R言語のさらなる理解とデータベース管理関数の修正

市東 亘

平成30年1月9日

1　概観

目 次

2　ポインタとは

```
a <- 1:5
```

- 1から5のベクトルを格納する領域がメモリ上に確保される.
- 変数aの領域がメモリ上に確保される.
- メモリの領域を指し示すアドレス（ポインタ）がaに設定される.

```
a
```

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

- aにアクセスすると，aに格納されているポインタが指し示す先のベクトルが返る.

3 Copy-On-Modify ルール

```
b <- a
```

変数 b に a の値（= ベクトルを指すポインタ）が代入される．この時点で a と b はメモリ上の同じ領域を参照している．

```
a
```

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

```
b
```

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

同じベクトルを参照しているので，a と b は同じ値（ベクトル）を返す．

変数が保持するポインタを調べる address() 関数を使用するために，pryr パッケージをロード．

```
if (!require("pryr")) {       # pryr パッケージをロード
    install.package("pryr")   # ロードに失敗したらインストールしてからロード
    library(pryr)
}
```

```
## Loading required package:  pryr
```

```
address(a)
```

```
## [1] "0x7fd5751fd320"
```

```
address(b)
```

```
## [1] "0x7fd5751fd320"
```

上の結果を見ると，確かに a と b には同じポインタが格納されている．

```
a[3] <- 9999
```

上の代入で生じていること

- a が指し示すベクトルの 3 番目の要素を変更する．
- 変更された「新しいベクトル」が作成され，そのアドレスを指すポインタが a に設定される．
  $\Longrightarrow$ a は b と異なるアドレスを参照している．下のコードの結果を確認せよ！
  $\Longrightarrow$ a の変更は b に影響を与えない．

```
a
```

```
## [1]    1    2 9999    4    5
```

```
b
```

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

```
address(a)
```

```
## [1] "0x7fd573533340"
```

```
address(b)
```

```
## [1] "0x7fd5751fd320"
```

データ構造に変更が加えられると R はコピーを作成しそのコピーを変更する.
$\Longrightarrow$ Copy-on-Modify ルール

4 スコープ

```
a <- 1    # グローバル変数
foo <- function() {
    cat("In foo: a =", a, "\n")  # グローバル変数を参照
    a <- 2                       # foo 関数内でローカル変数が作成される
    cat("In foo: a =", a, "\n")  # ローカル変数を参照
    bar()                        # foo 関数内で bar を起動
}
bar <- function () {
    cat("In bar: a =", a, "\n")  # bar は foo の外にあるので foo のスコープ内の変数は見えない
    a <- 3                       # bar 関数内でローカル変数が作成される
    cat("In bar: a =", a, "\n")  # ローカル変数を参照
}
foo()

## In foo: a = 1
## In foo: a = 2
## In bar: a = 1
## In bar: a = 3

a

## [1] 1
```

上のコードの考察

- bar 関数は foo 関数内で呼び出されているが，bar 関数は foo 関数内の a にはアクセスできない.
- 各変数や関数名にアクセスできる範囲を「スコープ」と呼ぶ.
- foo 関数や bar 関数はグローバル変数は参照できる.
  $\Rightarrow$ 内側のスコープから外側のスコープにはアクセスできる.
- グローバル変数と同じ名前のローカル変数を作成すると，ローカル変数が優先される.
  $\Rightarrow$ 内側のスコープの名前がシャドウ（shadow）する.

```
a <- 1
foo <- function() {
    bar <- function () {              # bar 関数を foo 関数内で定義
        cat("In bar: a =", a, "\n")   # foo 関数のスコープ内なので foo のローカル変数が見える
        a <- 3                        # bar 関数内でローカル変数を作成
        cat("In bar: a =", a, "\n")
    }
    cat("In foo: a =", a, "\n")       # この時点では foo 内の a が作成されてないのでグローバル変数を参照
    a <- 2                            # foo 関数内でローカル変数を作成
    cat("In foo: a =", a, "\n")
    bar()
}
foo()
```

```
## In foo: a = 1
## In foo: a = 2
## In bar: a = 2
## In bar: a = 3

a

## [1] 1
```

上のコードの考察

- `bar` 関数は `foo` 関数内で定義されているので，外側の `foo` 関数内の変数にアクセスできる．

5 Environment

オブジェクト（変数や関数）のスコープ（可視範囲）はどのように決まるのか？

- 関数に入るたびに environment が作成される．
- 関数内で定義されたオブジェクトは全て，その関数の environment に保存される．
- 各 environment は親 environment（外側のスコープの environment）への参照を保持する．
  $\Longrightarrow$ 子 environment は親 environment の中を覗けるが，親 environment は子 environment の中を覗けない．
- REPL のトップレベルの environment は `R_GlobaEnv`（グローバル environment）で，`globalenv()` 関数でアクセスできる．

オブジェクト（変数や関数）へのアクセス時

(1) 現在の environment 内にオブジェクトがあればその値（or 関数）を返す．

(2) 見つからなければ親 environment の中を探しに行く．

(3) 最上位の environment まで探しに行って見つからなければエラーとなる．

オブジェクト（変数や関数）への代入時

(1) 現在の environment 内にオブジェクトがあれば，そのオブジェクトに値（or 関数定義）を代入する．

(2) 見つからなければ現在の environment 内にオブジェクトを作成して値を代入する．
  $\Longrightarrow$ 上位の environment のオブジェクトをシャドウ（マスク）する！

グローバル environment のオブジェクトを更新する方法

```
a <- 1
foo <- function() {
    bar <- function () {
        cat("In bar: a =", a, "\n")
        assign("a", 3, envir=globalenv()) # G_GlobalEnv の a という名前のオブジェクトに代入
        cat("In bar: a =", a, "\n")
    }
    cat("In foo: a =", a, "\n")
    a <- 2
    cat("In foo: a =", a, "\n")
    bar()
}
foo()

## In foo: a = 1
## In foo: a = 2
## In bar: a = 2
## In bar: a = 2

a

## [1] 3
```

```
a <- 1
foo <- function() {
    bar <- function () {
        cat("In bar: a =", a, "\n")
        assign("a", 3, envir=globalenv()) # G_GlobalEnv の a という名前のオブジェクトに代入
        cat("In bar: a =", a, "\n")
    }
    cat("In foo: a =", a, "\n")
    assign("a", -2, envir=globalenv()) # G_GlobalEnv の a という名前のオブジェクトに代入
    cat("In foo: a =", a, "\n")
    bar()
}
foo()

## In foo: a = 1
## In foo: a = -2
## In bar: a = -2
## In bar: a = 3

a

## [1] 3
```

6 純粋関数と副作用

データベースを更新するプログラムは以下のように呼び出した．

```
card.db <- create.card.db()

card.db <- add.card(card.db, "superficial", "うわべだけの，表面的な．", 1)$db
```

上の `add.card()` 関数は**純粋関数（pure function）**．

- 引数で渡された `card.db` を変更しない．⇐ Copy-On-Modify ルール
- 関数の外にある変数を変更しない．
- 引数に渡された値が同じなら，常に同じ結果を返す．

純粋関数とは

- 副作用（side effect）がない．
  - 引数で渡されたオブジェクトを変更しない．
  - 関数の外部の状態を変更しない．
- 参照透過である．
  引数に渡された値が同じなら，常に同じ結果を返す．

副作用のある関数

以下の `add.card2()` 関数は，関数内部でデータベースを更新する．したがって，

(1) 引数に `card.db` を渡す必要がないし，

(2) 返り値を受け取って `card.db` に代入する必要もない．

```
add.card2("superficial", "うわべだけの，表面的な．", 1)  #（定義は示していないので実行できません）
```

考察

- `add.card()` は純粋関数だが，`add.card2()` は純粋関数ではない．
- `add.card2()` は値を受け取って返す以外に，関数外のオブジェクトの値を変更している．
  $\Longrightarrow$ 副作用（side effect）を持つ．

純粋関数の利点

- 純粋関数の動作は入力（引数）のみに依存し，ソフトウェアの状態に依存しないので，関数単体で動作テストが行える．
- 純粋関数ではない関数で書かれたプログラムを読む際は，様々な関数がプログラムの状態を変えていくため，更新される状態に注意しながら読まなければならない．
- 副作用のある関数を変更した場合，プログラム中の影響箇所を全てチェックしなければならない．
- 関数外部の変数を書き換えると，その変数を参照している別の箇所にバグを招く恐れがある．

> ソフトウェアをできるだけ純粋関数で構築するプログラミング手法を，関数型プログラミングと呼ぶ．

これまでに実装した関数はできるだけ純粋関数として実装している．副作用があるのは，

- `save.card.db()`，`save.category.db()`: HD 上のファイルの状態を変化させる．関数の入出力が目的ではない．
- `load.card.db()`，`load.category.db()`: 返る値がデータベースの保存内容に依存するので参照透過ではない．

7　エラー処理

- これまで，データベースの作成，保存，ロードを実装した．
- フラッシュカードアプリを起動した時に，最初にデータベースをファイルから読み込みたい．
  $\Longrightarrow$ 初回起動時にはデータベース・ファイルはまだない！
- 初回か否かチェックする方法を考える代わりに，以下の様に処理すれば良い．
  (1) ファイルを読み込み data.frame を返す．
  (2) ファイルが存在せずエラーが発生したら，単にデータベース用 data.frame を作成し返す．

```
tryCatch (<エラーが発生する可能性のある式>,
          error=<エラーが発生した場合に起動される式>)
```

```
tryCatch(readRDS("myfile.rds"),
         error=function(e){ cat("ファイルがありません") })

## Warning in gzfile(file, "rb"): 圧縮されたファイル 'myfile.rds' を開くことができません，理由は 'No
such file or directory' です

## ファイルがありません
```

エラー処理の方法　R の警告を出したくなければ，suppressWarnings() 関数でエラーが発生する可能性のある関数を囲む．

```
tryCatch(suppressWarnings(readRDS("myfile.rds")),
         error=function(e){ cat("ファイルがありません") })

## ファイルがありません
```

8　データベース・ロード関数の変更

flashcard.R のデータベース・ロード関数を以下の様に変更．ファイルが存在しない場合は，新規データベース用 data.frame を作成して返す．

```
load.card.db <- function (filename) {
    tryCatch(suppressWarnings(readRDS(filename)),
             error=function(e){ create.card.db() })
}
load.category.db <- function (filename) {
    tryCatch(suppressWarnings(readRDS(filename)),
             error=function(e){ create.category.db() })
}
```

9　アプリ用データベースの準備

flashcard.R に以下のコードを追加し，アプリを起動するときに毎回，同じファイルからデータベースを読み込む様にする．追加する位置は，前回追加した CUI コードの前．なぜなら，これらの関数やデータベースは CUI アプリから使用するので，その前で定義しなければならない．

```
# アプリ用データベースファイル名
card.db.filename <- "~/Documents/git-repositories/R-programming-lecnote/flashcard/card-db.rds"
cat.db.filename <- "~/Documents/git-repositories/R-programming-lecnote/flashcard/category-db.rds"

# アプリ専用データベース．グローバル変数として提供．
CARD.DB <- load.card.db(card.db.filename)
CAT.DB <- load.category.db(cat.db.filename)
```

今後はアプリ・コードからグローバル変数のデータベースに直接アクセスして，データの更新を行う．$\Longrightarrow$ 副作用のある関数を使うことになる．ただし使用は最小限にとどめる．

アプリ作成用に作成してきたライブラリ関数と区別するために，カードデータベースとカテゴリデータベースの保存関数を大文字で定義する．

```
# アプリ専用カード保存関数
SAVE.CARD <- function() {
    save.card.db(CARD.DB, card.db.filename)  # 変更をファイルに保存
}

# アプリ専用カテゴリ保存関数
SAVE.CATEGORY <- function () {
    save.category.db (CAT.DB, cat.db.filename)  # 変更をファイルに保存
}
```