

Functional programming

with **purrr**

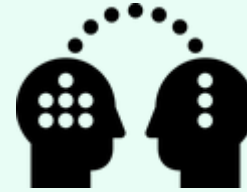


A. Ginolhac | rworkshop | 2021-09-10

Learning objectives

You will learn

- Functional programming approach to focus on **actions**
- Iteration machinery done by someone else
- Pass **functions** as **arguments** to higher order functions
- Use **map()** to replace **for** loops



Reminders

Vectors

Atomic means only one type of data

- The type of each atom is the **same**
- The size of each atom is **1** (single element)
- is the **conversion** between types
- can be
 - **explicit** (using **as.*()** functions)
 - **implicit**

```
# Logical  
c(TRUE, FALSE, TRUE)
```

```
[1] TRUE FALSE TRUE
```

```
# double  
c(1, 5, 7)
```

```
[1] 1 5 7
```

```
# automatic coercion  
str(c(1, 5, 7, "seven"))
```

```
chr [1:4] "1" "5" "7" "seven"
```

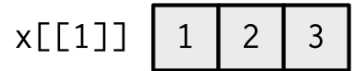
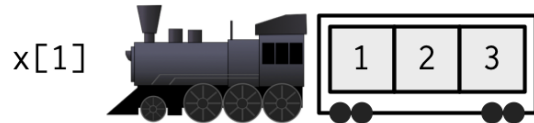
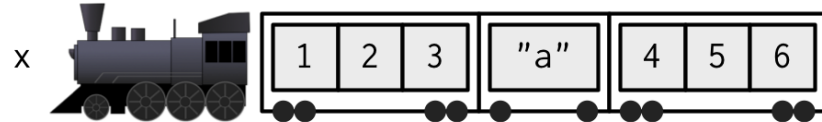
```
# voluntary coercion  
as.character(c(1, 5, 7, "seven"))
```

```
[1] "1"      "5"      "7"      "seven"
```

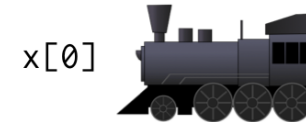
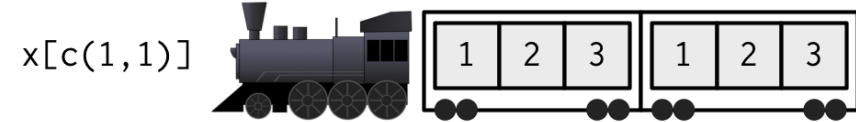
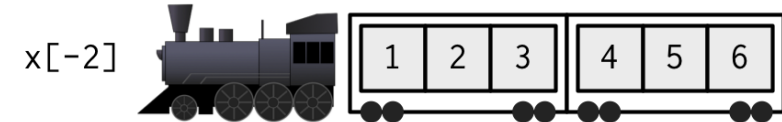
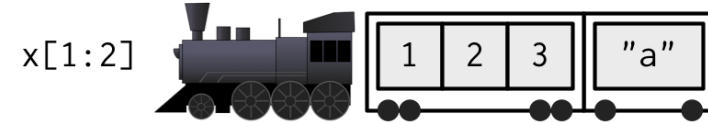
Adapted from the [tutorial](#) of **Jennifer Bryan**

Lists as trains

The steam machine is the `list()` structure



Sub-selection



from [Advanced R, second Ed.](#) by **Hadley Wickham**

Actions: functions

Declared function

```
my_function <- function(my_argument) {  
  my_argument + 1  
}
```

- Is defined in the global environment

```
ls()
```

```
[1] "my_function"
```

```
ls.str()
```

```
my_function : function (my_argument)
```

- Is reusable

```
my_function(2)
```

```
[1] 3
```

Anonymous functions

- Are not stored in an object and are used "on the fly"

```
(function(x) { x + 2 })(2) # (\(x) x + 2)(2)
```

```
[1] 4
```

- Does not alter the global environment

```
ls()
```

```
[1] "my_function"
```

```
# remove the previous my_function to convince you  
rm(my_function)  
(function(x) { x + 1 })(2)
```

```
[1] 3
```

```
ls()
```

```
character(0)
```



purrr enhances R with consistent tools for working with functions and vectors

Functional programming [...] treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data

– *Wikipedia*

Function, the LEGO figures example

Consider a hypothetical `put_on` function



+



=



`put_on(figures, antenna)` returns a LEGO figure with antenna

Figures LEGO pictures courtesy by **Jennifer Bryan**

Iteration, the LEGO figures example

Illustration for several legos, how to apply `put_on()` to more than 1 input?



+



`put_on(figures, antenna)`

=



[LEGO figure pictures from Jennifer Bryan](#)

Back to R programming

For loop approach

```
out <- vector("list", length(legos))
for (i in seq_along(legos)) {
  out[[i]] <- put_on(legos[[i]], antenna)
}
out
```

Functional programming approach

- named function

```
antennate <- function(x) put_on(x, antenna)
map(figures, antennate)
```

- anonymous function

```
map(figures, function(x) put_on(x, antenna))
```

Of course, someone has to write loops. It doesn't have to be you.

— *Jenny Bryan*

Your turn

Questions ?

Calculate the **mean** of each column of the **swiss** dataset, which is packaged with base **R**.

Tips

- `purrr::map()` expects 2 arguments:
 1. a **list**
 2. a **function**
- a data frame is a list
- Each column represents an element of the list *i.e.* a **data frame is a list of columns**

04:00

Answer

Functional programming focuses on actions

```
map(swiss, mean) %>%  
  str()
```

```
List of 6  
$ Fertility      : num 70.1  
$ Agriculture    : num 50.7  
$ Examination    : num 16.5  
$ Education      : num 11  
$ Catholic       : num 41.1  
$ Infant.Mortality: num 19.9
```

The for loop machinery

```
means <- vector("list", ncol(swiss))  
for (i in seq_along(swiss)) {  
  means[i] <- mean(swiss[[i]])  
}  
# need to manually add names  
names(means) <- names(swiss)  
means %>%  
  str()
```

```
List of 6  
$ Fertility      : num 70.1  
$ Agriculture    : num 50.7  
$ Examination    : num 16.5  
$ Education      : num 11  
$ Catholic       : num 41.1  
$ Infant.Mortality: num 19.9
```

For loops are fine

Growing vector

```
for_loop <- function(x) {  
  res <- c() # initialize an empty vector  
  for (i in seq_len(x)) {  
    res[i] <- i  
  }  
}
```

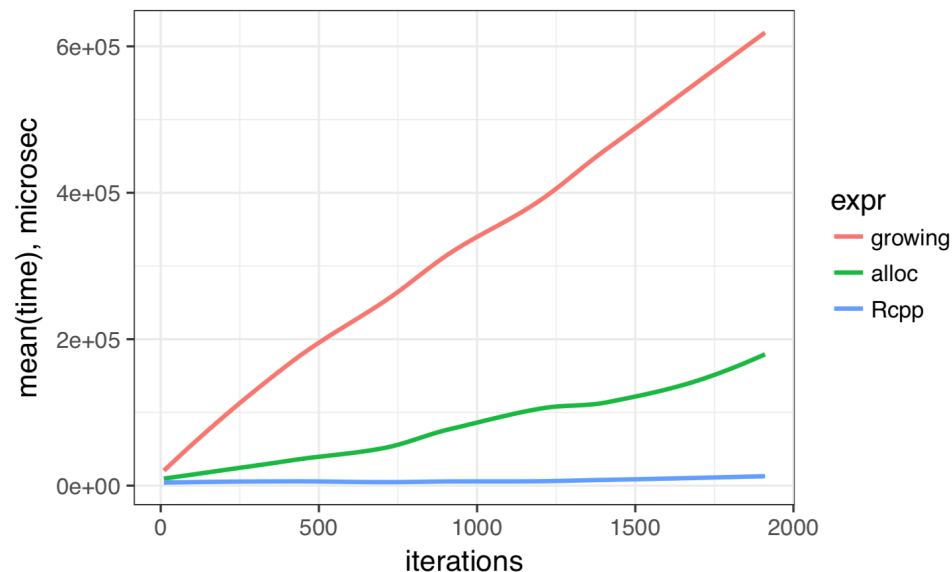
Correct memory allocation

```
for_loop <- function(x) {  
  res <- vector(mode = "integer", length = x)  
  for (i in seq_len(x)) {  
    res[i] <- i  
  }  
}
```

Using Rcpp

```
library(Rcpp) # binds cpp compilation to R  
cppFunction("NumericVector rcpp(int x) {  
  NumericVector res(x);  
  for (int i=0; i < x; i++) {  
    res[i] = i;  
  }  
}")
```

for loops in R
should avoid growing a vector



The `purrr::map()` family of functions

- Are designed to be consistent
- `map()` is the general function and close to `base::lapply()`
- `map()` introduces shortcuts (absent in `lapply()`)
- Variants to specify the type of vectorized output:
 - `map_lgl()`
 - `map_int()`
 - `map_dbl()`
 - `map_chr()`
 - `map_dfr()` data.frame rows
 - `map_df()` data.frame cols
- **Fail** if coercion is impossible

```
map_dbl(swiss, mean)
```

Fertility	Agriculture	Examination	Educ
70.14255	50.65957	16.48936	10.9
Catholic	Infant.Mortality		
41.14383	19.94255		

```
map_chr(swiss, mean)
```

Fertility	Agriculture	Examination	Educa
"70.142553"	"50.659574"	"16.489362"	"10.978"
Catholic	Infant.Mortality		
"41.143830"	"19.942553"		

```
map_int(swiss, mean)
```

```
Error: Can't coerce element 1 from a double to a integer
```

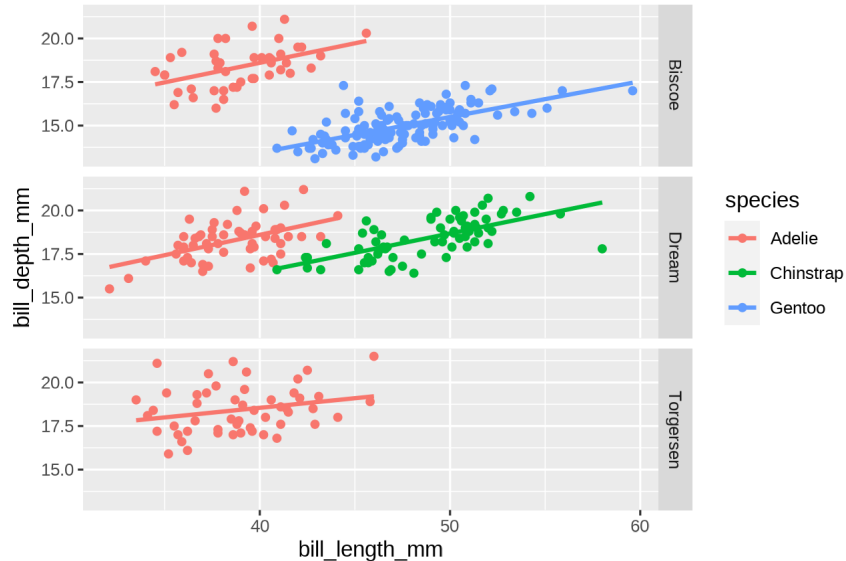
Linear modelling example

Palmer penguins from previous lecture

```
library(palmerpenguins)
ggplot(penguins,
      aes(x = bill_length_mm,
          y = bill_depth_mm,
          colour = species)) +
  geom_point() +
  geom_smooth(method = "lm", formula = 'y ~ x',
             # no standard error ribbon
             se = FALSE) +
  facet_grid(island ~ .)
```

How to perform those 5 linear model?

From [R for Data Science](#)



`palmerpenguins` from Horst AM, Hill AP, Gorman KB (2020)

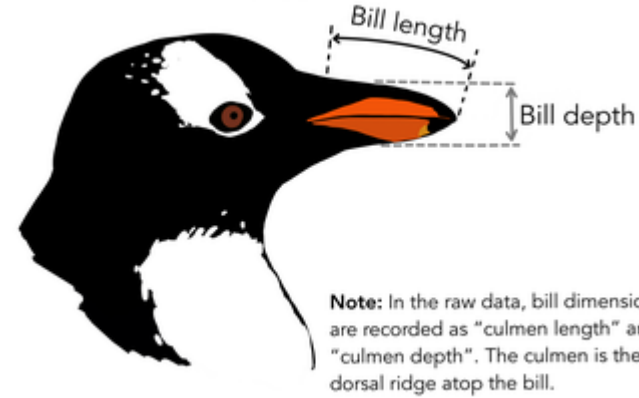
Reminder: fit a linear model

- Using the `penguins` dataset we can fit a linear model to explain the `bill depth` by the `bill length` using:

```
lm(bill_depth_mm ~ bill_length_mm, data = penguins)
```

```
Call:
lm(formula = bill_depth_mm ~ bill_length_mm, data = penguins)

Coefficients:
(Intercept)  bill_length_mm
  20.88547      -0.08502
```



Reminder: `lm` outputs complex objects

Summarise a linear model
with `base::summary()`

- r^2 is low (`0.05525`)
because we mix **all** individuals.
- We need **one** tibble **per** group

```
summary(lm(bill_depth_mm ~ bill_length_mm, data = penguins))
```

```
Call:
lm(formula = bill_depth_mm ~ bill_length_mm, data = penguins)

Residuals:
    Min       1Q   Median       3Q      Max
-4.1381 -1.4263  0.0164  1.3841  4.5255

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  20.88547    0.84388   24.749  < 2e-16 ***
bill_length_mm -0.08502    0.01907   -4.459  1.12e-05 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 1.922 on 340 degrees of freedom
(2 observations deleted due to missingness)
Multiple R-squared:  0.05525,    Adjusted R-squared:  0.05247
F-statistic: 19.88 on 1 and 340 DF,  p-value: 1.12e-05
```

Split the penguin data

```
# split a data.frame by group, returns a list
split_peng <- group_split(palmerpenguins::penguins,
                           island,
                           species)

# dimensions of each tibble
# returns a list
map(split_peng, dim)
# How many observations per group?
# coerce to a double
map_dbl(split_peng, nrow)
```

```
[[1]]
[1] 44  8

[[2]]
[1] 124  8

[[3]]
[1] 56  8

[[4]]
[1] 68  8

[[5]]
[1] 52  8
```

```
[1] 44 124 56 68 52
```

Coercion from a list to a numeric vector is required by the user.

Map the linear model

- `map(YOUR_LIST, YOUR_FUNCTION)`
- `YOUR_LIST = spl_mtcars`
- `YOUR_FUNCTION` can be an anonymous function (declared on the fly)

```
map(split_peng, function(x) {  
  lm(bill_depth_mm ~ bill_length_mm,  
    data = x)  
})
```

Curly braces can be used for:

- Several code lines
- Wrap lines to improve readability

```
[[1]]
```

```
Call:  
lm(formula = bill_depth_mm ~ bill_length_mm, data = x)
```

```
Coefficients:  
  (Intercept)  bill_length_mm  
      9.6263         0.2244
```

```
[[2]]
```

```
Call:  
lm(formula = bill_depth_mm ~ bill_length_mm, data = x)
```

```
Coefficients:  
  (Intercept)  bill_length_mm  
      5.2510         0.2048
```

```
[[3]]
```

```
Call:  
lm(formula = bill_depth_mm ~ bill_length_mm, data = x)
```

```
Coefficients:  
  (Intercept)  bill_length_mm  
      9.2607         0.2335
```

```
[[4]]
```

To extract r^2

`base::summary()` generates a list

```
lm_all <- summary(lm(bill_depth_mm ~ bill_length_mm,  
  data = penguins))  
str(lm_all, max.level = 1, give.attr = FALSE)
```

```
List of 12  
 $ call      : language lm(formula = bill_depth_mm ~ bill_le  
 $ terms     :Classes 'terms', 'formula' language bill_dept  
 $ residuals : Named num [1:342] 1.139 -0.127 0.541 1.535 3.  
 $ coefficients : num [1:2, 1:4] 20.8855 -0.085 0.8439 0.0191 2  
 $ aliases    : Named logi [1:2] FALSE FALSE  
 $ sigma      : num 1.92  
 $ df         : int [1:3] 2 340 2  
 $ r.squared   : num 0.0552  
 $ adj.r.squared: num 0.0525  
 $ fstatistic  : Named num [1:3] 19.9 1 340  
 $ cov.unscaled : num [1:2, 1:2] 1.93e-01 -4.32e-03 -4.32e-03 9  
 $ na.action   : 'omit' Named int [1:2] 4 272
```

base R or purrr list elements extraction

```
lm_all$r.squared
```

```
[1] 0.05524985
```

```
lm_all[["r.squared"]]
```

```
[1] 0.05524985
```

```
pluck(lm_all, "r.squared")
```

```
[1] 0.05524985
```

Extract r^2 for all groups

```
split_peng %>%  
  map(function(x) lm(bill_depth_mm ~ bill_length_mm,  
                     data = x)) %>%  
  map(summary) %>%  
  map(function(x) pluck(x, "r.squared"))
```

```
[[1]]  
[1] 0.2192052  
  
[[2]]  
[1] 0.4139429  
  
[[3]]  
[1] 0.2579242  
  
[[4]]  
[1] 0.4271096  
  
[[5]]  
[1] 0.06198376
```

purrr::map() shortcuts I

Anonymous functions

- One sided formula create anonymous functions:
 - Define the function using `~ (*)`
 - Use the placeholder `.x` to refer to the current list element (`.x` represents the argument of the anonymous function)

Initial code

```
split_peng %>%  
  map(function(x) lm(bill_depth_mm ~ bill_length_mm,  
                    data = x)) %>%  
  map(summary) %>%  
  map(function(x) pluck(x, "r.squared"))
```

With anonymous function shortcuts (~)

```
split_peng %>%  
  map(~ lm(bill_depth_mm ~ bill_length_mm,  
          data = .x)) %>%  
  map(summary) %>%  
  map(~ pluck(.x, "r.squared"))
```

*: R base next version [4.1](#) is likely to get a shortcut too for anonymous functions ([Luke Tierney, useR!2020](#))

purrr::map() shortcuts II

With shortcuts I

```
split_peng %>%  
  map(~ lm(bill_depth_mm ~ bill_length_mm, data = .x)) %>%  
  map(summary) %>%  
  map(~ pluck(.x, "r.squared"))
```

With shortcuts II

```
split_peng %>%  
  map(~ lm(bill_depth_mm ~ bill_length_mm, data = .x)) %>%  
  map(summary) %>%  
  map("r.squared")
```

Obtain a vector

With shortcuts II

```
split_peng %>%  
  map(~ lm(bill_depth_mm ~ bill_length_mm, data = .x)) %>%  
  map(summary) %>%  
  map("r.squared")
```

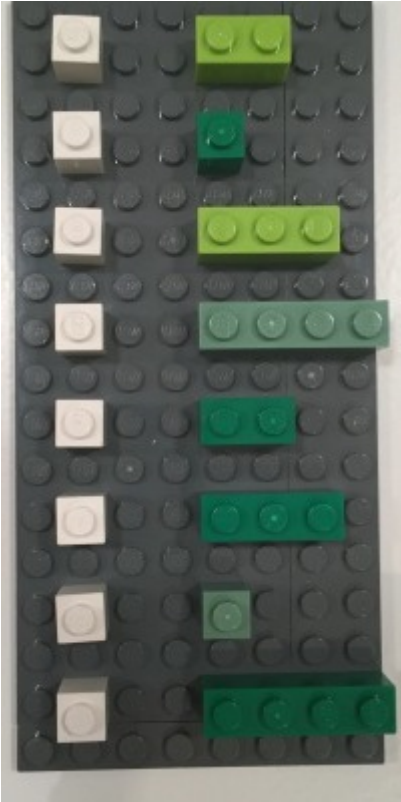
With double coercion

```
split_peng %>%  
  map(~ lm(bill_depth_mm ~ bill_length_mm, data = .x)) %>%  
  map(summary) %>%  
  map_dbl("r.squared")
```

```
[1] 0.21920517 0.41394290 0.25792423 0.42710958 0.06198376
```


Lists as a column in a tibble

Example



```
tibble(numbers = 1:8,  
       my_list = list(a = c("a", "b"), b = 2.56,  
                      c = c("a", "b", "c"), d = rep(TRUE, 4),  
                      d = 2:3, e = 4:6, f = FALSE, g = c(1, 4, 5, 6)))
```

```
# A tibble: 8 × 2  
  numbers my_list  
    <int> <named list>  
1       1 1 <chr [2]>  
2       2 2 <dbl [1]>  
3       3 3 <chr [3]>  
4       4 4 <lgl [4]>  
5       5 5 <int [2]>  
6       6 6 <int [3]>  
7       7 7 <lgl [1]>  
8       8 8 <dbl [4]>
```

Picture by **Jennifer Bryan**

Rewriting our previous example

Nesting the tibble by island and species

```
penguins %>%  
  group_by(island, species) %>%  
  tidyr::nest()
```

```
# A tibble: 5 × 3  
# Groups:   species, island [5]  
  species island data  
  <fct>   <fct>  <list>  
1 Adelie  Torgersen <tibble [52 × 6]>  
2 Adelie  Biscoe    <tibble [44 × 6]>  
3 Adelie  Dream     <tibble [56 × 6]>  
4 Gentoo  Biscoe    <tibble [124 × 6]>  
5 Chinstrap Dream     <tibble [68 × 6]>
```

Rewriting our previous example

With modelling using mutate and map

```
penguins %>%
  group_by(island, species) %>%
  tidyr::nest() %>%
  mutate(model = map(data,
    ~ lm(bill_depth_mm ~ bill_length_mm,
        data = .x)),
    summary = map(model, summary),
    r_squared = map_dbl(summary, "r.squared"))
```

- Very powerful
- Data rectangle
- Next lecture will show you how **dplyr**, **tidyr**, **tibble**, **purrr** and **broom** nicely work together

```
# A tibble: 5 × 6
# Groups:   species, island [5]
  species island data model summary r_squared
  <fct>   <fct> <list> <list> <list> <dbl>
1 Adelie Torgersen <tibble [52 × 6]> <lm> <summary> 0.81
2 Adelie Biscoe <tibble [44 × 6]> <lm> <summary> 0.81
3 Adelie Dream <tibble [56 × 6]> <lm> <summary> 0.81
4 Gentoo Biscoe <tibble [124 × 6]> <lm> <summary> 0.81
5 Chinstrap Dream <tibble [68 × 6]> <lm> <summary> 0.81
```

Without map and dplyr 1.0

With modelling using mutate and list

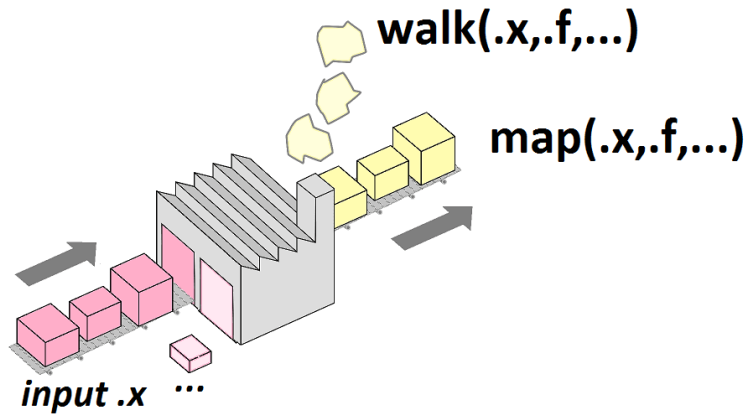
```
penguins %>%  
  nest_by(island, species) %>%  
  mutate(model = list(lm(bill_depth_mm ~ bill_length_mm,  
                        data = data)),  
         summary = list(summary(model)),  
         r_squared = pluck(summary, "r.squared"))
```

- Might be easier for some people
- **rowwise** integrated so no **map**
- **summarise v1.0** handles more than one output per group

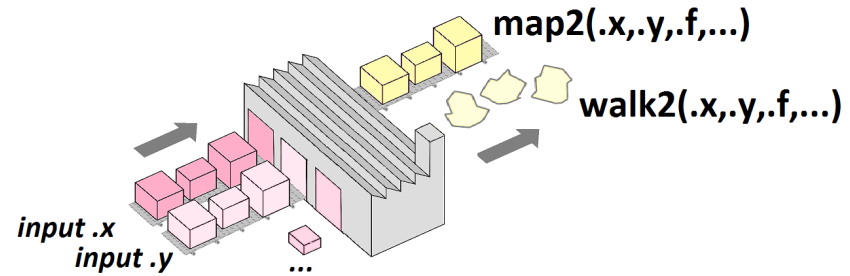
```
# A tibble: 5 × 6  
# Rowwise: island, species  
  island species data model summary r_sq  
  <fct>   <fct>   <list<tibble[,6]>> <list> <list>  
1 Biscoe  Adelie      [44 × 6] <lm> <summary> 0  
2 Biscoe  Gentoo     [124 × 6] <lm> <summary> 0  
3 Dream   Adelie      [56 × 6] <lm> <summary> 0  
4 Dream   Chinstrap  [68 × 6] <lm> <summary> 0  
5 Torgersen Adelie     [52 × 6] <lm> <summary> 0
```

Wrap up

`map()` or `walk()`



`map2()` or `walk2()`



`pmap()`

Pictures from [Lise Vaudor's blog](#)

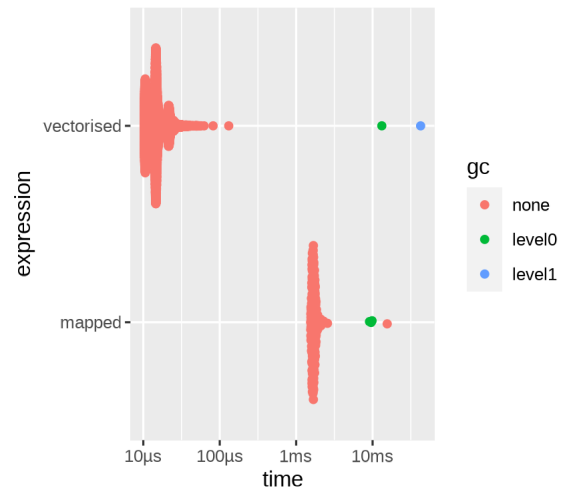
Don't forget vectorisation

Warning

Don't overmap functions! Use `map` only if required (non vectorised function)

```
nums <- sample(1:10,  
              size = 1000,  
              replace = TRUE)  
  
log_vec <- log(nums)  
log_map <- map_dbl(nums, log)  
  
identical(log_vec, log_map)  
  
bench::mark(  
  vectorised = log(nums),  
  mapped = map_dbl(nums, log)) %>%  
  autoplot()
```

[1] TRUE



Before we stop

You learned to:

- Functional programming: focus on **actions**
- **for** loops are fine, but don't write them
- Pass **functions** as **arguments**
- Apprehend nested tibbles with **list-columns**

Acknowledgments ☐ ☐

- Eric Koncina for writing the initial content
- Jennifer Bryan ([LEGO pictures](#), courtesy CC licence)
- Hadley Wickham
- Lise Vaudor
- Ian Lyttle
- Jim Hester

Further reading 📖

- Jennifer Bryan - [lessons & tutorial](#)
- Hadley Wickham - [R for data science](#) ([iteration](#), [many models](#))
- Ian Lyttle - [purrr applied for engineering](#)
- Robert Rudis - [purrr, comparison with base](#)
- Rstudio's blog - [purrr 0.2 release](#) [purrr 0.3 release](#)
- Kris Jenkins - What is Functional Programming? ([Blog version](#) and [Talk video](#))
- Lise Vaudor - [R-atique](#) (*in french*)

Thank you for your attention!