Iteration - r4ds

Tomoya Fukumoto

2019-09-06

Iteration

繰り返し作業をどうやって自動化するための二つの手法

1. ループ
2. 関数型プログラミング (functional programming)

準備

```
library(tidyverse)
```

ループに関わるのは base ライブラリ

FP に関わるのは purrr ライブラリ

21.2 For loops

最も標準的なループ

例：各行の median を求める（ループなし）

```
df <- tibble(
          a = rnorm(10),
          b = rnorm(10),
          c = rnorm(10),
          d = rnorm(10)
)
median(df$a)
median(df$b)
median(df$c)
median(df$d)
```

例：各行の median を求める（ループ）

```
output <- vector("double", ncol(df))  # 1. output
for (i in seq_along(df)) {            # 2. sequence
  output[[i]] <- median(df[[i]])      # 3. body
}
output
```

```
## [1] -0.2082671 -0.3499398 -0.4117896 -0.3206133
```

ループの構成要素 output

```
output <- vector("double", ncol(df))
```

- ループの出力の器
- ループが始まる前に作る
- vector 関数で型と長さを指定する
- 長さを指定しないと処理が遅くなる

ループの構成要素 sequence

```
i in seq_along(df)
```

- どうループを回すか
- 一周するたびに `i` がベクトル `seq_along(df)` の中で値を変化させる
- `seq_along(df)` は `1:length(df)` とほぼ同じ
  - 今回は `c(1,2,3,4)`
  - `length(df)` が `0` のときだけ違う

ループの構成要素 body

```
output[[i]] <- median(df[[i]])
```

- ループで実際に処理する内容
  - 1 回目は `output[[1]] <- median(df[[1]])`
  - 2 回目は `output[[2]] <- median(df[[2]])`
  - …

21.2.1 Exercises

1 を見てみる

21.3 For loop variations

1. オブジェクトを修正するループ（作成するのではなく）
2. 要素の名前や値で回すループ（インデックスではなく）
3. 出力の長さが不明の場合
4. ループ回数が不明の場合

21.3.1 Modifying an existing object

```
df <- tibble(a = rnorm(10), b = rnorm(10), c = rnorm(10), d
rescale01 <- function(x) {
    rng <- range(x, na.rm = TRUE)
  (x - rng[1]) / (rng[2] - rng[1])
}
df$a <- rescale01(df$a)
df$b <- rescale01(df$b)
df$c <- rescale01(df$c)
df$d <- rescale01(df$d)
```

等価

```
for (i in seq_along(df)) {
    df[[i]] <- rescale01(df[[i]])
}
```

21.3.2 Looping patterns

ループを回すときの sequence の作法

1. `for (i in seq_along(xs))` インデックスとして 1 から順に数え上げる
   - 最も一般的かつ汎用的
2. `for (x in xs)` ベクトル `xs` の要素を一つずつとる
   - 出力を保存しにくい
     - オブジェクト操作でなく副作用 (plot, write など) に使う
3. `for (nm in names(xs))` 要素の名前を一つずつとる
   - 要素の名前を使いたい場合
     - plot のタイトルとか

インデックスを使った方法が最も汎用的

インデックスを使った方法は他の２つをシミュレートできる

```
for (i in seq_along(x)) {
  name <- names(x)[[i]]
  value <- x[[i]]
}
```

21.3.3 Unknown output length

ループする前に出力の長さがわからないとき

乱数でベクトルの長さが変わる場合

ダメな例

```
means <- c(0, 1, 2)
output <- double()
for (i in seq_along(means)) {
  n <- sample(100, 1)
  output <- c(output, rnorm(n, means[[i]]))
}
```

毎回全データコピーするので $O(n^2)$ の計算量がかかる

良い例

```
out <- vector("list", length(means))
for (i in seq_along(means)) {
  n <- sample(100, 1)
  out[[i]] <- rnorm(n, means[[i]])
}
output <- unlist(out)
```

他の例

- paste(out, x) と追加していくのではなく、out をまとめて作ったあと paste(out, collapse=TRUE) でくっつける
- データフレームの行を追加していくのではなく、リストで作って bind_rows(output) でくっつける

21.3.4 Unknown sequence length

事前にループする回数がわからないとき

while

```
while (condition){
  #body
}
```

シミュレーションでよく使う

- 精度が出るまで反復する、とか

21.3.5 Exercises

21.4 For loops vs. functionals

関数型プログラミングとは

for loop を使った例

```
df <- tibble(a = rnorm(10),
             b = rnorm(10),
             c = rnorm(10),
             d = rnorm(10))

output <- vector("double", length(df))
for (i in seq_along(df)) {
    output[[i]] <- mean(df[[i]])
}
output
```

```
## [1]  0.04440174 -0.64643055 -0.06354274 -0.68199421
```

何回も使いそうだから関数にする

```
col_mean <- function(df) {
    output <- vector("double", length(df))
  for (i in seq_along(df)) {
        output[i] <- mean(df[[i]])
    }
    output
}
```

平均以外の集計でも使うだろう

```
col_median <- function(df) {
    output <- vector("double", length(df))
  for (i in seq_along(df)) {
        output[i] <- median(df[[i]])
    }
    output
}
col_sd <- function(df) {
    output <- vector("double", length(df))
  for (i in seq_along(df)) {
        output[i] <- sd(df[[i]])
    }
    output
}
```

二回以上のコピペをしてしまった！

関数型プログラミング

```
col_summary <- function(df, fun) {
    out <- vector("double", length(df))
  for (i in seq_along(df)) {
        out[i] <- fun(df[[i]])
    }
    out
}
col_summary(df, median)
col_summary(df, sd)
```

R では関数の引数として関数を渡すことができる

⇒ 関数型プログラミング

21.4.1 Exercises

1,2

21.5 The map functions

for loop を functional で置き換える関数 `purrr::map`

purrr::map

Usage

```
map(.x, .f, ...)
```

Arguments

.x	list か atomic vector
.f	関数か formula
…	関数に渡すパラメータ。たとえば `na.rm = TRUE`

return

ベクトル `.x` の要素をそれぞれ関数 `.f` に適用し、その結果をリストで返す

Example 1

```
sheets <- c( " (NAP) 全国",
         " (NAP) 関西",
         " (NAP) 関西以外")

poo <- map(sheets,
       read_excel,
       path = "input/ 【weekly】 POO_Flex_Histrical_Data.xls
       trim_ws = F)
```

map 一族

戻り値をリストではない形式に変換して返すこともできる

- `map` makes a list.
- `map_lgl` makes a logical vector.
- `map_int` makes an integer vector.
- `map_dbl` makes a double vector.
- `map_chr` makes a character vector.
- `map_df` makes a dataframe **特に重要**

これらが使えるかは`.f`の戻り値しだい

Example 2

```
data_all <- left_join(data_used_mean,data_used_sgm,by=g
  left_join(data_used_sgm2,by=grp) %>%
  left_join(data_used_max,by=grp) %>%
  right_join(data_used_att,by=grp) %>%
  map_df(prod_chkin) #x ベクトル内の全ての要素に対して関
```

prod_chkin は dataframe の各変数をベクトルからベクトルに変換する関数

map でやると何がいいのか

- 読みやすい、書きやすい
  - ループは本来処理したいことに対して余計なモノが多すぎる
- 処理が早い
  - と昔は言われていたが今はそうでもないらしい
  - 内部的には C で処理しているから少し早い

21.5.1 Shortcuts

`map` を使う上でのテクニック

余裕のあるやり方

関数を定義する

```
mylm <- function(df){
  lm(mpg ~ wt, data = df)
}
models <- mtcars %>%
  split(.$cyl) %>%
  map(mylm)
```

せっかちなやり方

無名関数

```
models <- mtcars %>%
  split(.$cyl) %>%
  map(function(df)lm(mpg ~ wt, data = df))
```

formula

```
models <- mtcars %>%
  split(.$cyl) %>%
  map(~lm(mpg ~ wt, data = .))
```

リストの各要素から情報を取り出す

```
models %>%
  map(summary) %>%
  map_dbl(~.$r.squared)
```

```
##         4         6         8
## 0.5086326 0.4645102 0.4229655
```

shortcut

```
models %>%
  map(summary) %>%
  map_dbl("r.squared")
```

21.5.2 BaseR

`lapply`, `sapply`, `vapply` という関数が base にある。

`map` を学べば不要

21.5.3 Exercises

全部自分でやらなければ身につかない

21.6 Dealing with failure

`map` 関数を使ったプログラムのデバッグ技術

`map` 中に一つエラーがあると全部出力しなくてむかつく

for programmer

例外処理

purrr::safely

関数 safely は関数.f を引数として、.f を修正した関数を返す。

修正された関数は必ず長さ 2 のリストを返す関数で、その戻り値は下記の通り

1. 要素 result は元々の関数.f の結果。ただしその結果がエラーだった場合は NULL
2. 要素 error はエラーオブジェクト。.f の結果がエラーでなければ NULL

Example 1

```
safe_log <- safely(log)
safe_log(10)
```

```
## $result
## [1] 2.302585
##
## $error
## NULL
```

```
safe_log("a")
```

```
## $result
## NULL
##
## $error
## <simpleError in .Primitive("log")(x, base): non-numeric
```

Example 2

```r
x <- list(1, 10, "a")
y <- x %>% map(safely(log))
str(y)
```

```
## List of 3
##  $ :List of 2
##   ..$ result: num 0
##   ..$ error : NULL
##  $ :List of 2
##   ..$ result: num 2.3
##   ..$ error : NULL
##  $ :List of 2
##   ..$ result: NULL
##   ..$ error :List of 2
##   .. ..$ message: chr "non-numeric argument to mathemati
##   .. ..$ call   : language .Primitive("log")(x, base)
##   .. ..- attr(*, "class")= chr [1:3] "simpleError" "erro
```

purrr::transpose

関数 transpose でもっと見易く

```
y <- y %>% transpose
str(y)
```

```
## List of 2
##  $ result:List of 3
##   ..$ : num 0
##   ..$ : num 2.3
##   ..$ : NULL
##  $ error :List of 3
##   ..$ : NULL
##   ..$ : NULL
##   ..$ :List of 2
##   .. ..$ message: chr "non-numeric argument to mathemati
##   .. ..$ call   : language .Primitive("log")(x, base)
##   .. ..- attr(*, "class")= chr [1:3] "simpleError" "erro
```

一族 purrr::possibly

関数.f がエラーのときに返す値を第二引数に指定できる。

```
possibly_log <- possibly(log, NA)
possibly_log(10)
```

```
## [1] 2.302585
```

```
possibly_log("a")
```

```
## [1] NA
```

全くエラーを返さなくなる

21.7 Mapping over multiple arguments

複数入力の `map`

purrr::map2

Usage

```
map2(.x, .y, .f, ...)
```

Arguments

.x	list か atomic vector
.y	list か atomic vector。.x と.y とのそれぞれのベクトルの長さが同じ
.f	関数か formula
…	関数に渡すパラメータ。たとえば `na.rm = TRUE`

return

ベクトル.x,.y の要素をそれぞれ関数.f に適用し、その結果を
リストで返す

Example

```
mu <- list(5, 10, -3)
sigma <- list(1, 5, 10)
map2(mu, sigma, rnorm, n = 5)
```

```
## [[1]]
## [1] 5.489210 5.734225 3.690561 5.714282 4.897131
##
## [[2]]
## [1] 13.599892  4.208838 12.638960  6.875779 26.941896
##
## [[3]]
## [1]  -2.985130 -11.045972  -9.563942   1.819610 -10.467
```

Visualization of map2

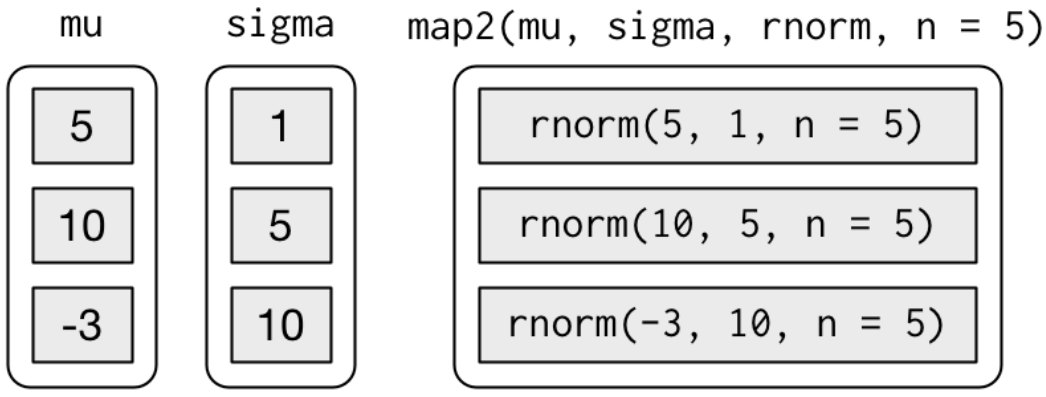

Figure 1: structure of map2

purrr::pmap

入力が３つ以上の場合

```
n <- list(1, 3, 5)
args2 <- list(mean = mu, sd = sigma, n = n)
args2 %>% pmap(rnorm) %>% str()
```

```
## List of 3
##  $ : num 4.17
##  $ : num [1:3] 0.442 7.228 4.561
##  $ : num [1:5] 3.988 4.514 -0.806 1.765 -5.19
```

Visualization of pmap

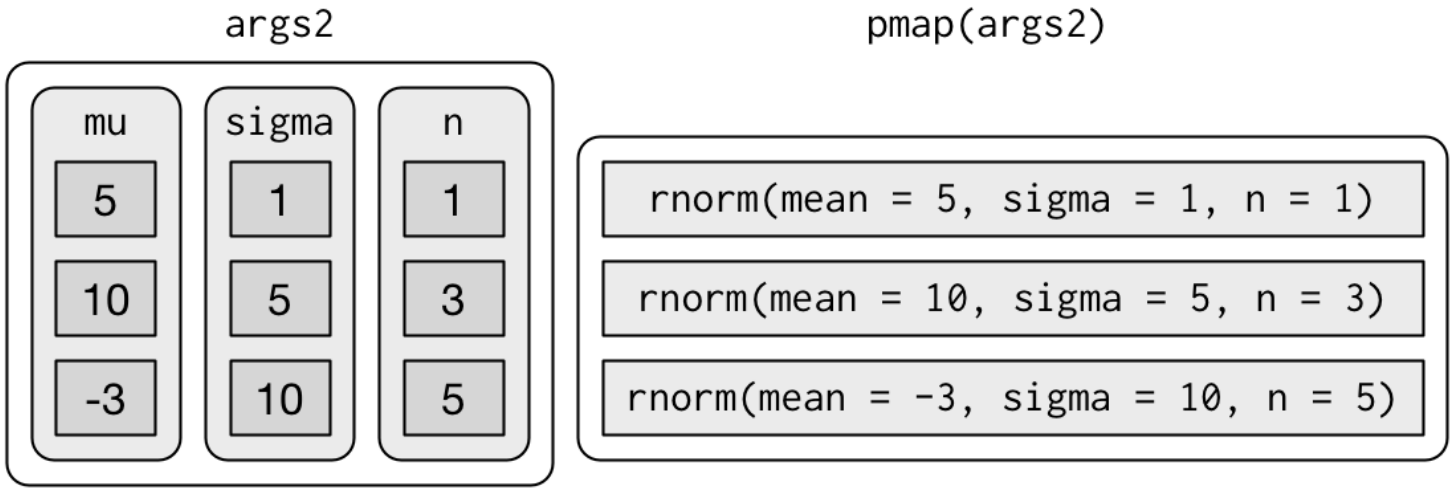

Figure 2: structure of pmap

これでもいいよ

```
params <- tribble(
            ~mean, ~sd, ~n,
            5,     1,   1,
            10,    5,   3,
            -3,    10,  5)
params %>% pmap(rnorm)
```

```
## [[1]]
## [1] 4.882609
##
## [[2]]
## [1] 10.2085411 -0.3950189  4.3204454
##
## [[3]]
## [1] -16.573692  15.466755   1.642132  -1.366528  -2.030
```

21.7.1 Involing different functions

入力変数だけでなく処理する関数すら変えたい場合

Notes!!
現在 `invoke_map` は非推奨になっている。`exec` で代用するべしとある

invoke_map

関数名を文字列でわたす

```
f <- c("runif", "rnorm", "rpois")
param <- list(
          list(min = -1, max = 1),
          list(sd = 5),
          list(lambda = 10))
invoke_map(f, param, n = 5)
```

```
## [[1]]
## [1] -0.0002563163  0.0044804169  0.5215238971  0.652667
##
## [[2]]
## [1] -8.0675832 -0.6323985  3.9644148  4.9581136  3.85336
##
## [[3]]
## [1] 11 11 11 12 14
```

Visualization of `invoke_map`

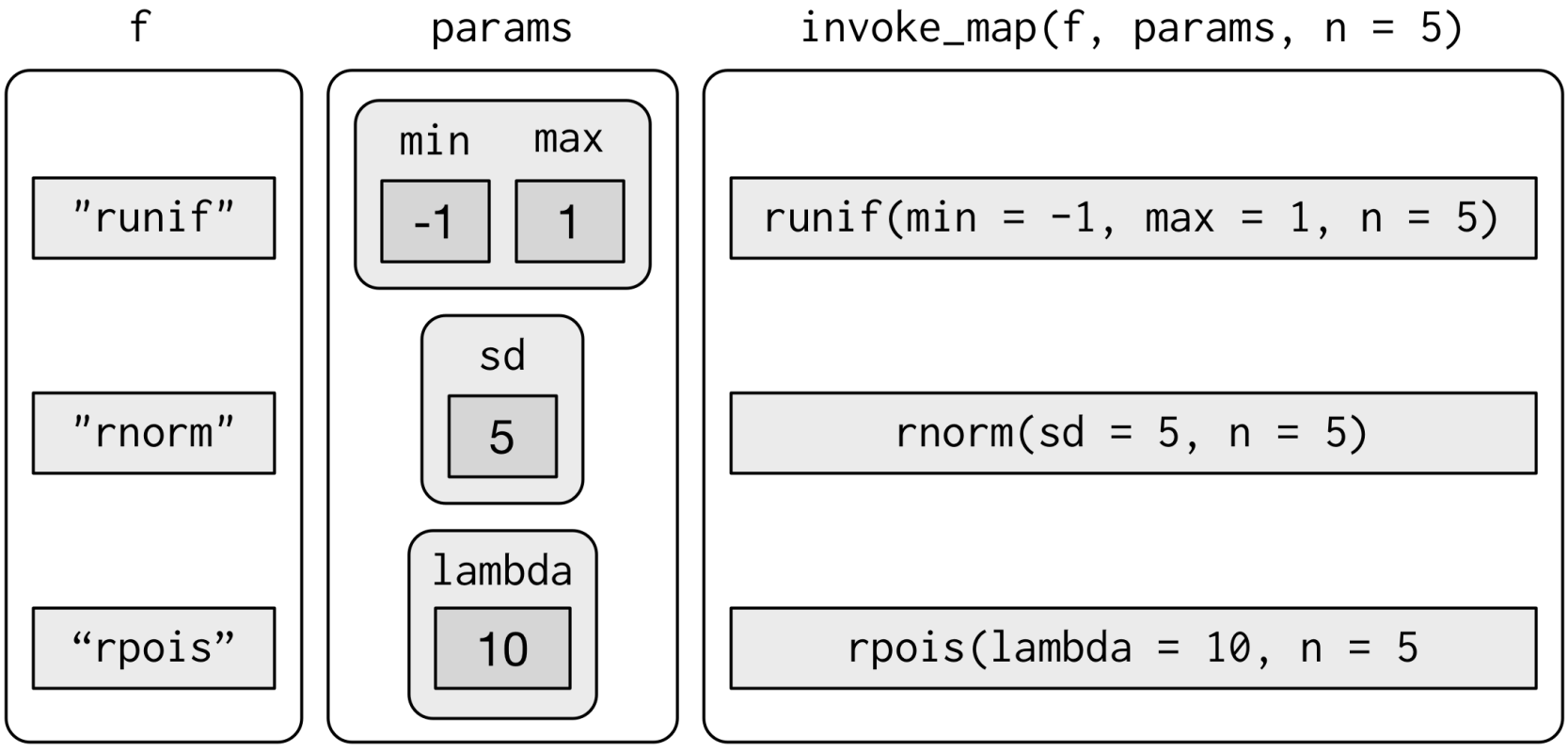

Figure 3: structure of invoke

exec で置き換え

```
# Before:
invoke_map(fns, list(args))
invoke_map(fns, list(args1, args2))

# After:
map(fns, exec, !!!args)
map2(fns, list(args1, args2), function(fn, args) exec(fn,
```

https://github.com/tidyverse/purrr/blob/master/NEWS.md#retirement-of-invoke

21.8 Walk

副作用を目的に FP を

purrr::walk

```r
x <- list(1, "a", 3)

x %>% walk(print)
```

```
## [1] 1
## [1] "a"
## [1] 3
```

purrr::pwalk

```
plots <- mtcars %>%
  split(.$cyl) %>%
  map(~ggplot(., aes(mpg, wt)) + geom_point())
paths <- stringr::str_c(names(plots), ".pdf")
pwalk(list(paths, plots), ggsave, path = tempdir())
```

21.9 Other patterns of for loops

FP のバリエーション

map ほどは使わないが頭のどこかに置いておくといいことがあるかもしれない

21.9.1 Predicate functions

入力関数として出力が論理値の関数を受け付けて特殊な動作をする

purrr::keep と purrr::discard

入力ベクトルの要素のうち、関数の判定が TRUE の要素のみを残す（捨てる）

入力がデータフレームなら dplyr::select_if と同じ

```
iris %>% keep(is.factor) %>% str
```

```
## 'data.frame':    150 obs. of  1 variable:
##  $ Species: Factor w/ 3 levels "setosa","versicolor",..
```

```
iris %>% discard(is.factor) %>% str
```

```
## 'data.frame':    150 obs. of  4 variables:
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0
```

purrr::some と purrr::every

入力ベクトルの要素のうち、関数の判定がどれか（すべて）が TRUE なら TRUE

```
x <- list(1:5, letters, list(10))

x %>% some(is_character)
```

```
## [1] TRUE
```

```
x %>% every(is_vector)
```

```
## [1] TRUE
```

purrr::detect

条件が成立する最初の要素の値を返す。

```
x <- sample(10)
x
```

```
##  [1] 10  9  5  3  1  8  7  6  4  2
```

```
x %>% detect(~ . > 5)
```

```
## [1] 10
```

```
x %>% detect_index(~ . > 5)
```

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

purrr::head_while と purrr::tail_while

条件が成り立つまでの要素をすべて返す。

前から調べるか、後ろから調べるか

```
x %>% head_while(~ . > 5)
```

```
## [1] 10  9
```

```
x %>% tail_while(~ . > 5)
```

```
## integer(0)
```

21.9.2 Reduce and accumulate

２入力１出力関数のみを受け付ける特殊な関数

２つの入力の順序を入れ替えても出力の値が変わらない場合が望ましい

purrr::reduce

```
dfs <- list( age = tibble(name = "John", age = 30),
        sex = tibble(name = c("John", "Mary"), sex = c("M",
        trt = tibble(name = "Mary", treatment = "A"))
dfs %>% reduce(full_join)
```

```
## Joining, by = "name"
## Joining, by = "name"
```

```
## # A tibble: 2 x 4
##   name    age sex   treatment
##   <chr> <dbl> <chr> <chr>
## 1 John     30 M     <NA>
## 2 Mary     NA F     A
```

purrr::accumulate

```
x <- sample(10)
x
```

```
##  [1]  6  4  5  3  1  2  8  7 10  9
```

```
x %>% accumulate(`+`)
```

```
##  [1]  6 10 15 18 19 21 29 36 46 55
```

21.9.3 Exercises

頭の訓練に