リリース6: テスト機能の実装

市東 亘

平成 30 年 1 月 16 日

1　概観

目 次

1 概観 1
2 テスト機能の外部仕様 1
3 スタックの選択 2
4 出題用スタックの準備 3
5 出題カード抽出関数の実装 4
6 カードの表示 5
7 カテゴリ選択補助関数の修正 6
8 正誤処理用補助関数の実装 7
9 フラッシュカード機能の実装 8
10 スタックの表示 9
11 メインメニューへの追加 9
12 ここから先は？ 9

2　テスト機能の外部仕様

- テストするカードのカテゴリを選べるようにする.
- 正解するとスタックを移動する.
- 各スタックからのカード出題率は 5 : 3 : 1.5 : 0.5 とする.
- 空のスタックが選ばれた場合は次の問題へ.

- 間違えるとスタック 1 にカードを戻す.
- 各スタック内では過去の出題数の少ない順にランダムに出題する.

3 スタックの選択

まず，1–4 のスタックを 5 : 3 : 1.5 : 0.5 で抽出する補助関数を作成する．0 から 1 の区間を比率で以下のように分割する．

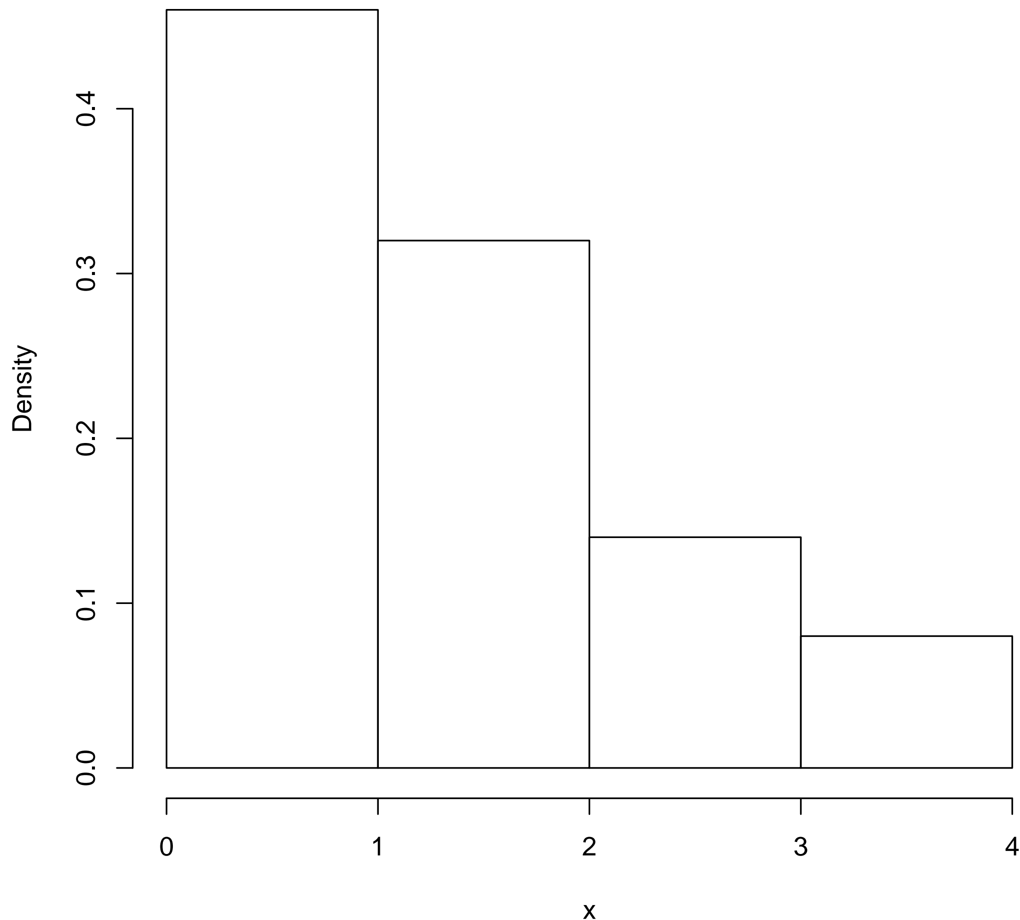
- スタック 1: $0 <= x < 0.5$
- スタック 2: $0.5 <= x < 0.8$
- スタック 3: $0.8 <= x < 0.95$
- スタック 4: $0.95 <= x <= 1$

一様分布に従う乱数は `runif()` 関数（random uniform）で生成できる．以下の `draw.stack()` 補助巻数は，0 から 1 までの乱数を生成し，上で分割した区間に応じてスタックを選択する．

```
draw.stack <- function() {
    x <- runif(n=1, min=0, max=1)  # 一様分布から乱数を生成
    if (x < 0.5)  return(1)
    if (x < 0.8)  return(2)
    if (x < 0.95) return(3)
    return(4)
}
```

試しに 50 回スタック番号を抽出した結果をヒストグラムに表示してみよう．

```
x <- c()
for (i in 1:50) {
    x <- c(x, draw.stack())
}
hist(x, breaks=c(0, 1, 2, 3, 4), freq=FALSE)
```



概ね分割比率に従っているのがわかる．抽出回数を増やせば分割比率により近づく．

4 出題用スタックの準備

ユーザが指定したカテゴリからカードを抽出し，出題回数の小さい順にカードを並べたデータフレームを返す補助関数を作成する．

```
## スタック番号とカテゴリ ID を取り，出題回数の少ない順に並べた
## データフレームを返す．スタックが空の場合は NULL を返す．
## カテゴリ ID 引数に NULL が指定された場合，全てのカードが対象になる．
create.card.pool <- function(stack.num, category.id=NULL) {
    if (category.id==NULL)
        df <- subset(CARD.DB, スタック==stack.num)
    else
        df <- subset(CARD.DB, スタック==stack.num && カテゴリ==category.id)
    df[order(df$出題回数, decreasing=FALSE),]
}
```

5 出題カード抽出関数の実装

出題用にカードを次々に抽出する補助関数を実装する．get.card.drawer() 関数は，最初に create.card.pool() を用いて CARD.DB から出題用カードスタックを生成し，stack リストに格納する．stack リストはローカル変数だが，そのすぐ下で定義されている draw.next.card() ローカル関数に参照されているため，クロージャとして get.card.drawer() 関数実行後も生き続ける．

```
## カードを出題回数の少ない順にランダムに抽出する関数を返す.
## category.id が NULL の場合，全てのカテゴリからカードを抽出する.
## [使い方]
##     drawer <- get.card.drawer(cat.id)
##     drawer()
get.card.drawer <- function(category.id) {
    ## 出題用 temporary スタックを作成
    stack <- lapply(1:4, create.card.pool, category.id)
    if (all(sapply(stack, is.null))) {
        cat("\n------------------------------------------")
        cat("\n 指定されたカテゴリにはカードがありません．")
        cat("\n------------------------------------------\n")
        return(NULL)
    }
    draw.next.card <- function() {
        repeat {
            s <- draw.stack()
            df <- stack[[s]]
            if (!is.null(df))
                break
        }
        obj.name <- paste("stack[", s, "]", sep="")
        ## 空なら temporary スタックを作り直す
        if (nrow(df) == 0) {
            df <- create.card.pool(s, category.id)
            assign(obj.name, df, envir=parent.env(environment()))
        }
        candidates <- subset(df, 出題回数==df[1,"出題回数"])
        i <- sample(1:nrow(candidates), size=1)
        ## 出題するカードを temporary スタックから削除
        assign("stack", `[[<-`(get('stack'), s, value=df[-i,]),
               envir=parent.env(environment()))
        #----- For Debugging ----
        # cat("\n----\n")
        # print(stack[[s]])
        # cat("----\n")
        return(df[i,])
    }
    return(draw.next.card) # 関数を返す
}
```

上記コード内には assign() を使った箇所が 2 箇所ある．1 つ目は以下のようになっている．

```
assign(obj.name, df, envir=parent.env(environment()))
```

s が 1 の時，obj.name は文字列で"stack[[1]]"となる．つまり親 environment にある stack[[1]] に df を代入している．著者の環境ではうまく代入されたが，R ヘルプによればリストの要素は assign() で代入できないと書いてある．うまくいかない場合は，下で示すやり方に書き換えると良い．

2 つ目の assign() 使用箇所でも stack リストの要素を変更している．なぜかこの部分では上のやり方で stack の要素が更新されなかったため，以下のやり方で代入している．

```
assign("stack", `[[<-`(get('stack'), s, value=df[-i,]),
       envir=parent.env(environment()))
```

例えば s が 1 ならば，第 2 引数の部分は stack[[1]] <- df[-i,] と同じで，更新後の stack オブジェクトが，親 environment の stack に代入される．

`[[<-` はリスト要素への代入を行う関数である（` はバッククォート）．R ではあらゆる演算子が関数として定義されている．例えば，+演算子は通常，中置記法で 4 + 2 と書かれるが，実は他の関数と同じように前に書いてカッコの中に引数を与えることもできる．

```
4 + 2

## [1] 6

`+`(4, 2)

## [1] 6
```

get() は引数で与えられた名前のオブジェクトを検索して返す．この場合は，draw.next.card() 関数の外側（親 environment）にある stack ローカル変数が返る．

get.card.drawer() 関数の動作を確認したければ，

```
drawer <- get.card.drawer(NULL)
drawer()
drawer()
```

と実行すれば良い．さらに関数定義内の「For Debugging」直下の 3 行をコメントアウトすると，カードを抽出する度に stack の内容も表示される．

今回は stack 変数を更新した後，draw.next.card() 関数内でもう一度参照することがないので，上の assgin() は 2 行に分けて

```
stack[[s]] <- df[-i,]
assign("stack", stack, envir=parent.env(environment()))
```

と書いても良い．つまり Copy-on-modify を使ってローカルスコープに stack のコピーを作成し，それをクロージャの environment にある stack に代入する．しかし，これを実行するとクロージャ内の stack をローカルスコープの stack がシャドウイングしてしまうので，もし，何らかの理由でシャドウイングしたくない場合には，`[[<-` を用いた方法を使う必要がある．こちらの方法を覚えておくと何かの役に立つかもしれない．

6　カードの表示

次にフラッシュカードを表示する機能を実装する．以下の flash() 関数は第一引数にカードのデータフレームの 1 行を受け取り，第二引数が TRUE なら表から，FALSE なら裏から表示する．カードをめくる動作は「リターン」キーによって行う．

```
## データフレームの 1 行を受け取り，is.front.first が T なら
## 表から，FALSE なら裏から表示する．
flash <- function (card, is.front.first) {
    disp <- ifelse(is.front.first,
                   paste("表:", card$表),
                  paste("裏:", card$裏))
```

```
    cat("-------------------\n")
    cat(disp, "\n")
    readline("-------------------")
    disp <- ifelse(is.front.first,
                   paste("裏:", card$裏),
                   paste("表:", card$表))
    cat(disp, "\n")
    cat("-------------------\n")
}
```

flash() 関数には正誤判定は付けない．これは，練習モードでカードをフラッシュ表示する場合にもこの関数を再利用したいからだ．正誤判定は flash() 関数を呼び出す側で行う．

7　カテゴリ選択補助関数の修正

学習するフラッシュカードをカテゴリに限定する場合，リリース 4 で作成した category.choice.menu() 関数が使える．選択肢の中に，【カテゴリん新規追加】と【親カテゴリなし】を加える機能があったが，今回は【全てのカテゴリ】を選択肢に加えたい．そこで，既存のコードに影響を与えないように以下のように変更する．

変更箇所は引数に add.choice を加えた点である．add.choice に文字列を設定すると，選択肢のトップにその文字列が表示されるように変更した．これで自由に新たな選択肢を加えることができる．

```
## 選択したカテゴリ ID を返す．-1 はキャンセルの選択を表す．
## show.no.parent=TRUE なら 0 は「親カテゴリなし」
## show.add=TRUE なら 0 は「カテゴリの新規追加」を表す．
## add.choice に文字列が指定されたら，選択肢のトップにそれを表示する．
## この場合，show.no.parent 引数は無視される．
category.choice.menu <- function(category.list, title=NULL, show.no.parent=TRUE,
                                 show.add=FALSE, add.choice=NULL) {
    if (is.null(title))
        title="\n カテゴリを番号で選択してください（0 でキャンセル）．"
    if (is.null(add.choice)) {
        if(show.add)
            add.choice <- "【カテゴリの新規追加】"
        else if (show.no.parent)
            add.choice <- "【親カテゴリなし】"
    }

    if (is.null(add.choice))
        choice <- menu(category.list$names, title)
    else
        choice <- menu(c(add.choice, category.list$names),
                       title=title) - 1

    if (choice > 0)
        return(category.list$data[choice, "ID"])
    else if (is.null(add.choice))
        return (-1)
    else
        return(choice)    # 0 or -1
}
```

8　正誤処理用補助関数の実装

フラッシュ・カードの正誤判定結果をデータベースに反映させる補助関数を実装する．ここでは，以下のフィールドを更新する．

- 出題回数
- 正解数 …… 今回はじめて登場．ここで追加する．
- スタック …… 正解なら次のスタックへ，不正解ならスタック 1 へ移動．
- 最終出題日

リリース 1 のカード・データベース設計時に「正解数」を記録する列を確保しておかなかったので，`add.card()` 関数を以下のように修正する．

```
add.card <- function(card.db, front, back, category.id) {
    timestamp <- as.numeric(Sys.time())
    id <- generateID(timestamp)
    list(db=
         rbind(card.db,
               list(ID=id, 表=front, 裏=back, カテゴリ=category.id,
                    作成日=timestamp, 更新日=0, 出題回数=0, 正解数=0, # ここを追加
                    最終出題日=0, スタック=1),
               stringsAsFactors = FALSE),
         id=id)
}
```

すでにカードを幾つか作成済みの場合，以下の手順で既存データベースを更新する．まずアプリを起動し，トップレベルに CARD.DB を作成する．それから以下を実行する．

```
## 「正解数」列を 0 で初期化して追加
CARD.DB <- cbind(CARD.DB, 正解数=0)
## add.card() に合わせて列順を入れ替える.
CARD.DB <- CARD.DB[, c(1,2,3,4,5,6,7,10,8,9)]
## 確認
CARD.DB
## 保存
SAVE.CARD()
```

これで，データベースが更新された．次にリリース 5 で実装したスタック移動補助関数 `move.to.next.stack()` を `mark.card()` に改名し，以下のように変更する．これでスタックの移動だけでなく，正解数，出題回数，最終出題日が正しく記録される．

```
MAX.STACK <- 4
## 与えられた card.id のカードの正・誤とスタックを更新し,
## 変更後のカードデータベースを返す.
mark.card <- function (card.db, card.id, is.correct) {
    index <- card.db$ID == card.id
    card <- card.db[index,]
    if (is.correct) {
        card$正解数 <- card$正解数 + 1
        stack <- card$スタック
        if ( stack < MAX.STACK)
            card$スタック <- stack + 1
    } else {
        card$スタック <- 1
    }
    card$出題回数 <- card$出題回数 + 1
```

```
    card$最終出題日 <- as.numeric(Sys.time())
    card.db[index, ] <- card
    card.db
}
```

9　フラッシュカード機能の実装

いよいよ最終段階に来た．ここではフラッシュカード機能，すなわち，テスト機能を実装する．テスト機能といってもフラッシュカードなので，自分でカードをめくった後，自分の考えていた答えが正しければ「正解」を，間違っていれば「不正解」をインターフェースに入力する単純なものだ．正解なら，正解数を更新しスタックを移動する．

実装はこれまで作成してきた補助関数を組み合わせて行う．

```
## フラッシュカード（メインメニュー項目）
flash.menu <- list(
    menu.title="学習",
    fn=function() {
        ## 正解・不正解を処理するローカル関数．終了なら-1を返す．
        judge.prompt <- function (card.id) {
            repeat {
                choice <- readline("正解 [リターンキー]/不正解 [n]/終了 [q] > ")
                if (str_length(choice)==0 || choice=="n" || choice=="q")
                    break
            }
            if (str_length(choice)==0) { # 正解
                SAVE.CARD(mark.card(CARD.DB, card.id, is.correct=TRUE))
                return(1)
            } else if (choice == "n") { # 不正解
                SAVE.CARD(mark.card(CARD.DB, card.id, is.correct=FALSE))
                return(1)
            } else
                return(-1)
        }
        ## カテゴリの選択
        cat.id <- category.choice.menu(make.category.list(CAT.DB),
                                       add.choice="【全カテゴリ】")
        if (cat.id < 0)
            return ()
        else if (cat.id == 0)
            cat.id <- NULL
        ## 表・裏の選択
        is.front.first <- TRUE
        ## 学習開始
        drawer <- get.card.drawer(cat.id)
        if (is.null(drawer))
            return ()
        repeat {
            card <- drawer()
            flash(card, is.front.first)
            if (judge.prompt(card$ID) < 0)
                break # 学習終了
        }
    })
```

あとは `flash.menu` をアプリに追加すれば良い．

10　スタックの表示

スタックの構成比をヒストグラムで表示する機能をメニューに追加する．

```
show.stack.menu <- list(
    menu.title="スタックを表示",
    fn=function() {
        hist(CARD.DB$スタック, breaks=c(0, 1, 2, 3, 4), freq=FALSE)
    })
```

11　メインメニューへの追加

上で作成した flash.menu と show.stack.men を追加して，完成．

```
# アプリ起動関数
run.app <- make.menu.func(list(flash.menu,
                               add.card.menu, category.menus,
                               edit.card.menu, show.category.menu,
                               show.stack.menu),
                           title="\n メインメニュー （0 で終了）")
```

12　ここから先は？

- 自分でメニュー画面や一覧表示項目などを改良しよう．例えば，正答率の列を加えて，カードの一覧表示で正答率も合わせて表示するように改善することができる．
- 「学習」を選択すると正誤が記録されるが，正誤が記録されない練習モードを追加してみよう．flash() 関数の後，judge.prompt() に替わる関数を組み合わせれば良い．
- これまでの成績統計を表示する機能を追加してみよう．
- R の解説本を幾つか読んで R 言語の理解を深めよう．
- RStudio と Shiny を使うと GUI を作成できる．Web や書籍などを探して，フラッシュカードアプリに GUI を追加してみよう．
- プログラミング言語を 1 つ習得すると，2 つ目のプログラミング言語の習得は速くなる．なぜなら基本的な概念は同じだからだ[1]．Swift を学び，今回のプログラムを Swift に置き換えれば iPhone や Mac で動くフラッシュカードアプリを作れるし，Java を学べば Android 携帯で動くアプリを作れる．今後もプログラミングの勉強を続けよう．

[1] 変数，関数，データ構造，ループ，条件分岐，関数型プログラミング，クロージャ，リファクタリング，等々．ただし，この授業ではオブジェクト指向型プログラミングについては解説していない．