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]  0.14917465 -0.38815854  0.06872737 -0.04199707
```

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
- 長さを指定せずとも処理が遅くなる

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)
  - `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] -0.04656869  0.08153351  0.25980515  0.55837785
```

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`

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

全部自分でやる

参考文献