リリース 3: ユーザ・インターフェースの実装

市東 亘

平成 30 年 1 月 9 日

目 次

1 ベクトルとリストの復習

ベクトルの要素は全て同じ型.

```
v <- 1:5
v

## [1] 1 2 3 4 5

# 全ての要素がより一般的な文字列型に変換される
v[2] <- "あ"
v

## [1] "1"  "あ" "3"  "4"  "5"
```

ベクトルの 2 番目の要素にリストを代入してみる.

```
v <- 1:5
# リストには異なる型を要素に入れられる
lst <- list(a=1:5, b="文字列", c=c("あ", "い"))
lst
```

```
## $a
## [1] 1 2 3 4 5
##
## $b
## [1] "文字列"
##
## $c
## [1] "あ" "い"

v[2] <- lst

## Warning in v[2] <- lst:  置き換えるべき項目数が、置き換える数の倍数ではありませんでした

# v はリストに変換される
v

## [[1]]
## [1] 1
##
## [[2]]
## [1] 1 2 3 4 5
##
## [[3]]
## [1] 3
##
## [[4]]
## [1] 4
##
## [[5]]
## [1] 5

str(v)

## List of 5
##  $ : int 1
##  $ : int [1:5] 1 2 3 4 5
##  $ : int 3
##  $ : int 4
##  $ : int 5
```

上のエラーは，代入先（v[2]）が1つの要素であるのに対し，代入しようとしているリスト（lst）が3つの要素を持っているため，数が合わないというエラーが出ている．ここではlstの1つ目の要素であるlst$aだけが代入される．

上の代入でリストに変換されたvには何でも格納できる．もう一度，2番目の要素にリストを代入してみる．

```
v[[2]] <- lst
v

## [[1]]
## [1] 1
##
## [[2]]
## [[2]]$a
## [1] 1 2 3 4 5
##
## [[2]]$b
## [1] "文字列"
##
## [[2]]$c
## [1] "あ" "い"
##
##
## [[3]]
```

```
## [1] 3
##
## [[4]]
## [1] 4
##
## [[5]]
## [1] 5
```

2 第一級関数

関数はデータと同じように，変数に代入できる．以下の f は 3 つの関数を要素に持つリスト．

```
foo <- function() { cat("関数 foo\n") }
bar <- function() { cat("関数 bar\n") }
who <- function(name) { cat("関数", name, "\n", sep="") }
# c() を使っても自動的にリストに変換されて格納される．関数はそれぞれが独自の型を持つ
f <- c(foo, bar, who)
str(f)

## List of 3
##  $ :function ()
##   ..- attr(*, "srcref")=Class 'srcref'  atomic [1:8] 1 8 1 44 8 40 1 1
##   .. .. ..- attr(*, "srcfile")=Classes 'srcfilecopy', 'srcfile' <environment: 0x7f8f2b6f1910>
##  $ :function ()
##   ..- attr(*, "srcref")=Class 'srcref'  atomic [1:8] 2 8 2 44 8 40 2 2
##   .. .. ..- attr(*, "srcfile")=Classes 'srcfilecopy', 'srcfile' <environment: 0x7f8f2b6f1910>
##  $ :function (name)
##   ..- attr(*, "srcref")=Class 'srcref'  atomic [1:8] 3 8 3 63 8 59 3 3
##   .. .. ..- attr(*, "srcfile")=Classes 'srcfilecopy', 'srcfile' <environment: 0x7f8f2b6f1910>
```

f の要素にアクセスし，通常の関数呼び出しと同じように () をつけると，関数を起動できる．

```
f[[1]]()

## 関数 foo

f[[2]]()

## 関数 bar

f[[3]](" 3rd function!")

## 関数 3rd function!
```

3 高階関数

第一級関数はデータと同じように扱えるため,「関数を引数に取る関数」や「関数を返す関数」を作ることができる．
高階関数の例 1.

- lapply(x, fun) は，リストやベクトルなど連続する要素を持つ x の各要素に関数 fun を適用する．
- lapply() は，$fun(x_1), fun(x_2), \ldots, fun(x_n)$ の結果をリストにして返す．

```r
# 引数を 2 乗して返す関数を定義
sqr <- function (x) {
    x * x
}
```

```r
v <- 1:4
lapply(v, sqr)

## [[1]]
## [1] 1
##
## [[2]]
## [1] 4
##
## [[3]]
## [1] 9
##
## [[4]]
## [1] 16
```

高階関数の例 2.

- sapply(x, fun) は，x の各要素に関数 fun を適用して，可能な限り simple な形（同じ型ならベクトルや配列）にして返す.

```r
v <- 5:20
sapply(v, sqr)

##  [1]  25  36  49  64  81 100 121 144 169 196 225 256 289 324 361 400
```

4 無名関数

高階関数の例で sqr() 関数を定義したが，sqr() 関数を使うのが 1 回だけの場合，わざわざ sqr という名前をつけるのはもったいない.「もったいない」というのは，他の用途で sqr という名前が使えなくなるからだ．例えば，以下のように他の用途に使用すると，sqr() 関数が上書きされてしまう.

```r
sqr <- function(x) {
    x * x
}
sqr <- 5      # sqr は関数ではなく 5 を保持する変数になる！
```

これまで関数名を書いていたところには，関数定義をそのまま書ける.

```r
lapply(1:5, function(x) { x*x })

## [[1]]
## [1] 1
##
## [[2]]
## [1] 4
##
## [[3]]
## [1] 9
##
## [[4]]
```

```
## [1] 16
##
## [[5]]
## [1] 25

# 複数行に書いても良い
sapply(2:25, function(x) {
    x * x
})

##  [1]   4   9  16  25  36  49  64  81 100 121 144 169 196 225 256 289 324
## [18] 361 400 441 484 529 576 625
```

このように名前のない関数を「無名関数（annonymous function）」と呼ぶ.

5 高階関数の応用

- リストの中からある要素だけを取り出してベクトルに結合する.

```
doTask1 <- list(title="新規カード",
                fn=function() {
                    cat ("処理: 新規カード作成中.....\n")
                })
doTask2 <- list(title="テスト開始",
                fn=function() {
                    cat ("処理: テスト中.....\n")
                })
doTask3 <- list(title="成績表示",
                fn=function() {
                    cat ("処理: 成績表示中.....\n")
                })
# doTask は list なので, list に格納する
tasks <- list(doTask1, doTask2, doTask3)
str(tasks)

## List of 3
##  $ :List of 2
##   ..$ title: chr "新規カード"
##   ..$ fn   :function ()
##   .. ..- attr(*, "srcref")=Class 'srcref'  atomic [1:8] 2 20 4 17 20 17 2 4
##   .. .. .. .. ..- attr(*, "srcfile")=Classes 'srcfilecopy', 'srcfile' <environment: 0x7f8f2b7afc78>
##  $ :List of 2
##   ..$ title: chr "テスト開始"
##   ..$ fn   :function ()
##   .. ..- attr(*, "srcref")=Class 'srcref'  atomic [1:8] 6 20 8 17 20 17 6 8
##   .. .. .. .. ..- attr(*, "srcfile")=Classes 'srcfilecopy', 'srcfile' <environment: 0x7f8f2b7afc78>
##  $ :List of 2
##   ..$ title: chr "成績表示"
##   ..$ fn   :function ()
##   .. ..- attr(*, "srcref")=Class 'srcref'  atomic [1:8] 10 20 12 17 20 17 10 12
##   .. .. .. .. ..- attr(*, "srcfile")=Classes 'srcfilecopy', 'srcfile' <environment: 0x7f8f2b7afc78>
```

全タスクの `title` だけ抜き出す.

```
sapply(tasks, function(x){ x$title })

## [1] "新規カード" "テスト開始" "成績表示"
```

6　メニューの表示

- CUI（キャラクタ・ユーザ・インターフェース）
- メニューを表示し，プロンプトでユーザに番号を選択してもらう．
- メニューを表示するには menu() 関数を使う．
- menu() は選択された番号を返す．0 が選択されると終了．

```
menu(c("新規カード", "テスト開始", "成績表示"))
##
## 1: 新規カード
## 2: テスト開始
## 3: 成績表示
##
## Selection: 1
## [1] 1
```

CUI の作り方

(1) メニューとプロンプトを表示．

(2) 入力された番号に応じて，関数を呼び出し処理する．

```
doTask1 <- function() {
    cat ("\n\n 処理: 新規カード作成中.....\n\n")
}
doTask2 <- function() {
    cat ("\n\n 処理: テスト中.....\n\n")
}
doTask3 <- function() {
    cat ("\n\n 処理: 成績表示中.....\n\n")
}
tasks <- list(doTask1, doTask2, doTask3)
```

```
while(TRUE) {
    choice <- menu(c("新規カード", "テスト開始", "成績表示"))
    if (choice == 0)
        return()
    tasks[[choice]]()
}

## 1: 新規カード
## 2: テスト開始
## 3: 成績表示
##
## Selection: 2
##
##
## 処理: テスト中.....
##
```

練習問題

- 上のメニューに，自分の好きな処理名でタスクを追加せよ．

7　変更箇所の分離

- メニューを追加するごとに，複数箇所を変更するのはバグを招く．
- 変更箇所を分離することによって，プログラム修正時の変更を**局所化**する．
  $\Longrightarrow$ 仕様変更に強いプログラムになる．
- メニューの文字列と処理関数を一緒に管理した方が，メニューを追加するときに便利．
- タスク処理の関数と，メニュー文字列をリストにパッケージ化．

```
doTask1 <- list(title="新規カード",
                fn=function() {
                    cat ("\n\n 処理: 新規カード作成中.....\n\n")
                })
doTask2 <- list(title="テスト開始",
                fn=function() {
                    cat ("\n\n 処理: テスト中.....\n\n")
                })
doTask3 <- list(title="成績表示",
                fn=function() {
                    cat ("\n\n 処理: 成績表示中.....\n\n")
                })
# doTask は list なので，list に格納する
tasks <- list(doTask1, doTask2, doTask3)
```

```
# 全タスクの title だけ抜き出す．
menu.items <- sapply(tasks, function(x){ x$title })

while(TRUE) {
    choice <- menu(menu.items)        # <-- 変更点
    if (choice == 0)
        return()
    tasks[[choice]]$fn()              # <-- 変更点
}
```

こうすると，新しいメニューを追加しても，メニュー表示ルーチンは一切変更しなくて済む．

練習問題

- 上のメニューに，4 番目のタスクを追加せよ．

8　アプリケーション本体

メニュー表示のルーチンを関数にまとめれば，次回以降，`run.app()` で起動できる．

```
run.app <- function() {
    # タスクを登録
    tasks <- list(doTask1, doTask2, doTask3)
    # 全タスクの title だけ抜き出す．
    menu.items <- sapply(tasks, function(x){ x$title })

    while(TRUE) {
        choice <- menu(menu.items)
        if (choice == 0)
            return()
        tasks[[choice]]$fn()
    }
}
```