"

$body_mass_g
[1] "integer"

$sex
[1] "integer"
```

chr vector as output

```
map_chr(penguins, typeof)
```

```
          species            island    bill_leng
        "integer"         "integer"          "do
flipper_length_mm       body_mass_g
        "integer"         "integer"         "int
```

# purrr::map( ) - atomic functions 2/3

logical vector as output

```
map_lgl(penguins, is.double)
```

```
          species            island    bill_length_mm     bill_depth_mm
            FALSE             FALSE              TRUE              TRUE
flipper_length_mm       body_mass_g               sex
            FALSE             FALSE             FALSE
```

integer vector as output

```
n_unique <- function(x) length(unique(x))
map_int(penguins, n_unique)
```

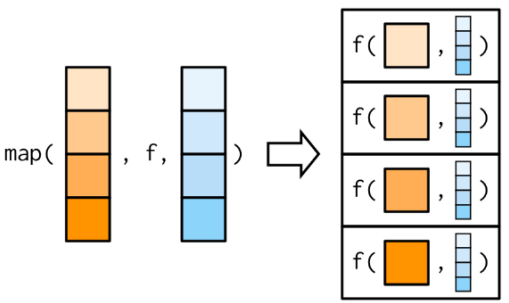
```
          species            island    bill_length_mm     bill_depth_mm
                3                 3               165                81
flipper_length_mm       body_mass_g               sex
               56                95                 3
```

# purrr::map( ) - atomic functions 3/3

```
map_dbl(penguins, mean, na.rm = TRUE)
```

```
          species            island    bill_length_mm     bill_depth_mm
               NA                NA          43.92193          17.15117
flipper_length_mm       body_mass_g               sex
        200.91520       4201.75439                NA
```

This is an example of passing arguments:




Extra arguments (after f) will be passed along as is (no decomposition).

# Anonymous functions and shortcuts 1/

```
# instead of:
n_unique <- function(x) length(unique(x))
map_int(penguins, n_unique)
```

```
          species            island    bill_length_mm     bill_depth_mm
                3                 3               165                81
flipper_length_mm       body_mass_g               sex
               56                95                 3
```

```
# you can create an inline anonymous function:
map_int(penguins, function(x) length(unique(x)))
```

```
          species            island    bill_length_mm     bill_depth_mm
                3                 3               165                81
flipper_length_mm       body_mass_g               sex
               56                95                 3
```

```
# or even with less characters, and with a tilde/twiddle:
map_int(penguins, ~length(unique(.x)))
```

```
          species            island    bill_length_mm     bill_depth_mm
                3                 3               165                81
flipper_length_mm       body_mass_g               sex
               56                95                 3
```

# Anonymous functions and shortcuts 2/

Very useful for making sets of random numbers

```
x <- map(1:3, ~ runif(.x))
str(x)
```

```
List of 3
 $ : num 0.416
 $ : num [1:2] 0.905 0.47
 $ : num [1:3] 0.3311 0.0935 0.4597
```

If at the second spot there is a vector/list instead of function,
result is extraction of elements from the input vector:

```
map_dbl(x, 1)
```

```
[1] 0.4164139 0.9046881 0.3310901
```

```
map_dbl(x, 2)
```

```
Error: Result 1 must be a single double, not NULL of length 0
```

# Anonymous functions and shortcuts 3/

```
map_dbl(x, 2)
```

```
Error: Result 1 must be a single double, not NULL of length 0
```

Prevent this error by supplying a default value:

```
map_dbl(x, 2, .default = NA)
```

```
[1]         NA 0.47013588 0.09346625
```

Extract elements by:

- location, with an integer vector
- by name, with a character vector
- combi name/location, with a list

# Intermezzo on *trim*

Plain mean:

```
mean(c(1,1,2,2,2,2,2,2,2,1000,1000))
```

```
[1] 168.1667
```

Trim 20% of the observations at both sides:

```
mean(c(1,1,2,2,2,2,2,2,2,1000,1000), trim = 0.2)
```

```
[1] 2
```

It cuts of from the ordered vector:

```
mean(c(1000,1,2,2,2,2,2,2,2,1000,1), trim = 0.2)
```

```
[1] 2
```

So it trims the smallest and the largest values.

# Use argument names

Calculate the means for the numeric columns in `penguins`:

```
map_dbl(penguins[,3:6], mean, TRUE)
```

```
Error in mean.default(.x[[i]], ...): 'trim' must be numeric of length one
```

What we wanted was to have the `na.rm = TRUE`;

```
map_dbl(penguins[,3:6], mean, na.rm = TRUE)
```

```
   bill_length_mm     bill_depth_mm flipper_length_mm       body_mass_g
         43.92193          17.15117         200.91520        4201.75439
```

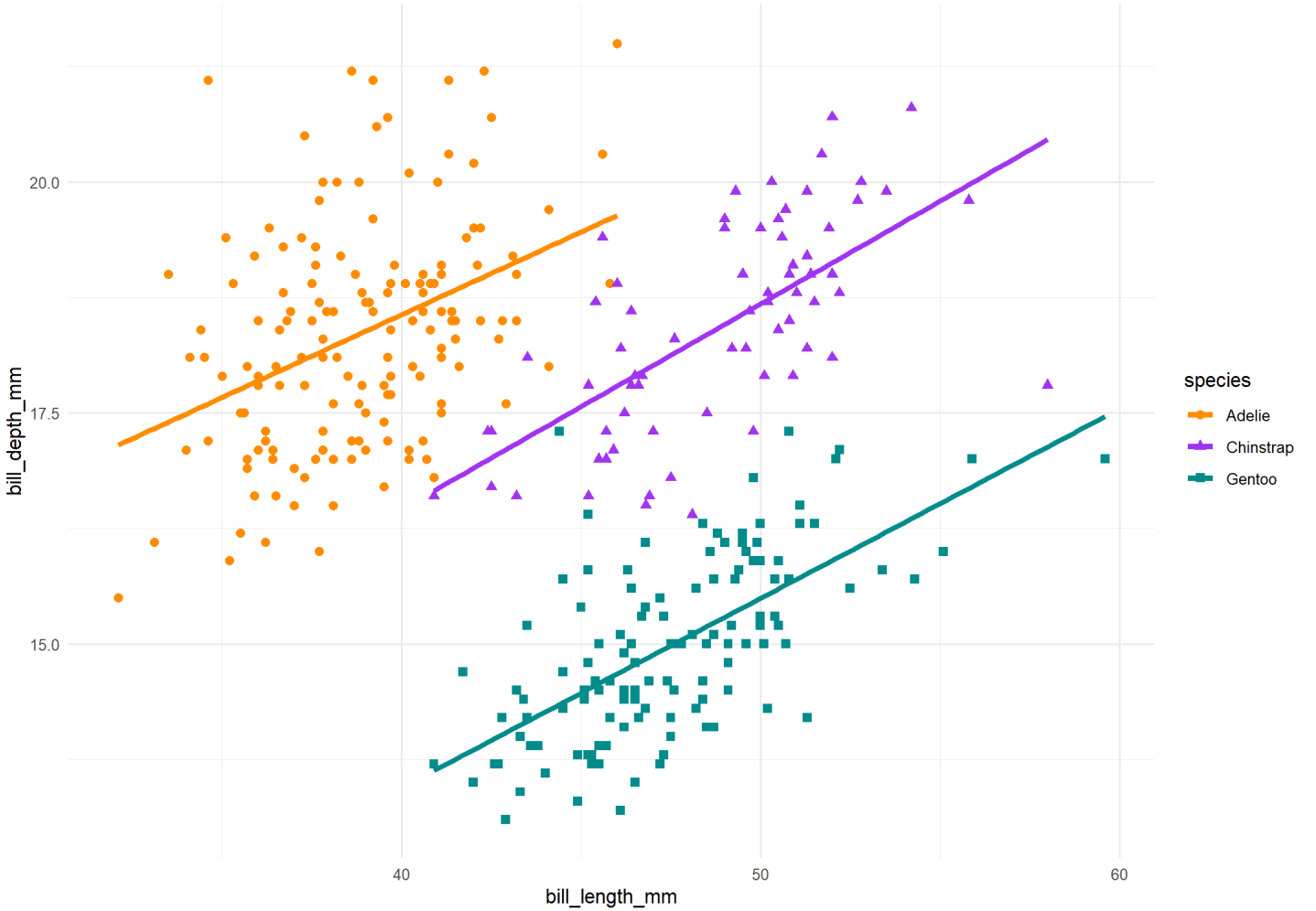
Another example:

```
map_dbl(penguins[,3:6], mean, na.rm = TRUE, 0.1)
```

```
   bill_length_mm     bill_depth_mm flipper_length_mm       body_mass_g
         43.90693          17.17263         200.33577        4154.01460
```

The 0.1 appears to be the *trim*. But would you know that by looking at the script?

# Regression example



# Regression example

```
penguins_species <- split(penguins, penguins$species)
```

```
penguins_species %>%
  # inputs are the 3 splits
  # of penguins
  # function is the lm:
  map(~ lm(bill_depth_mm ~
             bill_length_mm,
           data = .x))
```

```
$Adelie

Call:
lm(formula = bill_depth_mm ~ bill_length_mm, dat

Coefficients:
   (Intercept)  bill_length_mm
       11.4091          0.1788


$Chinstrap

Call:
lm(formula = bill_depth_mm ~ bill_length_mm, dat

Coefficients:
   (Intercept)  bill_length_mm
        7.5691          0.2222


$Gentoo

Call:
lm(formula = bill_depth_mm ~ bill_length_mm, dat
```

# Regression example

```
penguins_species <- split(penguins, penguins$species)
```

```
penguins_species %>%
  # inputs are the 3 splits
  # of penguins
  # function is the lm:
  map(~ lm(bill_depth_mm ~
             bill_length_mm,
           data = .x)) %>%
  # inputs are the models
  # function is coef()
  map(coef)
```

```
$Adelie
   (Intercept) bill_length_mm
    11.4091245      0.1788343

$Chinstrap
   (Intercept) bill_length_mm
     7.5691401      0.2222117

$Gentoo
   (Intercept) bill_length_mm
     5.2510084      0.2048443
```

# Regression example

```
penguins_species <- split(penguins, penguins$species)
```

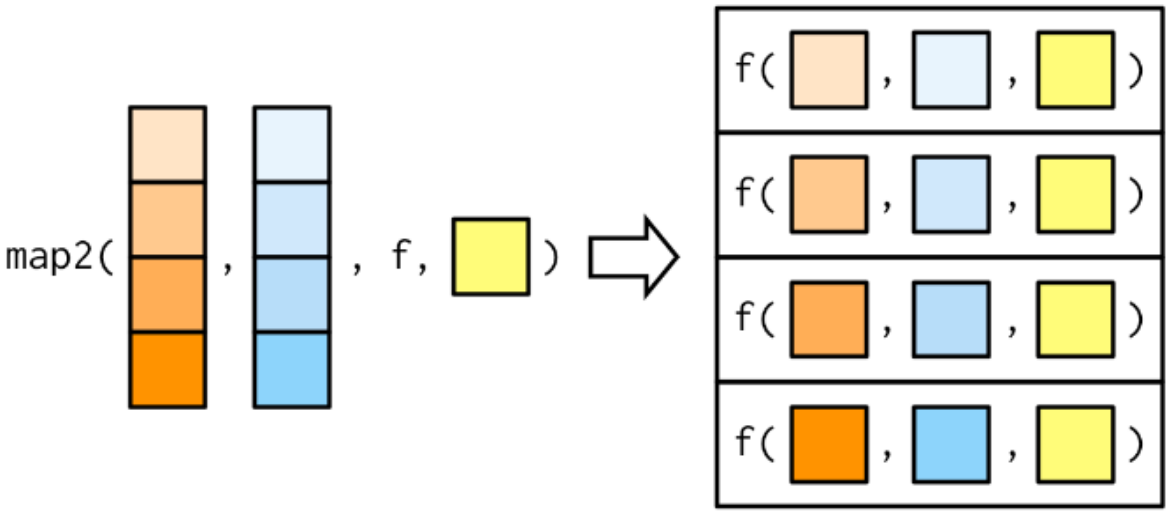
```
penguins_species %>%
  # inputs are the 3 splits
  # of penguins
  # function is the lm:
  map(~ lm(bill_depth_mm ~
             bill_length_mm,
           data = .x)) %>%
  # inputs are the models
  # function is coef()
  map(coef) %>%
  # inputs are sets of coefficients
  # subset to get the slopes
  map(2)
```

```
$Adelie
[1] 0.1788343

$Chinstrap
[1] 0.2222117

$Gentoo
[1] 0.2048443
```

# purrr::map2()




```
str(purrr::map2)
```

```
function (.x, .y, .f, ...)
```

input slot 1: vector
input slot 2: another vector
input slot 3: function
input slot 4: additional arguments to function
output: list

# purrr::map2( ) - examples

```
map2(1:3, 4:6, ~.x ^ .y)
```

```
[[1]]
[1] 1

[[2]]
[1] 32

[[3]]
[1] 729
```

```
map2_chr(penguins$species, penguins$island, str_c, sep = " - ") %>% head(30)
```

```
 [1] "Adelie - Torgersen" "Adelie - Torgersen" "Adelie - Torgersen"
 [4] "Adelie - Torgersen" "Adelie - Torgersen" "Adelie - Torgersen"
 [7] "Adelie - Torgersen" "Adelie - Torgersen" "Adelie - Torgersen"
[10] "Adelie - Torgersen" "Adelie - Torgersen" "Adelie - Torgersen"
[13] "Adelie - Torgersen" "Adelie - Torgersen" "Adelie - Torgersen"
[16] "Adelie - Torgersen" "Adelie - Torgersen" "Adelie - Torgersen"
[19] "Adelie - Torgersen" "Adelie - Torgersen" "Adelie - Biscoe"
[22] "Adelie - Biscoe"    "Adelie - Biscoe"    "Adelie - Biscoe"
[25] "Adelie - Biscoe"    "Adelie - Biscoe"    "Adelie - Biscoe"
[28] "Adelie - Biscoe"    "Adelie - Biscoe"    "Adelie - Biscoe"
```

# purrr::map2( ) recycles

```
map2(1:3, 4,  ~.x ^ .y)
```

```
[[1]]
[1] 1

[[2]]
[1] 16

[[3]]
[1] 81
```

This is equivalent with
treating the second argument as an additional argument to the function in a map():

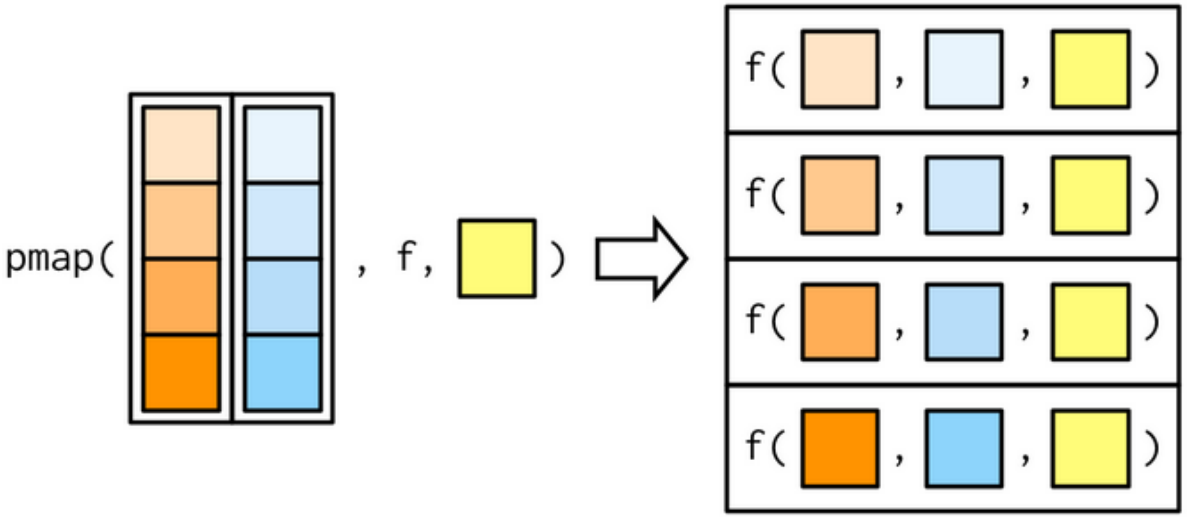
```
map(1:3, ~.x ^ .y, 4)
```

```
[[1]]
[1] 1

[[2]]
[1] 16

[[3]]
[1] 81
```

# purrr::pmap( )




```
str(purrr::pmap)
```

```
function (.l, .f, ...)
```

input slot 1: list
input slot 2: function
input slot 3: additional arguments to function
output: list

# purrr::pmap( ) - example

Use pmap to draw sets of random samples from a number of normal distributions:

```
str(rnorm)
```

```
function (n, mean = 0, sd = 1)
```

So we need a tibble with the desired inputs:

```
params <- tibble(n = 3:5, mean = c(30, 20, 10), sd = 1:3)
```

Because the variable names in `params` match the argument names in `rnorm`, this works:

```
pmap(params, rnorm)
```

```
[[1]]
[1] 29.13720 29.99630 29.05474

[[2]]
[1] 21.70650 19.42706 19.65767 19.48892

[[3]]
[1]  5.703459 14.434996  8.865970  8.252433  7.747931
```

# purrr::imap( ) - example names

```
str(imap)
```

```
function (.x, .f, ...)
```

`imap(x,f)`:

- x has names → `map2(x, names(x), f)`
- x has NO names → `map2(x, seq_along(x), f)`

**Example**

```
imap_chr(penguins, ~ str_c("First value of ", .y, " is ", .x[[1]]))
```

```
                                    species
      "First value of species is Adelie"
                                     island
    "First value of island is Torgersen"
                                bill_length_mm
  "First value of bill_length_mm is 39.1"
                                 bill_depth_mm
   "First value of bill_depth_mm is 18.7"
                               flipper_length_mm
"First value of flipper_length_mm is 181"
                                  body_mass_g
    "First value of body_mass_g is 3750"
```

# purrr::imap( ) - example no names

```
x <- map(1:3, ~ sample(1000, 10))
```

```
x
```

```
[[1]]
 [1] 391 564 895 218 150 244 372 190 227 640

[[2]]
 [1] 956 363 183 676 424 785 859 490 602 400

[[3]]
 [1] 520 197 360 957 817  83 162 760 753 351
```

```
imap_chr(x, ~ paste0("The highest value of ", .y, " is ", max(.x)))
```

```
[1] "The highest value of 1 is 895" "The highest value of 2 is 956"
[3] "The highest value of 3 is 957"
```

# modify( )

Returns a modified copy, same type of output as input.

```
str(modify)
```

```
function (.x, .f, ...)
```

```
map(penguins[3:6], ~.x*2) %>% str()
```

```
List of 4
 $ bill_length_mm   : num [1:344] 78.2 79 80.6 NA 73.4 78.6 77.8 78.4 68.2 84 ...
 $ bill_depth_mm    : num [1:344] 37.4 34.8 36 NA 38.6 41.2 35.6 39.2 36.2 40.4 ...
 $ flipper_length_mm: num [1:344] 362 372 390 NA 386 380 362 390 386 380 ...
 $ body_mass_g      : num [1:344] 7500 7600 6500 NA 6900 7300 7250 9350 6950 8500 ...
```

```
modify(penguins[3:6], ~.x*2) %>% str()
```

```
tibble [344 x 4] (S3: tbl_df/tbl/data.frame)
 $ bill_length_mm   : num [1:344] 78.2 79 80.6 NA 73.4 78.6 77.8 78.4 68.2 84 ...
 $ bill_depth_mm    : num [1:344] 37.4 34.8 36 NA 38.6 41.2 35.6 39.2 36.2 40.4 ...
 $ flipper_length_mm: num [1:344] 362 372 390 NA 386 380 362 390 386 380 ...
 $ body_mass_g      : num [1:344] 7500 7600 6500 NA 6900 7300 7250 9350 6950 8500 ...
```

# walk( )

Most functions are used for their return value.
Some mainly for their side-effects, like plot(), write.csv(), ...
These latter functions do have an invisible return value (see 6.7.2),
that become visible with map( ).

```
print(3:4)
```

```
[1] 3 4
```

```
map_chr(3:4, print)
```

```
[1] 3
[1] 4

[1] "3" "4"
```

Use `walk` when you only want the side-effect:

```
walk(3:4, print)
```

```
[1] 3
[1] 4
```

# walk( ) - example saving multiple files

```
temp <- tempfile()
# returns a vector of character strings
# which can be used as names for temporary files
temp
```

```
[1] "C:\\Users\\871112\\AppData\\Local\\Temp\\RtmpoXF7rC\\file4aa064ba477f"
```

```
# make the temporary directory
dir.create(temp)
# and see that it is empty
dir(temp)
```

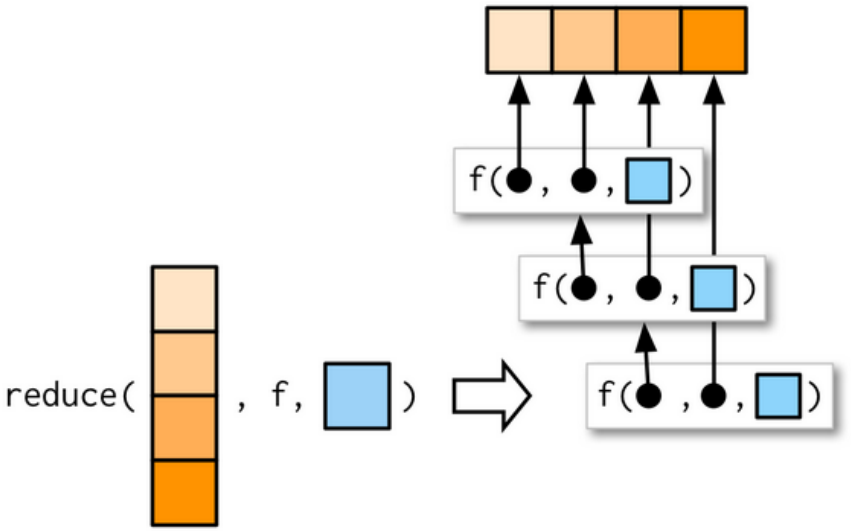
```
character(0)
```

```
peng <- split(penguins, penguins$species)
paths <- file.path(temp, paste0("species-", names(peng), ".csv"))
walk2(peng, paths, write.csv)
dir(temp)
```

```
[1] "species-Adelie.csv"    "species-Chinstrap.csv" "species-Gentoo.csv"
```

```
# to clean up all the temp directories and content made
unlink(temp, recursive = TRUE)
```

# reduce( )




```
str(reduce)
```

```
function (.x, .f, ..., .init, .dir = c("forward", "backward"))
```

# reduce( ) - example intersect

```
mylist <- map(1:3, ~ sample(1:10, 15, replace = T))
str(mylist)
```

```
List of 3
 $ : int [1:15] 9 7 8 10 2 6 4 3 8 4 ...
 $ : int [1:15] 2 10 2 7 1 9 6 4 1 8 ...
 $ : int [1:15] 3 3 1 6 8 10 4 8 5 4 ...
```

How to find the common numbers?

```
out <- mylist[[1]]
(out <- intersect(out, mylist[[2]]))
```

```
[1]  9  7  8 10  2  6  4  3
```

```
(out <- intersect(out, mylist[[3]]))
```

```
[1]  8 10  2  6  4  3
```

```
reduce(mylist, intersect)
```

```
[1]  8 10  2  6  4  3
```

```
accumulate(mylist, intersect)
```

```
[[1]]
 [1]  9  7  8 10  2  6  4  3  8  4  7  7 10  7

[[2]]
[1]  9  7  8 10  2  6  4  3

[[3]]
[1]  8 10  2  6  4  3
```

# reduce( ) - use of .init 1/2

```
reduce(1:4, `*`)
```

```
[1] 24
```

```
reduce(3, `*`)
```

```
[1] 3
```

```
reduce("purrrrrr", `*`)
```

```
[1] "purrrrrr"
```

So, obviously no check on input type ...

# reduce( ) - use of .init 2/2

```
reduce(integer(), `*`)
```

```
Error: `.x` is empty, and no `.init` supplied
```

From the help on `init`:
If supplied, will be used as the first value to start the accumulation, rather than using [1]. This is useful if you want to ensure that reduce returns a correct value when .x is empty. If missing, and .x is empty, will throw an error.

```
# prod returns the product of all the values present in its arguments.
prod(integer())
```

```
[1] 1
```

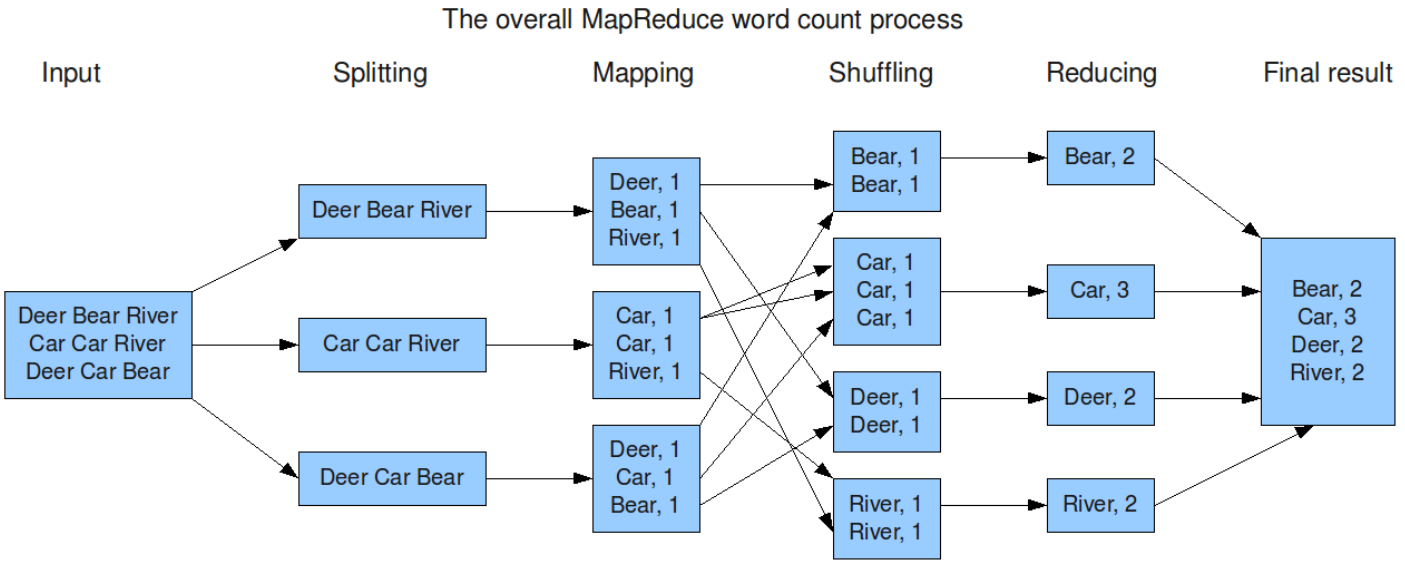
```
reduce(integer(), `*`, .init=1)
```

```
[1] 1
```

```
reduce("purrrrrr", `*`, .init = 1)   # and now this DOES throw an error
```

```
Error in .x * .y: non-numeric argument to binary operator
```

# Map-reduce




Source: https://whatsbigdata.be/wp-content/uploads/2014/06/MapReduce_Work_Structure.png

# Predicate functionals 1/4

A **predicate function** is a function that returns TRUE or FALSE.

```
is.numeric(penguins$species)
```

```
[1] FALSE
```

A **predicate functional** applies a predicate function to each element of a vector:

```
# some returns TRUE when first TRUE element is seen
some(penguins, is.numeric)
```

```
[1] TRUE
```

```
# every returns FALSE when first FALSE element is seen
every(penguins, is.numeric)
```

```
[1] FALSE
```

```
# none has to check entire vector
none(mylist, is.numeric)
```

```
[1] FALSE
```

# Predicate functionals 2/4

```
detect(penguins %>% head(), is.factor)        # value of the first match
```

```
[1] Adelie Adelie Adelie Adelie Adelie Adelie
Levels: Adelie Chinstrap Gentoo
```

```
detect_index(penguins %>% head(), is.factor)  # location of the first match
```

```
[1] 1
```

```
keep(penguins %>% head(n = 1), is.numeric)
```

```
# A tibble: 1 x 4
  bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
           <dbl>         <dbl>             <int>       <int>
1           39.1          18.7               181        3750
```

```
discard(penguins %>% head(n = 1), is.numeric)
```

```
# A tibble: 1 x 3
  species island    sex
  <fct>   <fct>     <fct>
1 Adelie  Torgersen male
```

# Predicate functionals 3/4

```
map_dbl(penguins, mean, na.rm = TRUE)
```

```
          species            island    bill_length_mm     bill_depth_mm
               NA                NA          43.92193          17.15117
flipper_length_mm       body_mass_g               sex
        200.91520        4201.75439                NA
```

```
map_dbl(keep(penguins, is.numeric), mean, na.rm = TRUE)
```

```
   bill_length_mm     bill_depth_mm flipper_length_mm       body_mass_g
         43.92193          17.15117         200.91520        4201.75439
```

# Predicate functionals 4/4

```
map_if(penguins,is.numeric, mean, na.rm = TRUE) %>% str()
```

```
List of 7
 $ species          : Factor w/ 3 levels "Adelie","Chinstrap",..: 1 1 1 1 1 1 1 1 1 1 ...
 $ island           : Factor w/ 3 levels "Biscoe","Dream",..: 3 3 3 3 3 3 3 3 3 3 ...
 $ bill_length_mm   : num 43.9
 $ bill_depth_mm    : num 17.2
 $ flipper_length_mm: num 201
 $ body_mass_g      : num 4202
 $ sex              : Factor w/ 2 levels "female","male": 2 1 1 NA 1 2 1 2 NA NA ...
```

```
modify_if(penguins,is.numeric, mean, na.rm = TRUE) %>% str()
```

```
tibble [344 x 7] (S3: tbl_df/tbl/data.frame)
 $ species          : Factor w/ 3 levels "Adelie","Chinstrap",..: 1 1 1 1 1 1 1 1 1 1 ...
 $ island           : Factor w/ 3 levels "Biscoe","Dream",..: 3 3 3 3 3 3 3 3 3 3 ...
 $ bill_length_mm   : num [1:344] 43.9 43.9 43.9 43.9 43.9 ...
 $ bill_depth_mm    : num [1:344] 17.2 17.2 17.2 17.2 17.2 ...
 $ flipper_length_mm: num [1:344] 201 201 201 201 201 ...
 $ body_mass_g      : num [1:344] 4202 4202 4202 4202 4202 ...
 $ sex              : Factor w/ 2 levels "female","male": 2 1 1 NA 1 2 1 2 NA NA ...
```

# bye


Let us take a break!
I feel amazing! It's all done.

# Breakout questions

- Did you use these functions before? Please share some details/tips/stories.
- What are the 23 primary variants of `map( )` Hadley Wickham mentiones in 9.4?
- In case you want to try some exercises, try these:
    - 9.2.6.3
    - 9.2.6.6
    - 9.6.3.5
    - or any other exercise :)