Functions - r4ds

Tomoya Fukumoto

2019-07-26

関数

神のお言葉

データサイエンティストとしてレベルアップする最高の方法は関数を書くこと

関数の利点（コピペに対する）

1. ある処理の塊に名前を付けて管理できる。
   - ▶ コードが理解しやすくなる
2. 変更があったときに一箇所だけ修正すればよい
   - ▶ 生産性 UP!
3. コピペしたときのミスの可能性を減らせる
   - ▶ 不具合の減少

19.2 When you should you write a function?

どういうときに関数を書くのか？

A. コピペの回数が２回を上回るとき

Don’t Repeat Yourself (DRY) principle

関数を書くべき例

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

df$a <- (df$a - min(df$a, na.rm = TRUE)) /
  (max(df$a, na.rm = TRUE) - min(df$a, na.rm = TRUE))
df$b <- (df$b - min(df$b, na.rm = TRUE)) /
  (max(df$b, na.rm = TRUE) - min(df$a, na.rm = TRUE))
df$c <- (df$c - min(df$c, na.rm = TRUE)) /
  (max(df$c, na.rm = TRUE) - min(df$c, na.rm = TRUE))
df$d <- (df$d - min(df$d, na.rm = TRUE)) /
  (max(df$d, na.rm = TRUE) - min(df$d, na.rm = TRUE))
```

関数を書くための stepl コードを分析する

処理の入力は？

```
(df$a - min(df$a, na.rm = TRUE)) /
    (max(df$a, na.rm = TRUE) - min(df$a, na.rm = TRUE))
```

答え

```
x <- df$a
(x - min(x, na.rm = TRUE)) /
  (max(x, na.rm = TRUE) - min(x, na.rm = TRUE))
```

関数を作る

```
rescale01 <- function(x) {
  rng <- range(x, na.rm = TRUE)
  (x - rng[1]) / (rng[2] - rng[1])
}
```

方法

1. 関数の名前を決定する
   - `rescale01`
2. 入力、または引数を `function` の中に入れる
3. 関数の内容を `function(...)` の後に続く `{`のブロックで表現

関数を使ってコードを書き直す

```
df$a <- rescale01(df$a)
df$b <- rescale01(df$b)
df$c <- rescale01(df$c)
df$d <- rescale01(df$d)
```

すっきりした！

まだコピペが残ってるやん

⇒ 次の次の章 iteration を待て

19.2.1 Practices

19.3 Functions are for humans and computers

関数はコンピュータのためだけではなく人間のために書く

- 関数名
- コメント

関数名の決め方

- 短く、しかし処理の内容を明確に表現したい
  - トレードオフになることも多い
  - どちらか一方を選択するとすれば明確にする
    - RStudio のオートコンプリート機能
- 名詞ではなく動詞で
  - 引数は名詞
  - 動詞が “get” や “compute” ならその目的語（名詞）もあり
- よりよい名前が見つかったら変更することをためらうな
- 関数群を作るときは前半部を統一する
  - オートコンプリート

関数名の例

```
# Too short
f()

# Not a verb, or descriptive
my_awesome_function()

# Long, but clear
impute_missing()
collapse_years()
```

関数群の命名

```
# Good
input_select()
input_checkbox()
input_text()

# Not so good
select_input()
checkbox_input()
text_input()
```

命名規則

- `snake_case`
  - Hadley おすすめ
- `camelCase`

どっちでもいいけど、どちらかに統一するべき

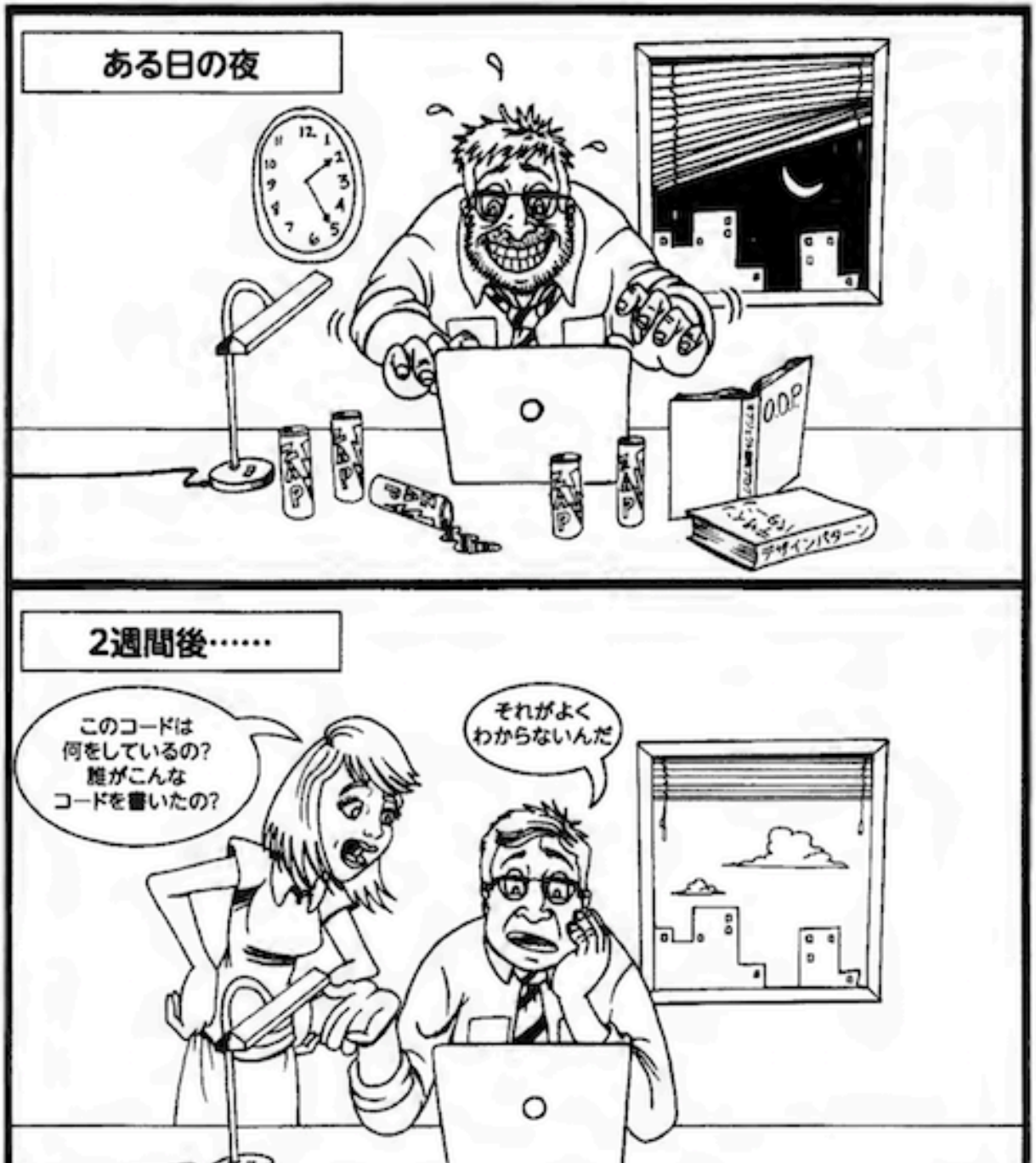
コメント

でその行の # より右側はコメントアウト

```
# this is comment
a <- pi #this also comment
```

- コメントは「なぜ」その処理をするのかを書くべし
  - 「何を」「どのようにして」では無い
- なぜ
  - 中間変数を置いたのか？
  - ２つの関数に分けたのか？
  - 他の方法は試したけどうまくいかなかったのか？
- なぜこのようなコードになったのか？

コメントを書かないと


ある日の夜
O.O.P
デザインパターン
2週間後……
このコードは
何をしているの?
誰がこんな
コードを書いたの?
それがよく
わからないんだ

コードセクション (Rstudio)

次のコードを使えばRStdudioでセクション区切りとみなされる

```
# Load data ------------------------------------

# Plot data ------------------------------------
```

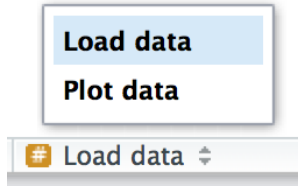


Figure 2: Tstudio

キーボード・ショートカット

Ctrl + Shift + R

19.3.1 Exercises

19.4 Condition execution

if 関数を使った条件分岐

```
if (condition) {
    # code executed when condition is TRUE
} else {
    # code executed when condition is FALSE
}
```

19.4.1 Conditions

```
if (condition) {
    # code executed when condition is TRUE
} else {
    # code executed when condition is FALSE
}
```

condition は TRUE か FALSE かのどちらか

- ベクトルは基本 NO!
  - 入ると warning を出して一番先頭の値だけ採用
  - でも使うのは推奨しません
- NA はエラー

Condition 内での演算

&& や || は AND や OR の役割

実際の働きは前から順に評価して TRUE や FALSE が出たら打ち切り - 後ろにエラーがあっても問題なく走る

ベクトル化関数に注意

- & や | は使うな
  - 関数 any, all
- == も使うな
  - 関数 identical, near

19.4.2 Multiple Contitions

```
if (this) {
    # do that
} else if (that) {
    # do something else
} else {
    #
}
```

19.4.3 Code style

インデントと改行の作法

- {と} の間の行はインデントを入れる
  - 半角スペース 2 個がおすすめ
- {の直前は改行しない。直後は改行
- } の直前に改行し、直後は else の場合を除いて改行

19.4.4 Exercises

19.5 Function arguments

関数の引数

データと詳細

関数の引数は data と details の二種に分類できる

- 関数 `log` は `x` が data で `base` が detail
- 関数 `mean` は `x` が data で `trim`, `na.rm` が details
- 関数 `t.test` は `x`, `y` が data で、`alternative`, `mu`, `paired`, `var.equal`, `conf.level` が detials
- 関数 `str_c` は... が data, `sep`, `collapse` が details

普通は data が先で、details にはデフォルト値が存在

デフォルト値

```
mean_ci <- function(x, conf = 0.95) {
    se <- sd(x) / sqrt(length(x))
  alpha <- 1 - conf
    mean(x) + se * qnorm(c(alpha / 2, 1 - alpha / 2))
}
```

- 一般的な値か安全サイドの値
  - na.rm=FALSE

引数名の指定方法

```r
# Good
mean(1:10, na.rm = TRUE)
mean(na.rm = TRUE, 1:10)

# Bad
mean(x = 1:10, , FALSE)
mean(, TRUE, x = c(1:10, NA))
```

- 関数をコールするとき引数名は省略可能
  - 前から順番に割り当てられる
- 名前を指定すれば順番はめちゃくちゃでもいい

Coding style

- 引数名を指定する = の前後にはスペース入れろ
- 引数値の後ろの , の直前はスペース無し、直後はスペースあり

```
mean(1:10, na.rm = TRUE)
```

19.5.1 Choosing names

一般的な引数名の割当

- `x,y,z` はベクトル
- `w` はベクトルの重み
- `df` はデータフレーム
- `i,j` は整数インデックス
- `n` は長さや行数（自然数）
- `p` は列数

固執する必要は無いが参考にはなる

19.5.2 Checking values

引数の値が正確かチェック

```
wt_mean <- function(x, w) {
    if (length(x) != length(w)) {
        stop("`x` and `w` must be the same length", call.
  }
  sum(w * x) / sum(w)
}
```

限度ってもんがある

```r
wt_mean <- function(x, w, na.rm = FALSE) {
  if (!is.logical(na.rm)) {
    stop("`na.rm` must be logical")
  }
  if (length(na.rm) != 1) {
    stop("`na.rm` must be length 1")
  }
  if (length(x) != length(w)) {
    stop("`x` and `w` must be the same length", call. = FAI
  }
  if (na.rm) {
    miss <- is.na(x) | is.na(w)
    x <- x[!miss]
    w <- w[!miss]
  }
  sum(w * x) / sum(w)
}
```

stopifnot

```r
wt_mean <- function(x, w, na.rm = FALSE) {
  stopifnot(is.logical(na.rm), length(na.rm) == 1)
  stopifnot(length(x) == length(w))
  if (na.rm) {
    miss <- is.na(x) | is.na(w)
    x <- x[!miss]
    w <- w[!miss]
  }
  sum(w * x) / sum(w)
}
```

- エラー条件ではなく、有るべき条件を指定できる

19.5.3 Dot-dot-dot (…)

任意の数の引数を受ける関数

例

```
sum(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
#> [1] 55
stringr::str_c("a", "b", "c", "d", "e", "f")
#> [1] "abcdef"
```

...

```
rule <- function(..., pad = "-") {
  title <- paste0(...)
  width <- getOption("width") - nchar(title) - 5
  cat(title, " ", stringr::str_dup(pad, width), "\n", sep =
}
rule("Important output")
```

```
## Important output ------------------------------------
```

関数がコールされたときに指定された引数で引数名が割当られなかったら... に入る

19.4.5 Lazy evaluation

遅延評価。引数の値は実際に必要とされるまでは評価されない

```
x <- FALSE
f <- function(y = x <- TRUE, z){
  if(z){
    return(x)
  }else if(y){
    return(x)
  }
}
f(z = TRUE)
```

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

```
f(z = FALSE)
```

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

19.5.5 Exercises

19.6 Return values

戻り値

関数をコールした結果得られるオブジェクト

19.6.1 Explicit return statement

デフォルトは関数内で最後に実行されたオブジェクトが戻り値になる

関数 `return` を使って、明示的にリターンを設定することでコードを読みやすくできる

```
complicated_function <- function(x, y, z) {
  if (length(x) == 0 || length(y) == 0) {
    return(0)
  }
  # Complicated code here
}
```

19.6.2 Writing pipeable functions

自作関数をパイプラインに繋げる

二種類のパイプライン接続関数

- 変換
  - 関数の入力データと戻り値とが同一のデータ型だとつないでいける
- 副作用
  - プロットしたり文字列を表示したり
  - 戻り値は入力データそのもの。しかし invisible であるべき

副作用の例

```
show_missings <- function(df) {
  n <- sum(is.na(df))
  cat("Missing values: ", n, "\n", sep = "")
  invisible(df)
}
```

19.7 Environment

R の環境

レキシカルスコープ

```
f <- function(x) {
  x + y
}
y <- 100
f(10)
```

```
## [1] 110
```

```
y <- 1000
f(10)
```

```
## [1] 1010
```

参考文献

- https://rion778.hatenablog.com/entry/2015/05/31/175055
- http://adv-r.had.co.nz/Functions.html#lazy-evaluation