第9回 大規模データを用いたデータフレーム操作実習(3)

2023年2月27日

目 次

1 ここで学ぶ事 **1**

2 データの準備 **1**

3 apply 系関数を用いた繰り返し処理 **1**

4 データ属性によるデータの集計 **5**

4.1 tapply() 関数 5

4.2 練習問題 6

4.3 by() 関数 6

4.4 練習問題 10

4.5 練習問題 10

5 apply 系関数への追加引数の渡し方 **10**

6 mapply 関数による複数引数を取る関数の適用 **11**

6.1 練習問題 13

7 apply 系関数による 2 重ループの実現方法 **13**

7.1 練習問題 15

1 ここで学ぶ事

- データの属性に応じた集計処理を学ぶ.
- apply 系関数を用いた繰り返し処理を学ぶ.

2 データの準備

データは第 8 回で修正保存した独立行政法人統計センターの大規模疑似ミクロデータを使用する.

```
> load("Microdata2.RData")  # E ドライブないなら"E:/Microdata2.RData"
> ls()                      # microdata オブジェクトが読み込まれた
[1] "microdata"
> dim(microdata)
[1] 45811    20
```

microdata は高々4 万 6 千件程度でビッグデータの足元にも及ばないが，それでも Excel などの表計算ソフトを使ってこの規模のデータを集計したり特徴を抽出するのは難しい．規模が大きくなればなるほど，繰り返し処理が得意なプログラミング言語の出番だ．繰り返し処理は第 6 回のプリントで学んだが，ここでは，関数型プログラミングと呼ばれるプログラミング手法を用いた繰り返し処理について学ぶ．

3　apply 系関数を用いた繰り返し処理

例として，microdata の列の中で数値データが格納されている全ての列に対し，各列で平均を求めたいとしよう．str() 関数で構造を見みると「年間収入」列以降が全て数値データのベクトルになっている．ここでは，これらの列の平均値を求めていく．

以前学んだループ処理（for や while）を使えば，データフレームから列を取り出し平均を求めることができる．例えば,「年間収入」列から最終列までのインデックスを用いて，以下のように各列の平均値を導出できる．

```
> num.index <- which(names(microdata) == "年間収入"):ncol(microdata)
> for(i in num.index)
+     mean(microdata[[i]]) # この計算結果は表示されない！
```

まず「年間収入」列のインデックスを which() 関数を用いて取得し，ncol() 関数で microdata の列数を取得している．ncol(microdata) の代わりにデータフレームの dimension 属性を使い dim(microdata)[2] で列数を取得しても良い．これで num.index には「年間収入」列から最終列までのインデックス番号のベクトルが入っている．

for ループではインデックス番号をカウンタ変数 i に代入し，microdata の列にアクセスすることで平均値を計算している．しかし，上のプログラムを実行しても結果は画面に表示されない．なぜなら for ループは関数のように値を返さないからだ．計算結果をどこかに代入して保存しておかなければ処理結果が失われる．以下の例では計算結果を保持するために一時変数 x を使っている．

```
> num.index <- which(names(microdata) == "年間収入"):ncol(microdata)
> x <- rep(NA, length(num.index))
> for(i in num.index) {
+     x[i-num.index[1]+1] <- mean(microdata[[i]])
+ }
> x  # 結果の表示
 [1]   6401.924 298373.681  68740.208  16127.913  19420.996   9373.958  12054.821
 [8]  13280.967  44692.369  15014.886  31099.288  68568.275
```

各平均値がどの列の計算結果かを保持しておきたい時は names 属性を設定すると良い.

```
> names(x) <- colnames(microdata)[num.index]
> x
       年間収入         消費支出           食料         住居       光熱・水道
      6401.924       298373.681      68740.208     16127.913      19420.996
 家具・家事用品     被服及び履物       保健医療   交通・通信         教育
      9373.958       12054.821      13280.967     44692.369      15014.886
       教養娯楽 その他の消費支出
     31099.288       68568.275
```

一般に for や while などのループ処理では，インデックスを表すカウンタ変数（上の例では i）や処理結果を保持する一時変数（上の例では x）が必要になる．R では変数宣言は必ずしも必要ないが，上の 2 行目では計算結果を保持する x を NA で初期化し保存場所を用意している．

各列の計算結果は x の第 1 要素から順に記録したい，列番号を保持する num.index には 9 から 20 までの整数が入っているため，x[i-num.index[1]+1] でインデックス番号を調整している．このようにカウンタ変数や一時変数を用いると，その値だけでなくどのように変化していくかについても考える必要が生じ面倒だ.

実はカウンタ変数だけなら使わずにうまく回避できる．処理結果を格納するベクトルを作成する際に，添字でアクセスするのではなく，作成済みのベクトルと新しい要素を c() で接続すれば良い.

```
> x <- c()  # 空のベクトルを用意
> for(col in microdata[num.index])
+     x <- c(x, mean(col))
> names(x) <- colnames(microdata)[num.index]
> x
       年間収入         消費支出           食料         住居       光熱・水道
      6401.924       298373.681      68740.208     16127.913      19420.996
 家具・家事用品     被服及び履物       保健医療   交通・通信         教育
      9373.958       12054.821      13280.967     44692.369      15014.886
       教養娯楽 その他の消費支出
     31099.288       68568.275
```

for in はベクトルの要素だけでなくリストの要素も順に走査できる．データフレームはリストでもある．上の例では 1 重括弧 [] を使うことでリスト要素の範囲指定を行い，データフレームの行を抽出している.

カウンタを使わない上の例では，プログラムもすっきりとして見やすいが，結果を保持する一時変数の使用は相変わらず避けられない．関数型プログラミングと呼ばれるプログラミング・スタイルでは，変数は定数を格納するためだけに用い，その値を変化させないようにプログラムを組む．変数の値を変化させないため一時変数も極力用いない．そうすることで，プログラムのバグを減らせる上，複数の CPU を使った並列計算などでも問題なくプログラムを動かすことができる[1]．そうした上級の話しについてはまた別の機会に譲るとして，ここでは関数型プログラミングでどのようにループ処理を実現するか説明しよう.

[1] 並行して走る複数のプロセスから一時変数がアクセスされ，一方のプロセスがその値を書き換えてしまうとエラーの原因になる.

関数型プログラミングでは，繰り返し処理の「手順」を記述するのではなく，単一処理を関数で表し，その関数を複数要素に適用（apply）することで繰り返し処理を実現する．R にはこのような処理を行う apply 系の関数が色々と用意されている[2]．

最も頻繁に使われるのが，リストやベクトルの要素に対して関数を適用する `lapply()` 関数（list apply）である．`lapply()` は第 1 引数にリストやベクトルを渡し，第 2 引数に各要素に適用する関数を渡す．データフレームはリストでもあるので，`lapply()` 関数にデータフレームを渡すと各列に対して関数を適用することができる．

```
> lapply(microdata[, num.index], mean)
$年間収入
[1] 6401.924

$消費支出
[1] 298373.7

$食料
[1] 68740.21

$住居
[1] 16127.91

$`光熱・水道`
[1] 19421

$`家具・家事用品`
[1] 9373.958

$被服及び履物
[1] 12054.82

$保健医療
[1] 13280.97

$`交通・通信`
[1] 44692.37

$教育
[1] 15014.89

$教養娯楽
[1] 31099.29

$その他の消費支出
[1] 68568.28
```

先ほどの `for` ループの例と比べて圧倒的に簡潔に書ける．一時変数もカウンタ変数も使わないので，カウンタの値について色々と頭を悩ませる必要もない．`lapply()` は結果をリストにして返

[2]他のプログラミング言語を知っている人は map 系の関数と同じである．

すが，これをデータフレームに変換すれば見やすい表形式にできる．

```
> x <- lapply(microdata[, num.index], mean)
> as.data.frame(x) # データフレームに変換して表示
  年間収入 消費支出      食料      住居 光熱. 水道 家具. 家事用品 被服及び履物 保健医療
1 6401.924 298373.7 68740.21 16127.91      19421       9373.958      12054.82 13280.97
  交通. 通信      教育 教養娯楽 その他の消費支出
1   44692.37 15014.89 31099.29          68568.28
```

また，sapply() 関数を使うと，結果をシンプルなデータ型（多次元なら配列型，1 次元ならベクトル型）に変換して結果を返す．

```
> sapply(microdata[, num.index], mean)
        年間収入          消費支出            食料            住居        光熱・水道
        6401.924       298373.681        68740.208        16127.913        19420.996
  家具・家事用品      被服及び履物        保健医療      交通・通信            教育
        9373.958        12054.821        13280.967        44692.369        15014.886
        教養娯楽  その他の消費支出
       31099.288        68568.275
```

4　データ属性によるデータの集計

前節では列ごとにデータを集計する方法を学んだ．本節ではデータ属性ごとにデータを集計する方法を学ぶ．

いま microdata の「年齢階級２」の因子グループ毎に，年間収入の平均を求めたいとしよう．データの属性毎に分析を行ったり集計をしたいことは良くある．このような集計処理も for や while ループを用いて記述できるが，それはプログラミング・エクササイズとして練習問題に譲り，ここでは apply 系関数である tapply() と by() を用いた集計処理を解説する．

4.1　tapply() 関数

tapply() 関数は第 1 引数に与えられたベクトルを第 2 引数（INDEX）に与えられた factor でグループ分けし，グループ毎に第 3 引数（FUN）の関数を適用する．年齢階級 2 のグループ毎に年間収入の平均を求める処理は以下のようになる．

```
> tapply(microdata$年間収入, INDEX=microdata$年齢階級２, FUN=mean)
（就業者）30 歳未満 （就業者）30〜34 歳 （就業者）35〜39 歳 （就業者）40〜44 歳
          4331.278           5416.315           6067.400           6969.943
（就業者）45〜49 歳 （就業者）50〜54 歳 （就業者）55〜59 歳 （就業者）60〜64 歳
          7862.956           8485.286           8367.246           6753.724
（就業者）65 歳以上
          6946.241
```

上の結果はベクトルで返っているが，これは tapply() 関数の simplify 引数のデフォルト値が TRUE だからである．もし，simplify=FALSE に設定すればリストが返る．

もし，factor にグループ分けした集計が目的ではなく，単に factor に基づいてベクトルデータをグループ分けしたいだけならば split() 関数が使える．

```
> sp <- split(microdata$年間収入, microdata$年齢階級２)
> str(sp)
List of 9
 $ （就業者）30 歳未満: num [1:1068] 3917 6675 2790 3452 9252 ...
 $ （就業者）30〜34 歳: num [1:2602] 6760 4026 2036 3921 5930 ...
 $ （就業者）35〜39 歳: num [1:4008] 6044 6841 4442 8561 10267 ...
 $ （就業者）40〜44 歳: num [1:4209] 6661 8159 5530 4870 4699 ...
 $ （就業者）45〜49 歳: num [1:4168] 4142 6750 10319 13474 3270 ...
 $ （就業者）50〜54 歳: num [1:4523] 4454 5302 3260 6035 5952 ...
 $ （就業者）55〜59 歳: num [1:5156] 8971 5841 7351 3084 14701 ...
 $ （就業者）60〜64 歳: num [1:4285] 7341 4401 4150 7498 3538 ...
 $ （就業者）65 歳以上: num [1:4326] 9847 3385 5686 10539 1731 ...
```

つまり tapply() 関数は split() 関数を適用した結果に sapply() を実行したのと同じである．

```
> sapply(split(microdata$年間収入, microdata$年齢階級２), mean)
（就業者）30 歳未満 （就業者）30〜34 歳 （就業者）35〜39 歳 （就業者）40〜44 歳
          4331.278           5416.315           6067.400           6969.943
（就業者）45〜49 歳 （就業者）50〜54 歳 （就業者）55〜59 歳 （就業者）60〜64 歳
          7862.956           8485.286           8367.246           6753.724
（就業者）65 歳以上
          6946.241
```

ちなみに，tapply() 関数に自分自身を返す関数 function(x) x を与えてデータに適用すれば，factor で分割されたデータそのものが返るので split() と同じ処理を実現できる．

```
> tsp <- tapply(microdata$年間収入, INDEX=microdata$年齢階級２,
+               FUN=function(x) x)
> str(tsp)
List of 9
 $ （就業者）30 歳未満: num [1:1068] 3917 6675 2790 3452 9252 ...
 $ （就業者）30〜34 歳: num [1:2602] 6760 4026 2036 3921 5930 ...
 $ （就業者）35〜39 歳: num [1:4008] 6044 6841 4442 8561 10267 ...
 $ （就業者）40〜44 歳: num [1:4209] 6661 8159 5530 4870 4699 ...
 $ （就業者）45〜49 歳: num [1:4168] 4142 6750 10319 13474 3270 ...
 $ （就業者）50〜54 歳: num [1:4523] 4454 5302 3260 6035 5952 ...
 $ （就業者）55〜59 歳: num [1:5156] 8971 5841 7351 3084 14701 ...
 $ （就業者）60〜64 歳: num [1:4285] 7341 4401 4150 7498 3538 ...
 $ （就業者）65 歳以上: num [1:4326] 9847 3385 5686 10539 1731 ...
 - attr(*, "dim")= int 9
 - attr(*, "dimnames")=List of 1
  ..$ : chr [1:9] "（就業者）30 歳未満" "（就業者）30〜34 歳" "（就業者）35〜39 歳" "（就業者）40〜44 歳" ...
```

先ほどの str(sp) と結果を比較してみよ.

4.2 練習問題

tapply(microdata$年間収入, INDEX=microdata$年齢階級２, FUN=mean) と同じ結果を for ループを用いて実現せよ.

4.3 by() 関数

前節では年間収入の列のみで平均を計算した. ここでは, 他の列についても年齢階級 2 のグループ毎に平均を求めたい場合について考える. 当然 for や while ループを使って書けるが, これまでに学んだ apply 系関数を組み合わせても記述できる. 紙面制約上すべての列の計算結果を表示できないので年間収入に続く 3 列のみで集計する. まず, 年間収入列のインデックスを beg に格納し, それに 2 列加えて合計 3 列分のインデックスを num.index に作成する.

```
> beg <- which(names(microdata) == "年間収入")
> num.index <- beg:(beg+2) # beg 列+2 列で 3 列を指定するインデックス
```

lapply() 関数に microdata の 3 列から成るデータフレームを渡すことで, 3 つの各列に関数を適用する.

```
> res1 <- lapply(microdata[num.index],
+                function(x) tapply(x, microdata$年齢階級２, mean))
> res1
$年間収入
 （就業者）30 歳未満 （就業者）30〜34 歳 （就業者）35〜39 歳 （就業者）40〜44 歳
          4331.278           5416.315           6067.400           6969.943
 （就業者）45〜49 歳 （就業者）50〜54 歳 （就業者）55〜59 歳 （就業者）60〜64 歳
          7862.956           8485.286           8367.246           6753.724
 （就業者）65 歳以上
          6946.241

$消費支出
 （就業者）30 歳未満 （就業者）30〜34 歳 （就業者）35〜39 歳 （就業者）40〜44 歳
          233723.8           264278.0           276683.2           302350.7
 （就業者）45〜49 歳 （就業者）50〜54 歳 （就業者）55〜59 歳 （就業者）60〜64 歳
          352396.0           376141.1           342372.2           314716.4
 （就業者）65 歳以上
          280220.5

$食料
 （就業者）30 歳未満 （就業者）30〜34 歳 （就業者）35〜39 歳 （就業者）40〜44 歳
          46295.59           55284.70           63299.24           71718.85
 （就業者）45〜49 歳 （就業者）50〜54 歳 （就業者）55〜59 歳 （就業者）60〜64 歳
          76681.27           77570.34           74009.64           74220.03
```

```
（就業者）65 歳以上
         70053.94
```

第 2 引数の関数には lapply() によってデータフレームの各列が渡されるので，それを x で受け取り，関数内で tapply() によって factor 毎に mean() 関数を適用している．このように apply 系の関数を組み合わせることで複雑な処理をいくらでも記述できる．

上の例は列毎に factor でグループ分けして集計した．同じことだが，形式を変えて，集計結果をグループで分割し，各グループ毎に年間収入と消費支出，食料の列で平均を出したいとしよう．この場合はまず，factor で分割したデータフレームを用意し，各データフレームの列に対し mean() を実行する必要がある．これを apply 系関数を組み合わせて実現するのはエクササイズとして練習問題にとっておき，ここでは R に用意されている by() 関数を用いて簡単に実現する方法を提示する．

tapply() はベクトルを引数に取り factor でデータをグループ分けして関数を適用するのに対し，by() 関数はデータフレームを引数にとり，factor で分割したデータフレームに対して関数を適用する．by() 関数で先にグループ分けしたデータフレームの各列に対し，sapply() で mean() 関数を適用すれば，先ほどの res1 と反転したデータ構造の結果を得られる．

```
> res2 <- by(microdata[num.index], microdata$年齢階級２,
+            function(x) sapply(x, mean))
> res2
microdata$年齢階級２: （就業者）30 歳未満
  年間収入    消費支出        食料
  4331.278 233723.810  46295.593
------------------------------------------------------------
microdata$年齢階級２: （就業者）30〜34 歳
  年間収入    消費支出        食料
  5416.315 264277.989  55284.698
------------------------------------------------------------
microdata$年齢階級２: （就業者）35〜39 歳
 年間収入   消費支出       食料
  6067.40 276683.24  63299.24
------------------------------------------------------------
microdata$年齢階級２: （就業者）40〜44 歳
  年間収入    消費支出        食料
  6969.943 302350.720  71718.855
------------------------------------------------------------
microdata$年齢階級２: （就業者）45〜49 歳
  年間収入    消費支出        食料
  7862.956 352396.050  76681.275
------------------------------------------------------------
microdata$年齢階級２: （就業者）50〜54 歳
  年間収入    消費支出        食料
  8485.286 376141.101  77570.343
------------------------------------------------------------
microdata$年齢階級２: （就業者）55〜59 歳
  年間収入    消費支出        食料
  8367.246 342372.214  74009.642
------------------------------------------------------------
```

```
microdata$年齢階級２：（就業者）60〜64 歳
  年間収入    消費支出        食料
  6753.724 314716.450  74220.033
------------------------------------------------------------
microdata$年齢階級２：（就業者）65 歳以上
  年間収入    消費支出        食料
  6946.241 280220.532  70053.941
```

res1 と res2 の微妙な違いをよく観察してみよう．各計算結果は同じだが，結果のプレゼンテーションが異なる．ちなみに，by() 関数は結果を整形して画面に印字するが，戻り値はリストである．

```
> str(res2)
List of 9
 $ （就業者）30 歳未満: Named num [1:3] 4331 233724 46296
  ..- attr(*, "names")= chr [1:3] "年間収入" "消費支出" "食料"
 $ （就業者）30〜34 歳: Named num [1:3] 5416 264278 55285
  ..- attr(*, "names")= chr [1:3] "年間収入" "消費支出" "食料"
 $ （就業者）35〜39 歳: Named num [1:3] 6067 276683 63299
  ..- attr(*, "names")= chr [1:3] "年間収入" "消費支出" "食料"
 $ （就業者）40〜44 歳: Named num [1:3] 6970 302351 71719
  ..- attr(*, "names")= chr [1:3] "年間収入" "消費支出" "食料"
 $ （就業者）45〜49 歳: Named num [1:3] 7863 352396 76681
  ..- attr(*, "names")= chr [1:3] "年間収入" "消費支出" "食料"
 $ （就業者）50〜54 歳: Named num [1:3] 8485 376141 77570
  ..- attr(*, "names")= chr [1:3] "年間収入" "消費支出" "食料"
 $ （就業者）55〜59 歳: Named num [1:3] 8367 342372 74010
  ..- attr(*, "names")= chr [1:3] "年間収入" "消費支出" "食料"
 $ （就業者）60〜64 歳: Named num [1:3] 6754 314716 74220
  ..- attr(*, "names")= chr [1:3] "年間収入" "消費支出" "食料"
 $ （就業者）65 歳以上: Named num [1:3] 6946 280221 70054
  ..- attr(*, "names")= chr [1:3] "年間収入" "消費支出" "食料"
 - attr(*, "dim")= int 9
 - attr(*, "dimnames")=List of 1
  ..$ microdata$年齢階級２: chr [1:9] "（就業者）30 歳未満" "（就業者）30〜34 歳" "（就業者）35〜39 歳" "（就業者
 - attr(*, "call")= language by.data.frame(data = microdata[num.index], INDICES = microdata$年齢階級２,
 - attr(*, "class")= chr "by"
```

by() 関数で適用させたい関数に自分自身を返す関数 function(x) x を与えれば，factor で分割されたデータフレームを得ることができる．

```
> res3 <- by(microdata[num.index], microdata$年齢階級２,
+            function(x) x)
> str(res3)
List of 9
 $ （就業者）30 歳未満:'data.frame':        1068 obs. of  3 variables:
  ..$ 年間収入: num [1:1068] 3917 6675 2790 3452 9252 ...
  ..$ 消費支出: num [1:1068] 201649 166381 114511 152109 192439 ...
  ..$ 食料    : num [1:1068] 47756 34054 41664 34924 68882 ...
```

```
 $ （就業者）30〜34 歳:'data.frame':        2602 obs. of  3 variables:
  ..$ 年間収入: num [1:2602] 6760 4026 2036 3921 5930 ...
  ..$ 消費支出: num [1:2602] 176625 118396 202926 259507 219547 ...
  ..$ 食料    : num [1:2602] 43112 50864 31411 45714 29372 ...
 $ （就業者）35〜39 歳:'data.frame':        4008 obs. of  3 variables:
  ..$ 年間収入: num [1:4008] 6044 6841 4442 8561 10267 ...
  ..$ 消費支出: num [1:4008] 162895 173330 109483 335349 165003 ...
  ..$ 食料    : num [1:4008] 61363 77210 50865 61149 44174 ...
 $ （就業者）40〜44 歳:'data.frame':        4209 obs. of  3 variables:
  ..$ 年間収入: num [1:4209] 6661 8159 5530 4870 4699 ...
  ..$ 消費支出: num [1:4209] 162306 238773 132441 132558 113552 ...
  ..$ 食料    : num [1:4209] 41530 70028 33264 44799 51370 ...
 $ （就業者）45〜49 歳:'data.frame':        4168 obs. of  3 variables:
  ..$ 年間収入: num [1:4168] 4142 6750 10319 13474 3270 ...
  ..$ 消費支出: num [1:4168] 111686 219142 384826 325165 261185 ...
  ..$ 食料    : num [1:4168] 28461 42365 77786 62846 45761 ...
 $ （就業者）50〜54 歳:'data.frame':        4523 obs. of  3 variables:
  ..$ 年間収入: num [1:4523] 4454 5302 3260 6035 5952 ...
  ..$ 消費支出: num [1:4523] 335898 129143 98772 221929 278508 ...
  ..$ 食料    : num [1:4523] 46066 37961 22435 60268 61786 ...
 $ （就業者）55〜59 歳:'data.frame':        5156 obs. of  3 variables:
  ..$ 年間収入: num [1:5156] 8971 5841 7351 3084 14701 ...
  ..$ 消費支出: num [1:5156] 180107 145941 540513 167655 364450 ...
  ..$ 食料    : num [1:5156] 54549 41392 59832 58368 109103 ...
 $ （就業者）60〜64 歳:'data.frame':        4285 obs. of  3 variables:
  ..$ 年間収入: num [1:4285] 7341 4401 4150 7498 3538 ...
  ..$ 消費支出: num [1:4285] 269788 325149 194339 159264 305398 ...
  ..$ 食料    : num [1:4285] 75126 42586 66839 48959 78388 ...
 $ （就業者）65 歳以上:'data.frame':        4326 obs. of  3 variables:
  ..$ 年間収入: num [1:4326] 9847 3385 5686 10539 1731 ...
  ..$ 消費支出: num [1:4326] 533188 432327 210371 246561 158283 ...
  ..$ 食料    : num [1:4326] 50870 79995 52180 75219 39533 ...
 - attr(*, "dim")= int 9
 - attr(*, "dimnames")=List of 1
  ..$ microdata$年齢階級２: chr [1:9] "（就業者）30 歳未満" "（就業者）30〜34 歳" "（就業者）35〜39 歳" "（就業者
 - attr(*, "call")= language by.data.frame(data = microdata[num.index], INDICES = microdata$年齢階級２,
 - attr(*, "class")= chr "by"
```

4.4 練習問題

ここではプログラミング練習のために apply 系関数も by() も用いずに因子グループ毎の集計処理を行ってみる．for ループか while ループのみを用いて，以下の手順に従い「年齢階級２」の因子グループ毎に平均年間収入を求めよ．

(1) 年齢階級２の水準毎に microdata をデータフレームに分割し，結果を順にリストに格納せよ．ただし各リスト要素には年齢階級２の水準名を names 属性に設定すること．

(2) 上で作成した年齢階級毎のデータフレームから，年間収入の平均を求め，結果を名前付きのベクトルに格納せよ．ベクトル要素の名前は年齢階級 2 の水準名になっていること．

4.5 練習問題

by(microdata[num.index], microdata$年齢階級２, function(x) sapply(x, mean)) と同じ結果を apply 系関数を用いて実現せよ．

5 apply 系関数への追加引数の渡し方

lapply や sapply 等の apply 系関数で適用させたい関数に追加引数を指定したい場合がある．我々の例では microdata の数値データ列に NA が含まれていなかったが，もし含まれていたら mean() 関数は NA を返す．その場合，mean() 関数に na.rm=TRUE を指定する必要がある．こうした追加引数を設定するには，apply 系関数の最後に引数をおく．

```
> data <- list(a=c(1,2,3, NA, 4, 5), b=1:100, c=c(10:20, NA, 22:100))
> sapply(data, mean)
   a    b    c
  NA 50.5   NA
> sapply(data, mean, na.rm=TRUE) # 最後の na.rm 引数は mean 関数にそのまま渡される
       a        b        c
 3.00000 50.50000 55.37778
```

6 mapply 関数による複数引数を取る関数の適用

lapply() や sapply() 関数は，1 つのリストを引数に取りその要素に対してのみ繰り返し処理を行う．本節では，複数のリストの各要素を取り出して処理したい場合の方法を学ぶ．

例として，アメリカの人口データから，人口成長率を求める問題を考えよう．R の組み込みデータセットには 1790–1970 年のアメリカの人口データがある．追加ライブラリのインストールなしで，uspop でアクセスできる．

```
> str(uspop)
 Time-Series [1:19] from 1790 to 1970: 3.93 5.31 7.24 9.64 12.9 17.1 23.2 31.4 39.8 50.2 ...
```

Time-Series と冒頭にあるのは，R の組み込みのデータ型ではなく，それらを使って新たに定義された時系列データ型であることを示している．しかし，その右側の [1:19] から内部的には 19 個のベクトルデータと変わらないことが分かる．

```
> uspop
Time Series:
Start = 1790
End = 1970
```

```
Frequency = 0.1
 [1]   3.93   5.31   7.24   9.64  12.90  17.10  23.20  31.40  39.80  50.20  62.90
[12]  76.00  92.00 105.70 122.80 131.70 151.30 179.30 203.20
```

このデータから各年の人口成長率を求めてみよう．第 t 期の人口成長率 $x_t = (x_t - x_{t-1})/x_{t-1}$ を for ループで記述すれば以下のようになる．

```
> x <- rep(NA, length(uspop)-1)
> for(i in 2:length(uspop))
+     x[i-1] <- (uspop[i] - uspop[i-1])/uspop[i-1]
> x # 結果を表示
 [1] 0.35114504 0.36346516 0.33149171 0.33817427 0.32558140 0.35672515 0.35344828
 [8] 0.26751592 0.26130653 0.25298805 0.20826709 0.21052632 0.14891304 0.16177862
[15] 0.07247557 0.14882308 0.18506279 0.13329615
```

1 行目で人口成長率の計算結果を保持するベクトルを NA で初期化し，for ループ内の計算結果を x に格納している．人口成長率の計算では 2 期分のデータを使うため，x のインデックスと uspop のインデックス番号が 1 つずれる点に注意しなければならない．

これを apply 系関数を用いて関数型プログラミングのスタイルで書き換えてみよう．まず，単一処理を担う関数を用意し，apply 系関数で各要素にその関数を適用させれば良いことは既に学んだ．今回の例では，成長率を求める関数 growth を以下のように定義し，データの各要素に対しこの関数を適用することでループ処理を実現する．

```
> ## 関数の中身が 1 行なので関数本体を囲む中括弧を省略している
> growth <- function(x1, x0) (x1 - x0)/x0
```

しかし，ここで 1 つ問題が生じる．growth() 関数は引数を 2 つとる．lapply() や sapply() は 1 つのリストしか引数に取らなかった．

複数要素に対し関数を適用（apply）適用させるには mapply 関数を用いる．第 1 引数に適用させたい関数をとり，第 2 引数以降に第 1 引数の関数に順に渡すデータを指定する．

growth() は 1 期ずれたデータを必要とするので，usop のベクトルデータを 1 期ずらしたものと，元のデータの 2 つのベクトルを growth() に渡せば良い．

```
> mapply(growth, uspop[-1], uspop[-length(uspop)])
 [1] 0.35114504 0.36346516 0.33149171 0.33817427 0.32558140 0.35672515 0.35344828
 [8] 0.26751592 0.26130653 0.25298805 0.20826709 0.21052632 0.14891304 0.16177862
[15] 0.07247557 0.14882308 0.18506279 0.13329615
```

先ほどの for ループと同じ結果が得られていることを確認しよう．mapply の第 2 引数の uspop[-1] は人口データから第 1 要素を取り除いたもので，uspop[-length(uspop)] は最後のデータを取り除いたものである．つまり，各引数に与えられたベクトルの中身は表 1 のようになっている．

mapply は上の表の 1 行目から順に growth 関数に値を渡し，結果を再びベクトルとして返すことで，for ループと同じ処理を実現している．もう一度 mapply による計算を示す．

index	uspop[-1]	uspop[-length(uspop0)]
1	第 2 期の人口	第 1 期の人口
2	第 3 期の人口	第 2 期の人口
3	第 4 期の人口	第 3 期の人口
⋮	⋮	⋮
18	第 19 期の人口	第 18 期の人口

表 1: mapply に渡された引数の中身

```
> mapply(growth, uspop[-1], uspop[-length(uspop)])
```

apply 系関数を用いて関数型プログラミングのスタイルで書くことにより，ここでも for ループを用いた記述より簡潔に書けることが分かる．また，for ループでは，カウンタの i や i-1 といった 1 つずれた値などに間違いがないか注意してプログラムを組み立てなければならないが，apply 系関数を使った記述ではこういった心配は無用である．

上の例では分かりやすいように単一処理のために growth 関数を定義したが，ここの計算でしか用いないのならば以前学んだ無名関数を使って記述することもできる．

```
> mapply(function(x1, x0) (x1 - x0)/x0,
+        uspop[-1], uspop[-length(uspop)])
 [1] 0.35114504 0.36346516 0.33149171 0.33817427 0.32558140 0.35672515 0.35344828
 [8] 0.26751592 0.26130653 0.25298805 0.20826709 0.21052632 0.14891304 0.16177862
[15] 0.07247557 0.14882308 0.18506279 0.13329615
```

mapply の第 2 引数以降に渡されるデータの要素数が異なる場合には，ベクトル同士の計算の時と同じように，短い方のデータが繰り返され再利用される．例えば，アメリカの人口成長率の計算で，1 年間の成長率ではなく，データ開始時の 1970 年からの成長率を計算したい場合には，x0 に渡されるベクトルデータに 1970 年のデータを 1 つ渡すだけで良い．

```
> mapply(function(x1, x0) (x1 - x0)/x0,
+        uspop[-1], # 第 1 要素のみを除いたベクトルデータ
+        uspop[1])  # 第 1 要素の 1970 年のデータのみ与える
 [1]  0.3511450  0.8422392  1.4529262  2.2824427  3.3511450  4.9033079  6.9898219
 [8]  9.1272265 11.7735369 15.0050891 18.3384224 22.4096692 25.8956743 30.2468193
[15] 32.5114504 37.4987277 44.6234097 50.7048346
```

上の計算は以下の計算と同じである．

```
> mapply(function(x1, x0) (x1 - x0)/x0,
+        uspop[-1], # 第 1 要素のみを除いたベクトルデータ
+        rep(uspop[1], length(uspop)-1))  # 第 1 要素の 1970 年のデータを繰り返す
 [1]  0.3511450  0.8422392  1.4529262  2.2824427  3.3511450  4.9033079  6.9898219
```

```
 [8]  9.1272265 11.7735369 15.0050891 18.3384224 22.4096692 25.8956743 30.2468193
[15] 32.5114504 37.4987277 44.6234097 50.7048346
```

6.1 練習問題

関数型プログラミングの手法に基づき，大規模ミクロデータから年齢階級毎の平均エンゲル係数（消費支出に占める飲食費の割合（%））を求めよ．ただし，ここでは飲食費の代わり「食料」列の値を採用して計算せよ．結果は以下のようにならなければならない．

```
> result
（就業者）30 歳未満 （就業者）30〜34 歳 （就業者）35〜39 歳 （就業者）40〜44 歳
          21.15751             22.14934             24.53396             25.32359
（就業者）45〜49 歳 （就業者）50〜54 歳 （就業者）55〜59 歳 （就業者）60〜64 歳
          23.84767             23.24053             24.47761             26.39286
（就業者）65 歳以上
          27.54082
```

7 apply 系関数による 2 重ループの実現方法

第 6 回で for ループを入れ子にした 2 重ループによって，ベクトルの全組み合わせに対する以下のループ処理を紹介した．

```
> for(i in 1:5) {
+     for(j in 1:5)
+         cat(i, "-", j, ", ", sep="")
+     cat("\n")
+ }
1-1, 1-2, 1-3, 1-4, 1-5,
2-1, 2-2, 2-3, 2-4, 2-5,
3-1, 3-2, 3-3, 3-4, 3-5,
4-1, 4-2, 4-3, 4-4, 4-5,
5-1, 5-2, 5-3, 5-4, 5-5,
```

こうした 2 重ループを apply 系関数で実現する方法を説明する．まず繰り返し処理内の単一処理を担う関数 fnc() を以下のように定義する．

```
> fnc <- function(i) {
+     function(j) {
+         cat(i, "-", j, ", ", sep="")
+     }
+ }
```

関数定義の中に関数定義があって面食らうが，実行してみると難しくない．上の定義を見ると，fnc() は引数 i を取る．

```
> fnc(1)
function(j) {
        cat(i, "-", j, ", ", sep="")
    }
<environment: 0x7fc9d3c2bb48>
```

するとその返り値は内部の function(j) ... である．つまり，fnc(1) を実行すると，今度は引数に j を期待する関数が返る．fnc(1) が引数 j を取る関数を返すのだから，fnc(1) を関数とみなして今度はそれに引数 2 を渡してみよう．

```
> fnc(1)(2)
1-2,
```

これで中身の cat(i, "-", j, ", ", sep="") が実行された．このように，複数の引数を取る関数を，1 つの引数を取る関数に定義し直すことをカリー化と呼ぶ．このカリー化された関数を使い，i に 1 から 5 の値を埋め込んだ関数から成るリストを lapply() で生成できる．

```
> lapply(1:5, fnc)
```

上のリストの各要素に格納された関数は j を引数に取る関数である．1 から 5 にこの関数を適用させて改行すれば 2 重ループと同じ結果を得ることができる．ただし，我々の今の目的は cat() を使った画面表示であって，lapply() が返す値には興味がない．そこで，結果を void 変数に入れてプロンプトに戻り値が表示されないようにする．

```
> void <-
+     lapply(lapply(1:5, fnc), function(f) {
+         lapply(1:5, f)
+         cat("\n")
+     })
1-1, 1-2, 1-3, 1-4, 1-5,
2-1, 2-2, 2-3, 2-4, 2-5,
3-1, 3-2, 3-3, 3-4, 3-5,
4-1, 4-2, 4-3, 4-4, 4-5,
5-1, 5-2, 5-3, 5-4, 5-5,
```

上のプログラムには lapply() が 3 回現れ，必要以上に難解に見える．そこで，fnc() 関数に i の値を適用した結果を変数に保持しておけば，もう少し見通しの良いプログラムになる．

```
> fncj <- lapply(1:5, fnc) # 引数 i に値を適用済みで j を引数に取る関数のリスト
> void <- lapply(fncj, function(f) {
+     lapply(1:5, f)
+     cat("\n")
```

```
+ })
1-1, 1-2, 1-3, 1-4, 1-5,
2-1, 2-2, 2-3, 2-4, 2-5,
3-1, 3-2, 3-3, 3-4, 3-5,
4-1, 4-2, 4-3, 4-4, 4-5,
5-1, 5-2, 5-3, 5-4, 5-5,
```

実際には上のように処理関数に対し一部の処理（ここでは引数 i に関する処理）を 1 行目の lapply() 関数で部分的に適用して準備しておいて，2 行目以降の lapply() で残りの情報を関数に渡して一気に全てのクロス集計処理を行うことが多いだろう．

複雑な処理を行う関数に引数を一気に渡して実行するのではなく，一部の引数にのみ関数を適用した結果を関数として返すことによって，実際の処理を後に先延ばしするテクニックは，今後，複雑なデータ処理の現場で役に立つこともあるだろう．初めて目にすると複雑に思えるが，慣れてしまえば難しくないので覚えておいて損はない．理解するまで何度も読み返し，自分のものにしよう．

7.1 練習問題

以下は別の方法で 5×5 の数字の組み合わせを画面表示させたプログラムである．expand.grid() 関数と apply() 関数の使い方を help で調べて，下のプログラムを解読せよ．

```
> x <- expand.grid(1:5, 1:5)
> mat <- apply(x, MARGIN=1, FUN=function(x) paste(x[2], "-", x[1], ",", sep=""))
> every.five <- 1:length(mat) %% 5 == 0
> mat[every.five] <- paste(mat[every.five], "\n", sep="")
> cat(mat)
1-1, 1-2, 1-3, 1-4, 1-5,
 2-1, 2-2, 2-3, 2-4, 2-5,
 3-1, 3-2, 3-3, 3-4, 3-5,
 4-1, 4-2, 4-3, 4-4, 4-5,
 5-1, 5-2, 5-3, 5-4, 5-5,
```

2 行目以降が 1 文字下がっているのは，1 行目の終わりの改行文字 \n の 1 文字分であると思われる．