

20171216 Tokyo.R#66@Roppongi




そろそろ  
手を出す purrr



---

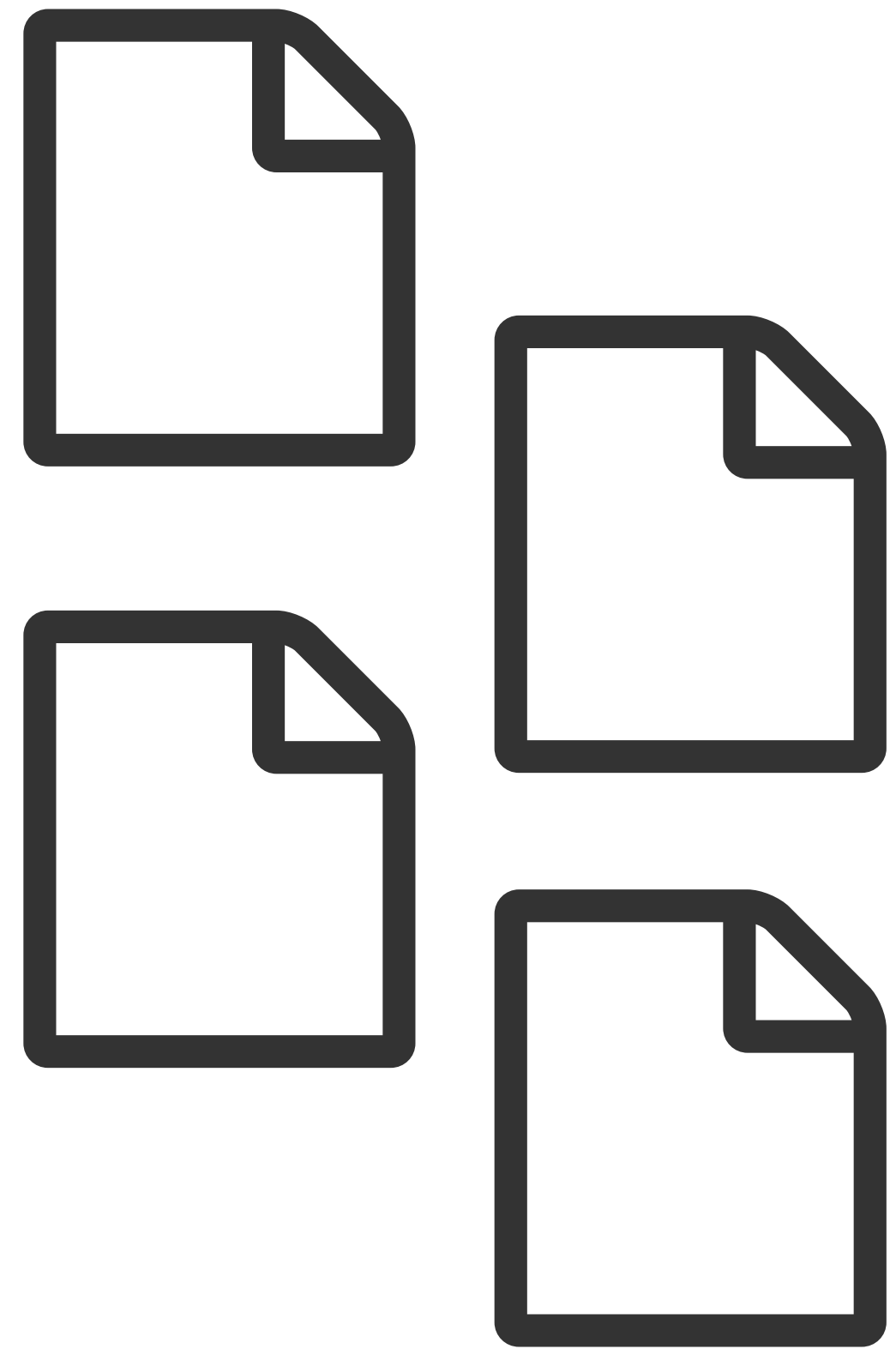
**Purrr that A functional programming toolkit for R**

 **Shinya Uryu (HOXO-M INC.)**

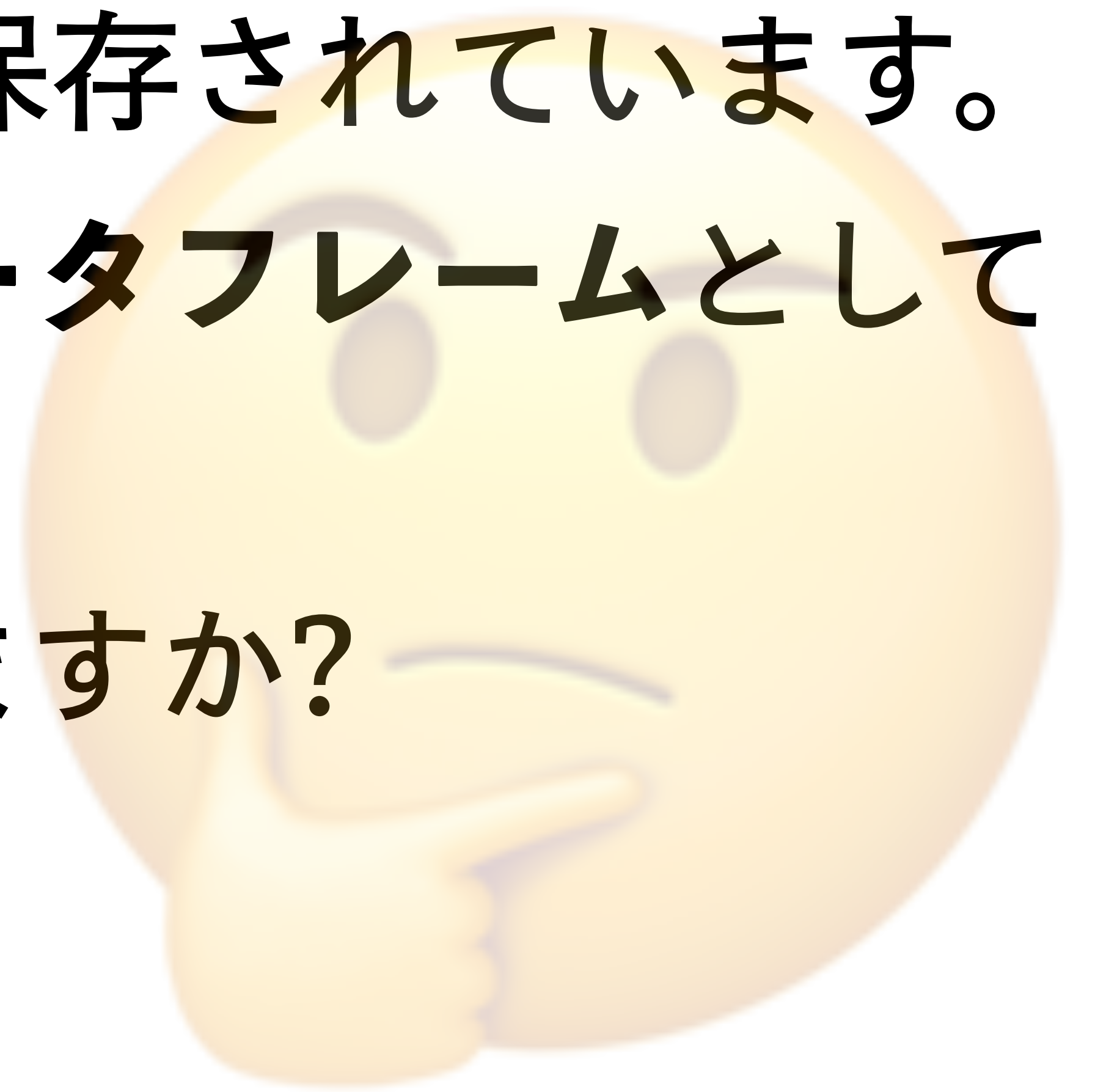
 **@u\_ribo**

 **@uribo**

# YOUR TURN



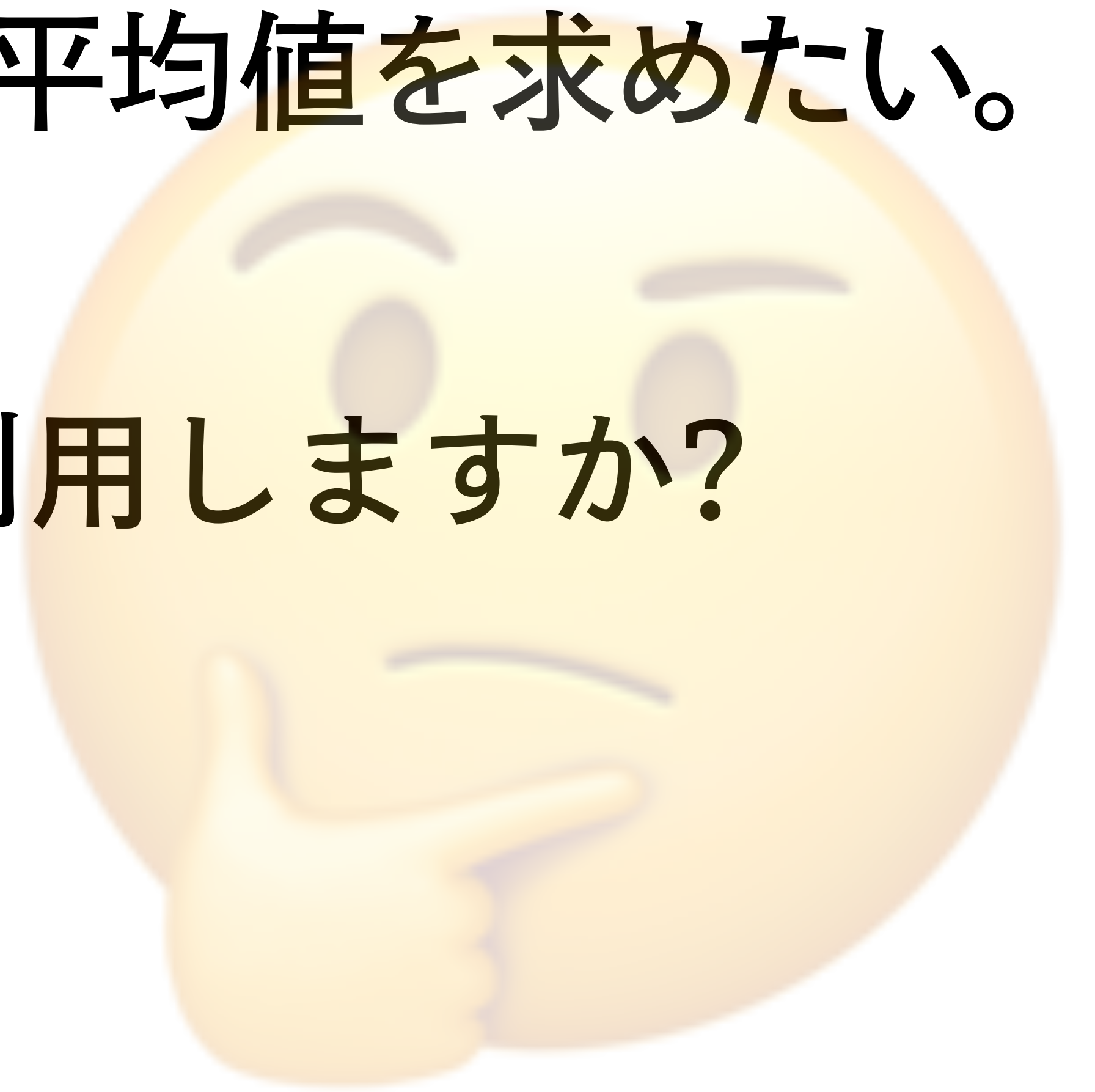
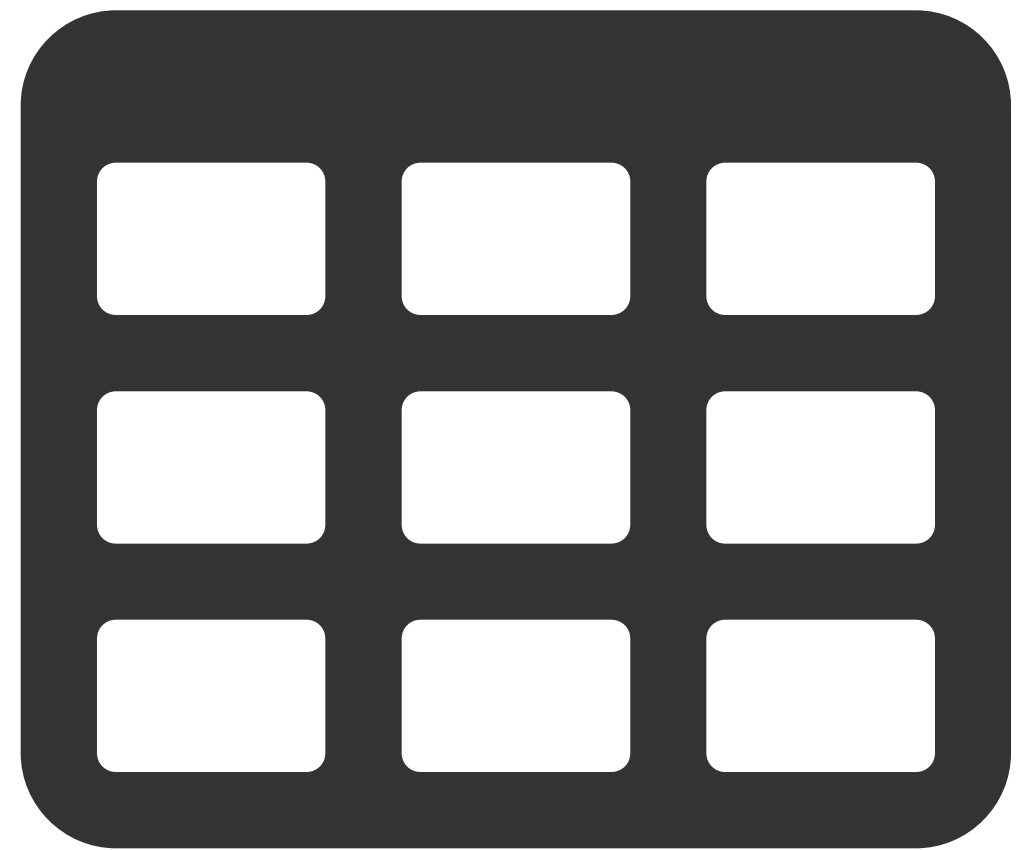
**Q1. 構造が同じ複数のcsvファイルが  
同一のフォルダに保存されています。  
これらを一つのデータフレームとして  
読み込みたい時、  
どのように処理しますか？**



# YOUR TURN

**Q2.** 実数列からなる10列のデータフレームから、各列の平均値を求めたい。

どのような手段を利用しますか？



好きにせい

`for()` によるループ

`apply`族の関数

`dplyr::summarise`

ただしコピペ

でめーはダメだ※

コンピュータが得意なこと

反復処理



マカセテ





# Rでよくある反復処理

- apply族, for関数
- グループ処理

 退屈なことはRに任せよう 

# purrr: 反復処理に対する策の一つ

## ご利益

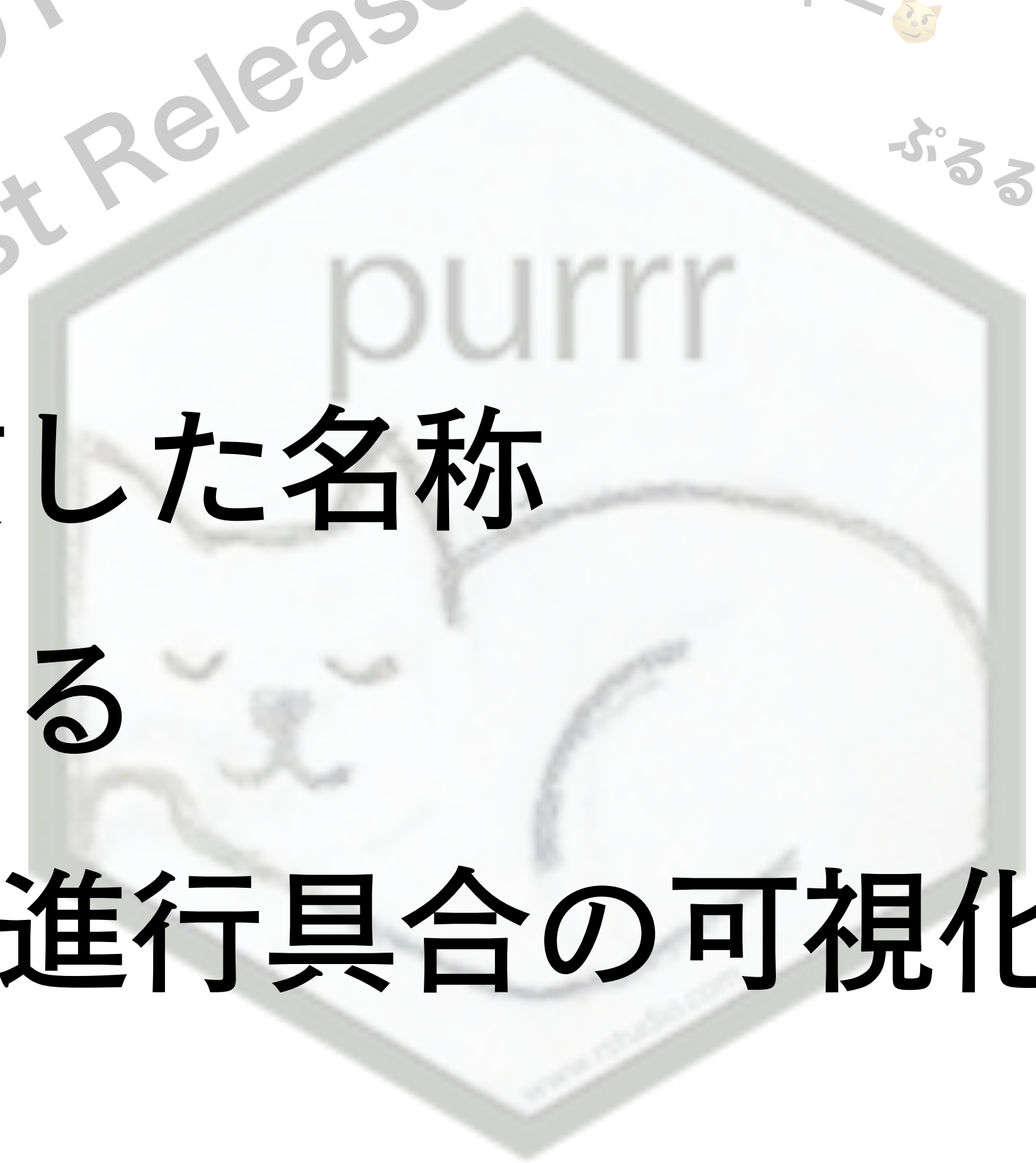
1. 関数や引数の適用に一貫した名称
2. コードの記述を簡略化する
3. (将来的な実装)並列化と進行具合の可視化

since 2015  
(First Release)



ふりゃー🐱

ふるるる🐱



# 或る人の経験



Uryu Shinya

@u\_ribo

purrrってなんだよ、意味ワカンネーというところから、purrrないとどうにもならないのでは...と思わせるあたり、やっぱりハドリーって神だわ

🌐 Translate from Japanese

10:14 AM - 19 Jun 2016



Uryu Shinya

@u\_ribo

今日はpurrrのおかげで仕事が捗ったが、purrrのせいで手間取った 😊

🌐 Translate from Japanese

7:07 PM - 20 Jan 2017



Uryu Shinya

@u\_ribo

purrr使い始めてからapply族の便利さを理解したマン

🌐 Translate from Japanese

4:55 PM - 3 Dec 2016

効能には  
個人差が  
あります



# 安心してください

**purrrむずいから一度で理解しなくてok**

**スピード的にforも悪くないよ（今は）**

**purrrを使うのはコードの保守性を高めるため**

**ってHadleyが言ってたヨ  
（R for Data Science）**

# map()

---



# map: ベクトルの各要素への関数の適用

# 文字数を数える処理 -----

```
x <- c("kazutan", "hoxom", "uribo")
```

```
nchar(x[1]) # 7
```

```
nchar(x[2]) # 5
```

```
nchar(x[3]) # 5
```

```
sapply(x, nchar) # 関数を引数にとる高階関数
```

```
# kazutan hoxom uribo
```

```
# 7      5      5
```

# map: ベクトルの各要素への関数の適用

```
library(purrr)
map(x, nchar)
# [[1]]
# [1] 7

# [[2]]
# [1] 5

# [[3]]
# [1] 5
```

対象 処理  
**map**(**.x**, **.f**, ...)  
リスト 関数  
(データフレーム) 表現式  
ベクトル

データフレームでは列 (ベクトル) が対象



# base::Mapじゃダメなの？

```
Map(nchar, x)
```

```
# $kazutan
```

```
# [1] 7
```

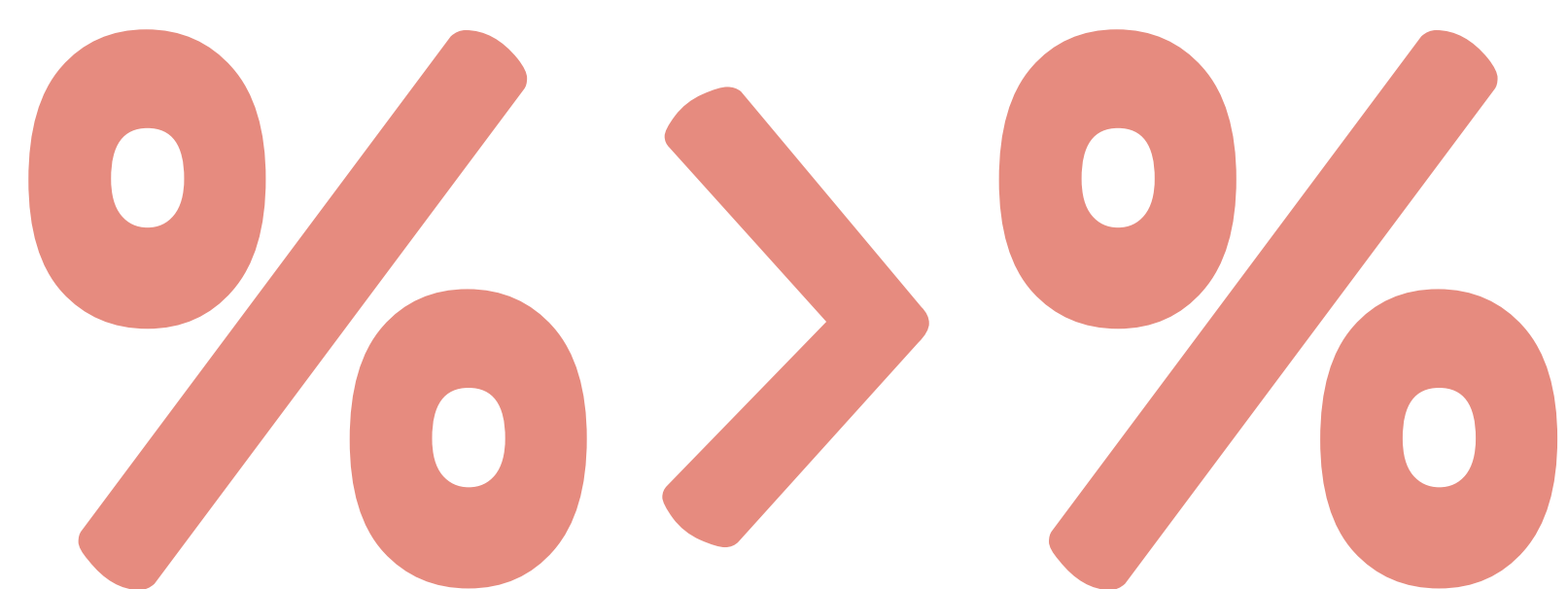
```
# $hoxom
```

```
# [1] 5
```

```
# $uribo
```

```
# [1] 5
```

…まあ良いけど

 脳的には  
NG

第一引数には操作の対象

柔軟な処理に不向き

**map\_\*()**

---



# map色々(1) 返り値を任意の型のベクトルに

返り値のデータ型に応じて適用する関数を変更

FUNCTION	RETURN
<b>map_lgl()</b>	論理型
<b>map_int()</b>	整数型
<b>map_dbl()</b>	実数型
<b>map_chr()</b>	文字列型

# map色々(1) 返り値を任意の型のベクトルに

```
x %>% map_int(nchar)
```

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

```
x %>% map_int(nchar) %>% sum()
```

```
# [1] 17
```

```
x %>% map_chr(nchar)
```

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





# データ型の変換規則に注意

**x %>% map\_lgl(nchar)**

**# Error: Can't coerce element 1 from a integer to a logical**

論理型、整数型、  
倍精度小数点型、文字列の順に柔軟性が高い

# map色々(2) 位置や条件による適用

x %>%

**map\_at(.at = 2, nchar)**

# [[1]]

# [1] "kazutan"

# [[2]]

[1] 5

# [[3]]

[1] "uribo"

x %>%

.pはpredicateを意味する

**map\_if(**

**.p = . == "kazutan",**

**nchar)**

# [[1]]

# [1] 7

# [[2]]

# [1] "hoxom"

# [[3]]

# [1] "uribo"

# 🐾なんか出てきた🐾

```
x %>% map_if(.p = . == "kazutan", nchar)
```

**x[1]: TRUE "kazutan" == "kazutan"**

**x[2]: FALSE "hoxom" == "kazutan"**

**x[3]: FALSE "uribo" == "kazutan"**

## "."は要素の代名詞として機能

magrittrの. と一緒だね（明示的にオブジェクトを与える）

# purrrrにおけるショートカット演算子

# Speciesごとにlm()が実行される

iris %>%

split(.\$Species) %>%

map(function(df) {

lm(Petal.Width ~ Sepal.Length, data = df)

})



# purrrrにおけるショートカット演算子

# 簡略化した記述

iris %>%

split(.\$Species) %>%

map(~ 1. 無名関数を定義

lm(Petal.Width ~ Sepal.Length, data = .))  
})

2. 第一引数の  
オブジェクトが  
渡される

irisのSpeciesごとに処理

# 🐾 どういうことだったよ 🐾

```
res = x %>% map(~nchar)  
res[[1]] %>% class()  
# [1] "function"
```

ただの関数  
(対象が与えられない)

```
res = x %>% map(~nchar(.))  
res[[1]] %>% class()  
# [1] "integer"
```

"."により  
対象が与えられ、  
関数が実行される

# purrrrにおけるショートカット演算子

# 変数の参照

iris %>%

split(.\$Species) %>%

map\_dfc(~ mean(.\$Sepal.Width))

# A tibble: 1 x 3

setosa versicolor virginica

<dbl>

<dbl>

<dbl>

1 3.428

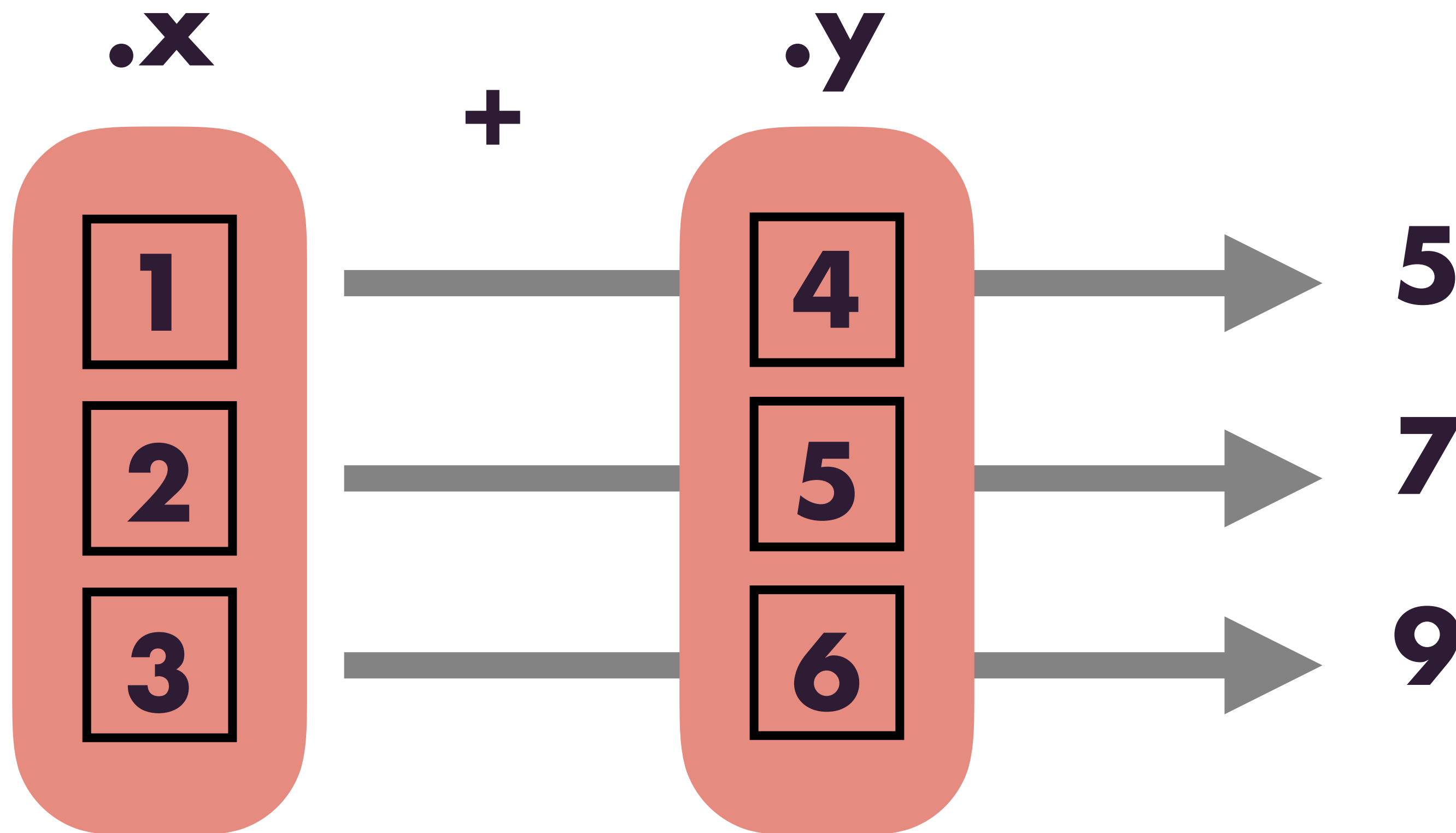
2.77

2.974

# map色々(3) 引数ごとに異なる対象を指定

```
map2_int(.x = 1:3, .y = 4:6, .f = `+`)
```

```
# [1] 5 7 9
```



要素の  
位置ごとに  
並列処理



# 適用する関数の引数への値の渡し方

```
# rnorm(n = 3, mean = 0, sd = 1)
```

3つの引数が定義される

```
# meanとsdの値を変更し、
```

```
# nは固定した正規分布に従う乱数を生成
```

```
map2(.x = c(0, -1, 1), # meanに適用
```

```
  .y = c(1, 1.5, 2), # sdに適用
```

```
  .f = rnorm,      .fはfunctionを意味する
```

```
  n = 3)
```

# 適用する関数の引数への値の渡し方

# 無名関数化して、明示的に名前を記述しても良い

```
map2(.x = c(0, -1, 1), # mean  
     .y = c(1, 1.5, 2), # sd  
     .f = ~ rnorm(mean = .x, sd = .y, n = 3))
```

# 位置も利用可能(前ページの例同様)

```
map2(.x = c(0, -1, 1), # mean  
     .y = c(1, 1.5, 2), # sd  
     .f = ~ rnorm(n = 3, .x, .y,))
```

# map色々(3) 引数ごとに異なる対象を指定

```
library(ipmesh)
```

```
# 緯度経度からメッシュコードを返却
```

```
# メッシュコードはメッシュサイズに応じて桁数が異なる
```

```
coords_to_mesh(longitude = 141.3468,  
               latitude   = 43.06462,  
               mesh_size  = "80km") 3つの引数が定義される
```

```
# [1] "6441"
```

```
coords_to_mesh(141.3468, 43.06462, "1km")
```

```
# [1] "64414277"
```

# map色々(3) 引数ごとに異なる対象を指定

```
d <- tibble::data_frame(  
  longitude = c(141.3468, 139.6917, 139.7147),  
  latitude  = c(43.06462, 35.68949, 35.70078),  
  mesh_size = c("80km", "1km", "500m"))
```

```
d %>%  
  pmap_chr(coords_to_mesh)  
# [1] "6441"      "53394525"  "533945471"
```

# map色々(3) 引数ごとに異なる対象を指定

d %>%

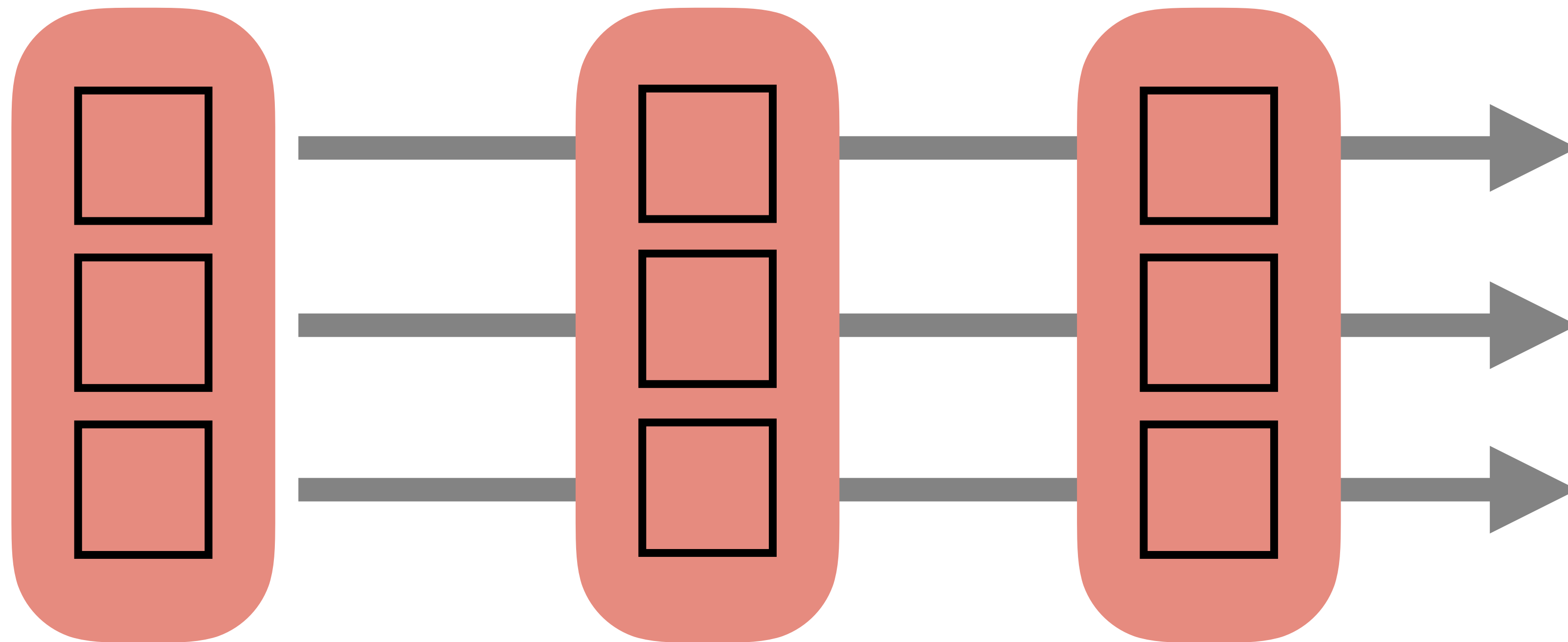
pmap\_chr(coords\_to\_mesh)

関数の引数名と  
データフレームの列名が一致

longitude

latitude

mesh\_size



名前  
または  
位置で指定

# map色々(3) 引数ごとに異なる対象を指定

# 名前で一致するのでこのデータフレームでもOK

```
d = tibble::data_frame(  
  mesh_size = c("80km", "1km", "500m"),  
  longitude  = c(141.3468, 139.6917, 139.7147),  
  latitude   = c(43.06462, 35.68949, 35.70078))
```

# 要素の位置で一致するのでこのデータフレームでもOK

```
d = tibble::data_frame(  
  lon  = c(141.3468, 139.6917, 139.7147),  
  lat  = c(43.06462, 35.68949, 35.70078),  
  size = c("80km", "1km", "500m"))
```



**20171216 Tokyo.R#66@Roppongi**

**purrr +  
tidyverse**

**※purrrもlibrary(tidyverse)でロードされます**

# 冒頭の問題をtidyverseで

## 複数のcsvから一つのデータフレーム

```
# irisデータセットをSpeciesごと(50行ずつ)に保存したcsv
(target_files = list.files("data/",
                           pattern = ".csv$",
                           full.names = TRUE))
target_files %>% map_df(readr::read_csv)
# A tibble: 150 x 5
# ...
```



# 冒頭の問題をtidyverseで

## 複数列の平均値を求める

# 全変数が実数型であれば

```
df %>% map_df(mean)
```

# 実数列だけを対象に

```
df %>% map_if(is.double, mean)
```

# グループ別の図をサクッと作成

```
walk2(paste0("img_", unique(iris$Species), ".png"),
iris %>%
  split(.$Species) %>%
  map(~ggplot(., aes(Sepal.Length, Petal.Width))
+ geom_point()),
ggsave,
# ggsave()に渡す引数 (固定値)
width = 4, height = 3)
```

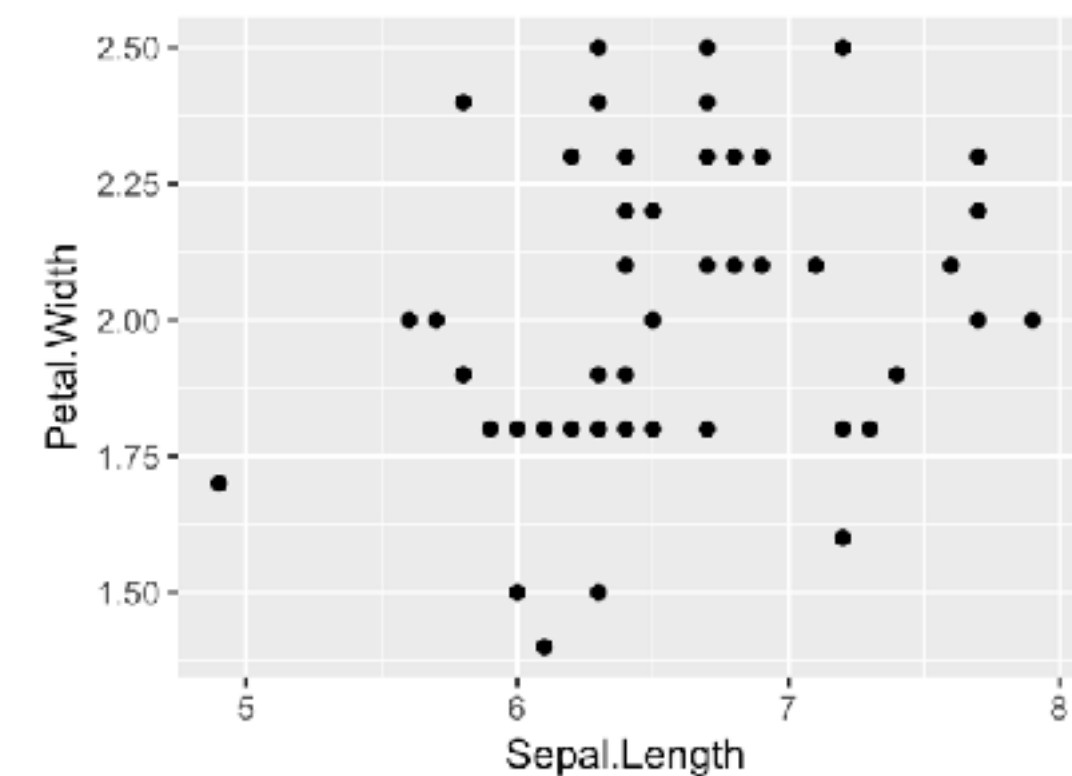
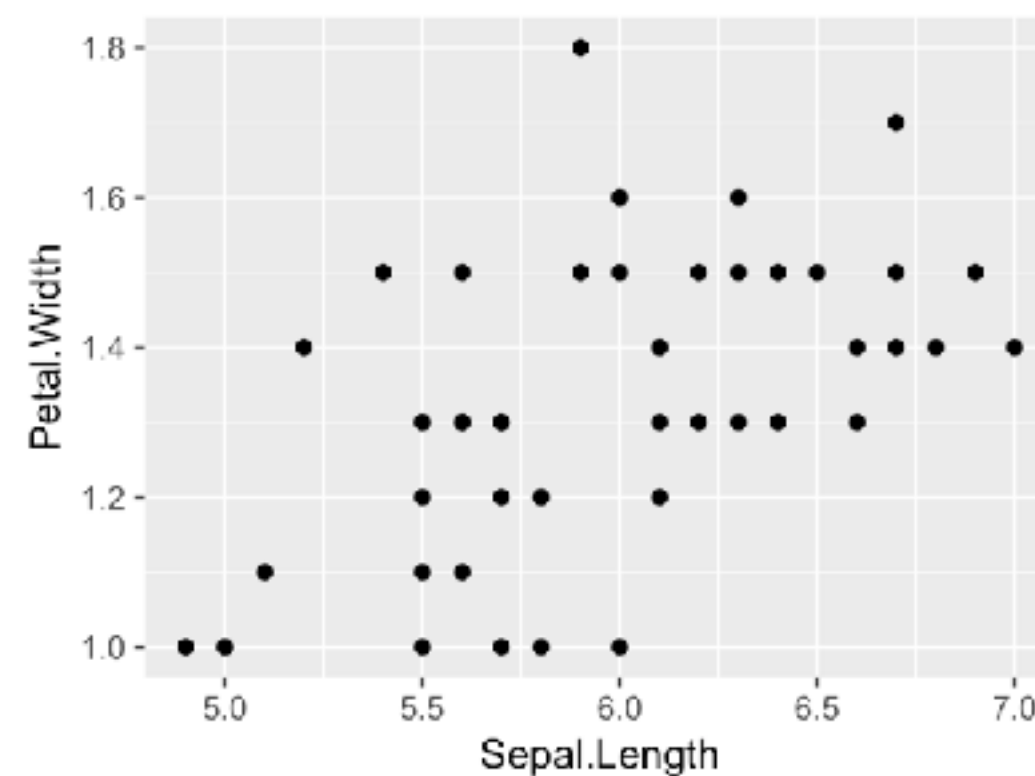
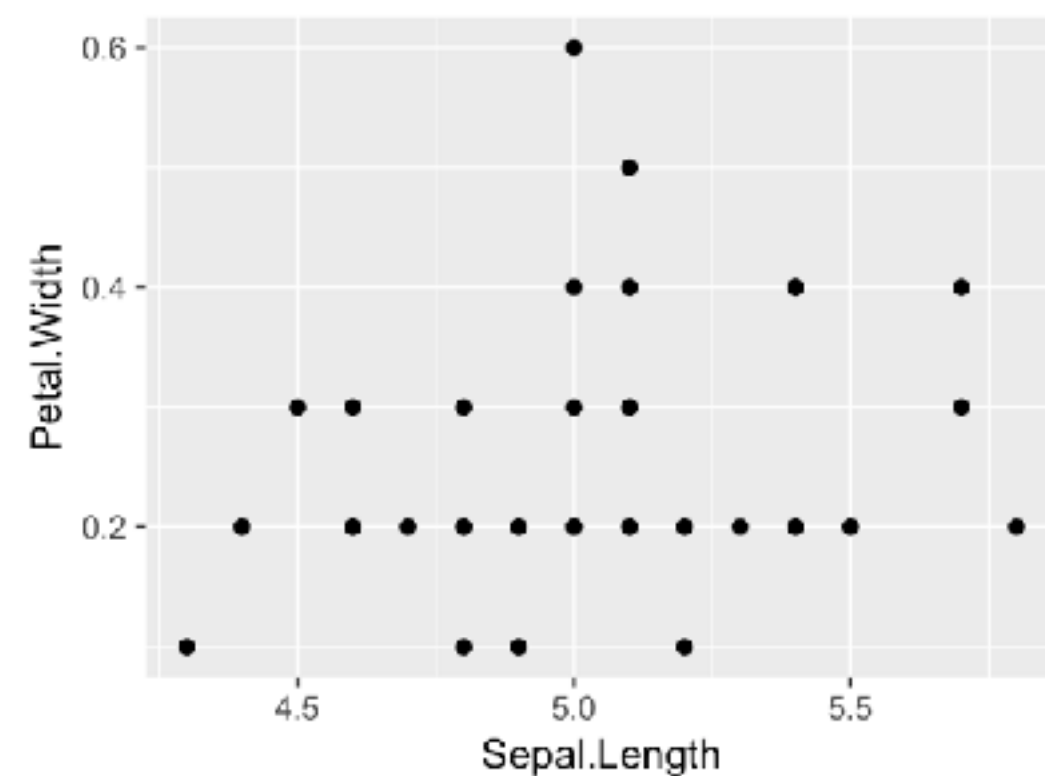


# ん? 何も表示されない?

## walk()は出力を伴わない

```
map2_int(1:3, 4:6, `+`)
```

```
walk2(1:3, 4:6, `+`) for side-effect
```



# dplyrの関数内でmap()

## 1kmメッシュの値を含んだデータ

```
df_mesh = read_csv("data/mesh_1km.csv",  
                    col_types = "c")  
df_mesh %>% sample_n(3L) # A tibble: 3 x 1  
                        mesh_1km  
                        <chr>  
1 36234703  
2 36235603  
...
```



# dplyrの関数内でmap()

```
ipmesh::mesh_to_coords(  
  meshcode = df_mesh$mesh_1km[1])  
# A tibble: 1 x 4  
# ...  
  
# meshcode引数に第一列が与えられる  
  
df_mesh_map = df_mesh %>%  
  mutate(out = pmap(., ~  
    ipmesh::mesh_to_coords(meshcode = .x)))
```

.xにはmesh\_1kmが入る

お?入ったけども?

# 階層構造のあるデータフレーム

```
df_mesh_map
```

```
# A tibble: 60 x 2
```

```
mesh_1km      out
```

```
<chr>        <list>
```

```
1 36225745 <tibble [1 x 4]>
```

```
2 36225746 <tibble [1 x 4]>
```

```
3 36225755 <tibble [1 x 4]>
```



# 階層構造のあるデータフレーム

## 中身はデータフレーム

```
df_mesh_map$out[[1]]
```

```
# A tibble: 1 x 4
```

lng_center	lat_center	lng_error	lat_error
<dbl>	<dbl>	<dbl>	<dbl>

1	122.94375	24.45416666667	0.0062499999999999999
			0.00416666667

# 階層構造のあるデータフレーム

## 展開するには `tidyr::unnest()`

```
df_mesh_map %>% tidyr::unnest()
```

```
# A tibble: 60 x 5
```

	mesh_1km	lng_center	lat_center	lng_error	lat_error
	<chr>	<dbl>	<dbl>	<dbl>	<dbl>
1	36225745	122.94375	24.45416666667	0.006249999999999	0.004166666670000
2	36225746	122.95625	24.45416666667	0.006249999999999	0.004166666670000
3	36225755	122.94375	24.46250000000	0.006249999999999	0.004166666666666
4	36225756	122.95625	24.46250000000	0.006249999999999	0.004166666666666
5	36225757	122.96875	24.46250000000	0.006249999999999	0.004166666666666
6	36225759	122.99375	24.46250000000	0.006250000000001	0.004166666666666
7	36225766	122.95625	24.47083333333	0.006249999999999	0.004166666663333
...					

元のデータフレームに列が追加される

# グループに対して $\text{nest} \rightleftharpoons \text{unnest}$

```
iris_nest = iris %>%  
  group_by(Species) %>% nest()  
# A tibble: 3 x 2  
  Species      data  
  <fctr>    <list>  
1  setosa <tibble [50 x 4]>  
2 versicolor <tibble [50 x 4]>  
3 virginica <tibble [50 x 4]>  
all_equal(iris_nest %>% unnest(), # [1] TRUE  
  iris)
```

# グループに対して nest $\rightleftharpoons$ unnest

```
iris_model = iris_nest %>%  
  transmute(out = map(data, function(df) {  
    broom::tidy(lm(Sepal.Length~Petal.Width, data=df))))
```

# A tibble: 3 x 1

out  
<list>

1 <data.frame [2 x 5]>

2 <data.frame [2 x 5]>

3 <data.frame [2 x 5]>



# グループに対してnest⇔unnest

## lm()の結果をデータフレームに

```
iris_model %>% unnest()
```

```
# A tibble: 6 x 5
```

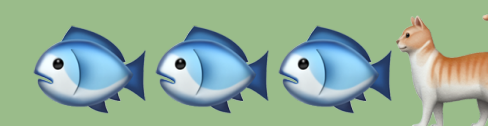
term	estimate	std.error	statistic	p.value
<chr>	<dbl>	<dbl>	<dbl>	<dbl>
1 (Intercept)	4.777177508269	0.123912416859	38.55285555207	8.98396856599e-38
...				

# あとはお好きに！



**20171216 Tokyo.R#66@Roppongi**

# Appendix



**keep(),**  
**discard()**

---



# 要素の保持と除外

```
res = x %>% map_at(1, nchar)
```

```
res %>%
```

```
  keep(~ is.integer(.) == TRUE) %>%
```

```
  as_vector()
```

```
[1] 7 # 要素1の結果が残る
```

```
res %>%
```

```
  discard(~ is.integer(.) == TRUE) %>%
```

```
  as_vector()
```

```
[1] "hoxom" "uribo" # 要素1の結果を除外
```

# invoke()



# 要素に適用する関数を変更

## 関数は文字列として与える

```
library(stringr)
c("str_to_upper", "str_to_title", "str_to_lower")
%>% invoke_map_chr(x)
# [1] "KAZUTAN" "Hoxom"  "uribo"
```

## 各関数への引数はリストで与える

# リストを畳み込み

```
1:3 %>% reduce(`+`)
```

```
# [1] 6
```

```
# 1 + 2 + 3
```

```
x %>% map(nchar) %>%  
  reduce(c)
```

```
# 7 5 5
```

```
# この場合、map_int()が良い
```

個人的に、map\_df()対応していない  
場合に特にオススメ  
(現在のsfパッケージ (v.0.5-5))とか

# partial()

---





# 関数の一部の引数値を固定（部分適用）

## 部分的に同じ処理を施す際に便利

```
set.seed(71)
```

```
f = partial(runif, n = rpois(1, 5), .lazy = FALSE)
```

```
f
```

```
# partial()引数で宣言した引数をもつ関数を作成される
```

```
function (...)
```

```
runif(n = 4L, ...)
```

4はset.seed(71)を与えた時の rpois(1,5)の結果

# 関数の一部の引数値を固定（部分適用）

**f()**

```
# [1] 0.555103868479 0.327369962120  
0.211666960036 0.316121358424
```

# `runif()` の `n` 以外の引数は変更可能

**f(min = 0.2)**

```
# [1] 0.957813141309 0.729371172562  
0.911536924727 0.470407903753
```

# flatten()



# リストの階層を一段上げる

```
# 2階層のリスト
```

```
x = list(  
  list(hijiyama = c("kazutan")),  
  list(tokyo = c("hoxom", "uribo")))
```

```
x %>% flatten()
```

2から1階層のリストに

```
# $hijiyama
```

```
# [1] "kazutan"
```

```
# $tokyo
```

```
# [1] "hoxom" "uribo"
```

# リストの階層を一段上げる

# unlist()とは異なり、一段ずつネストを解消する（より安全）

**x %>% unlist()**

# hijiyama tokyo1 tokyo2

# "kazutan" "hoxom" "uribo"

**x %>% flatten() %>% flatten\_chr()**

# [1] "kazutan" "hoxom" "uribo"

階層がなくなり、ベクトル（文字列）として返却

**20171216 Tokyo.R#66@Roppongi**

**ENJOY**  