2_array_if_loop

2017 年 3 月 16 日

1 Introduction to Data Science with Julia

1.1 目次

2 配列

前回は値を保存しておくために変数を使うことを学びました。今回は複数の値を保存することのできる配列について学びましょう。

と、その前にまずは前回の復習として、5 人の学生の試験の点数の平均を求めるプログラムを書いてみてください。各学生の点数は

九州太郎: 73 九州花子: 86 中州河旗: 30 福岡健 : 48 博多勉 : 53

とします。

In [ ]:

出来たでしょうか？ あなたが書いたコードは 1, 2 のどちらのタイプになったでしょうか？それとも全く別の方法？

(1)
```
(73 + 86 + 30 + 48 + 53) / 5
```

(2)
```
taro = 73
hanako = 86
kawabata = 30
ken = 48
tsutomu = 53
(taro + hanako + kawabata + ken + tsutomu) / 5
```

(1) は前回言ったように再利用性が低いコードになっています。今は平均しか求めなかったのでこのようなコードでも良いですが、他に分散や標準偏差も一緒に求めよと言われた場合、学生の点数が変わるごとに計算式自体に修正を加えなければならないため大変です。

(2) は変数に値を保存することによって再利用性が (1) に比べて上がっています。平均を求めるだけの場合ならば (1) よりタイピングする量が多くなり一見すると非効率的です。しかし、分散、標準偏差に関しても一度このように変数を使って計算式を書いておけば、学生の点数だけを修正するだけで計算式自体に手を加えず再利用することが出来ます。また修正箇所が少なくなることでバグが発生しづらくなります。

さて、今は 5 人と少数だったので良いですが、これが 30 人、100 人となったら変数名を考えるだけで大変です。このように多数の要素を一つにまとめて保存したい場合には配列を使うと便利です。配列を使うと一つの変数名で複数の要素を保存することが出来ます。

Julia で配列を作るには [ ] で格納したい要素を囲みます。各要素はコンマ"," または空白スペース" " で区切ります。コンマで区切ると数学で言う縦ベクトルが、スペースで区切ると横ベクトルが出来ます。

```
In [1]: score_col = [73, 86, 30, 48, 53]
```

```
Out[1]: 5-element Array{Int64,1}:
          73
          86
          30
          48
          53
```

```
In [2]: score_row = [73 86 30 48 53]
```

```
Out[2]: 1 × 5 Array{Int64,2}:
          73  86  30  48  53
```

変数に値を代入した時は代入した数だけが出力されましたが、配列を作った時には数だけでなく

```
5-element Array{Int64,1}
```

とも出力されました。これは今作った配列の型を表示しています。それぞれの意味を分解すると

- 5-element: 配列の要素数は 5 つ
- Array: 変数の型は配列型
- Int64: 配列の各要素は倍精度整数型
- 1: 1 次元配列

という意味になります。プログラミング言語での 1 次元や 2 次元というのは線形代数の次元とは違い、1 次元方向に広がっているか、2 次元的に広がっているかという意味になります。そのため行列は要素がいかなるものであっても 2 次元配列です。

横ベクトルの配列の型についても同様に意味を分解すると

```
1 × 5 Array{Int64,2}
```

- 1 × 5: サイズが 1 × 5 の行列
- Array: 変数の型は配列型
- Int64: 配列の各要素は倍精度整数型
- 2: 2 次元配列

という意味になります。横ベクトルは横に 1 次元的に広がっているのに 2 次元配列というのは奇妙な感じがしますが、Julia では横ベクトルは薄い行列と認識されます。これは言語上の仕様なので受け入れてください。

配列の i 番目の要素を抜き出すためには

配列名`[i]`

とします。C 言語や Python などの多くのプログラミング言語では配列の添字が 0 から始まりますが、MATLAB や Julia では通常の数学同様 1 から始まります。

```
In [3]: score_col[2] # 2 番目の要素
```

```
Out[3]: 86
```

```
In [4]: score_col[10] # 存在しないものを指定するとエラーが出ます
```

```
BoundsError: attempt to access 5-element Array{Int64,1} at index [10]
```

```
in execute_request(::ZMQ.Socket, ::IJulia.Msg) at /home/tk/.julia/v0.5/IJulia/s

in eventloop(::ZMQ.Socket) at /home/tk/.julia/v0.5/IJulia/src/eventloop.jl:8

in (::IJulia.##13#19)() at ./task.jl:360
```

配列の要素の値の更新は普通の変数同様に行うことが出来ます。

```
In [5]: score_col[2] = 14
```

```
Out[5]: 14
```

```
In [6]: score_col # 第 2 要素が 14 になった
```

```
Out[6]: 5-element Array{Int64,1}:
 73
 14
 30
 48
 53
```

しかし、整数型の配列に文字列を代入することは出来ません。

```
In [7]: score_col[2] = "foo" # エラーが出る
```

```
MethodError: Cannot `convert` an object of type String to an object of type Int6
This may have arisen from a call to the constructor Int64(...),
since type constructors fall back to convert methods.
```

```
in setindex!(::Array{Int64,1}, ::String, ::Int64) at ./array.jl:415

in execute_request(::ZMQ.Socket, ::IJulia.Msg) at /home/tk/.julia/v0.5/IJulia/s

in eventloop(::ZMQ.Socket) at /home/tk/.julia/v0.5/IJulia/src/eventloop.jl:8

in (::IJulia.##13#19)() at ./task.jl:360
```

また、整数型の配列には 1.0 や 2170.0 などの整数に変換できる小数は代入することは出来ますが、 1.5 などの小数は代入することは出来ません。

```
In [8]: score_col[1] = 190.0

Out[8]: 190.0

In [9]: score_col # 自動的に整数型に変換される。

Out[9]: 5-element Array{Int64,1}:
         190
          14
          30
          48
          53

In [10]: score_col[1] = 1.5 # 整数に変換できないためエラーが出る


        InexactError()



         in setindex!(::Array{Int64,1}, ::Float64, ::Int64) at ./array.jl:415


In [11]: score_col # 第 1 要素に変化なし

Out[11]: 5-element Array{Int64,1}:
          190
           14
           30
           48
           53
```

整数だけでなく小数も代入できる配列を作るためには始めから

```
[10.0, 12, 4, 2]
```

などと小数を混ぜて配列を作るか、または、下記のように明示的に小数型であることを宣言して配列を作ります。

```
Float64[10, 12, 4, 2] # Float64: 倍精度浮動小数点
```

```
In [12]: Float64[10, 12, 4, 2]
```

```
Out[12]: 4-element Array{Float64,1}:
          10.0
          12.0
           4.0
           2.0
```

C 言語や Java などのプログラミング言語では配列の要素に違う型の値を同時に入れることは出来ませんが、Julia では違う型の値も同時に格納することが出来ます。この時配列は Any 型になります。

```
In [13]: num_str = [1, 2.5, "hoge"] # 整数、浮動小数点、文字列が混在
```

```
Out[13]: 3-element Array{Any,1}:
          1
          2.5
           "hoge"
```

Any 型の配列では各要素を自由に更新することが出来ます。

```
In [14]: num_str[1]
```

```
Out[14]: 1
```

```
In [15]: num_str[1] = "piyopiyo"
```

```
Out[15]: "piyopiyo"
```

```
In [16]: num_str # 第 1 要素が 1 から "piyopiyo" に変わる
```

```
Out[16]: 3-element Array{Any,1}:
           "piyopiyo"
          2.5
           "hoge"
```

小数の時同様

```
Any[12, 3, 4, 2]
```

とすると要素がいかなるものであろうとも Any 型の配列が出来ます。しかし、Any 型はその柔軟性故にバグが起こる可能性が高くなるのでなるべく使わないようにしましょう。

プログラミングにおいてエラーは忌み嫌うものではなく、むしろ自分を成長させてくれるものです。エラーが出るとすぐに自分の間違いに気づくことが出来ますが、エラーが出ないとバグを見落としてしまう可能性が高くなります。

```
In [17]: Any[12, 3, 4, 2]
```

```
Out[17]: 4-element Array{Any,1}:
          12
           3
           4
           2
```

```
In [ ]:
```

目次に戻る

3　辞書

配列では各要素を指定するのに配列の何番目にあるかで指定しました。配列は学生の成績の平均などを求めたい場合など個人を特定する必要のないデータを保存するには良いですが、各個人の成績を知りたい場合、添字番号と学生を対応付けなければならないので不便です。

添字番号ではなく、名前や学籍番号で指定できたほうが便利なのはいうまでもないでしょう。このようなことをしたい場合には辞書 (連想配列) を使います。辞書を使うと

```
scores["九州太郎"]
```

といった方法で値を指定することが出来ます。

辞書の基本構文

```
Dict(a=>b, ...)
```

ここで、a は key、b は value と呼ばれる。

```
In [18]: scores = Dict("九州太郎" => 73, "九州花子" => 86)
```

```
Out[18]: Dict{String,Int64} with 2 entries:
           "九州… => 73
           "九州… => 86
```

```
In [19]: scores["九州太郎"]
```

```
Out[19]: 73
```

```
In [20]: scores["九州花子"]
```

```
Out[20]: 86
```

辞書の全ての key を調べるには keys 関数を使います

```
In [21]: keys(scores)
```

```
Out[21]: Base.KeyIterator for a Dict{String,Int64} with 2 entries. Keys:
           "九州太郎"
           "九州花子"
```

辞書の全ての value を調べるには values 関数を使います

```
In [22]: values(scores)
```

```
Out[22]: Base.ValueIterator for a Dict{String,Int64} with 2 entries. Values:
           73
           86
```

```
In [ ]:
```

次のように名前と点数が別の配列に入っていて、この 2 つを辞書に統合したい場合には zip 関数を使うと便利です。

※辞書型では要素の順番に意味はないので作った後の中身は元の配列の中身とは異なります。

```
In [23]: 名前 = ["九州太郎", "九州花子", "中州河旗", "福岡健", "博多勉"]
         点数 = [73, 86, 30, 48, 53]
         scores2 = Dict(zip(名前, 点数))
```

```
Out[23]: Dict{String,Int64} with 5 entries:
           "中州… => 30
           "九州… => 73
           "博多… => 53
           "福岡… => 48
           "九州… => 86
```

```
In [24]: scores2["博多勉"]
```

```
Out[24]: 53
```

```
In [25]: keys(scores2) # 上の 名前 配列とは順番が違う
```

```
Out[25]: Base.KeyIterator for a Dict{String,Int64} with 5 entries. Keys:
           "中州河旗"
           "九州太郎"
           "博多勉"
           "福岡健"
           "九州花子"
```

```
In [ ]:
```

目次に戻る

4 Tuple

配列や辞書ではあとで値を更新することが出来ましたが、時としては定数のように値を更新したくない場合もあります。そのような時には Tuple を使いましょう。

配列を作るときは要素を [ ] で囲みましたが、Tuple では ( ) で囲みます。

※ 本当は ( ) がなくても、コンマ区切りにするだけでも良いのですが、後からみて Tuple だということがわかりやすいので ( ) で囲むことをおすすめします。

```
In [26]: A = (10, "Hoge", [1, 2, 3])
```

```
Out[26]: (10,"Hoge",[1,2,3])
```

```
In [27]: A[1] # 要素へのアクセス方法は配列と同じ

Out[27]: 10

In [28]: A[2]

Out[28]: "Hoge"

In [29]: A[3]

Out[29]: 3-element Array{Int64,1}:
          1
          2
          3

In [30]: A[3][2]

Out[30]: 2

In [31]: A[1] = 100 # 値を更新することが出来ない

         MethodError: no method matching setindex!(::Tuple{Int64,String,Array{Int64,1}},



          in execute_request(::ZMQ.Socket, ::IJulia.Msg) at /home/tk/.julia/v0.5/IJulia/s

          in eventloop(::ZMQ.Socket) at /home/tk/.julia/v0.5/IJulia/src/eventloop.jl:8

          in (::IJulia.##13#19)() at ./task.jl:360

In [ ]:
```

目次に戻る

5 if 文

次に条件分岐を学びましょう。条件分岐とは下記のフローチャートのように条件により処理を変えるような操作のことを言います。[Conditional (computer programming) - Wikipedia]

Julia での if 文の基本構文は次のようになります。

```
if 条件式 1
    処理 1
elseif 条件式 2
    処理 2
elseif 条件式 3
.
.
```

```
    .
    else
        処理 n
    end
```

例として n が - 正の数の時: n は正の数です - 負の数の時: n は負の数です - 0 の時: n は 0 ですと と出力するプログラムを書いてみます。

```
In [32]: n = 10 # n の値を変えて実行してみましょう
         if n > 0
             print("$n は正の数です")
         elseif n < 0
             print("$n は負の数です")
         else
             print("$n は 0 です")
         end
```

```
10 は正の数です
```

```
In [ ]:
```

if 文の中にさらに if 文を入れることも出来ます

```
In [33]: n = 10
         if n > 0
             println("$n は正の数です")

             if n % 2 == 0
                 println("$n は偶数です")
             end

         elseif n < 0
             println("$n は負の数です")
         else
             println("$n は 0 です")
         end
```

```
10 は正の数です
10 は偶数です
```

```
In [ ]:
```

目次に戻る

6　ループ文

ループ分を使うとある処理を繰り返し行うことが出来ます。ループ分には for 文と while 文の 2 つがあります。

6.1　for 文

for 文を使うとある処理を指定した回数繰り返すことが出来ます。
for 文の基本構文

```
for dummy_var in list
    body
end
```

```
In [34]: for i in 1:10
             println("Hello World!")
         end
```

```
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
```

```
In [35]: for i in 1:10
             println(i)
         end
```

```
1
2
3
4
5
6
7
8
9
10
```

```
In [36]: for i in 1:2:10 # 1 ～ 10 までの数を 2 刻み
             println(i)
         end
```

```
1
3
5
7
9
```

if 文同様、for 文の中に for 文を入れることも出来ます。

```
In [37]: for i in 1:3
             for j in i:3
                 println((i,j))
             end
         end
```

```
(1,1)
(1,2)
(1,3)
(2,2)
(2,3)
(3,3)
```

```
In [ ]:
```

他のプログラミング言語では普通出来ませんが、Julia では多重の for 文では次の例のように for . . . end を省略する書き方が出来ます。

```
for i in ..., j in ...
    body
end
```

ただし、このような書き方ができるのは

```
for ...
    for ...
        body
```

のように for が連続する場合だけで

```
for ...
    body
    for ...
        body
```

のように for と for の間に処理が入る場合には使えません。

```
In [38]: for i in 1:3, j in i:3
             println((i,j))
         end
```

```
(1,1)
(1,2)
(1,3)
(2,2)
(2,3)
(3,3)
```

```
In [ ]:
```

list の部分には配列を使うことも出来ます。

```
In [39]: items = ["Kyushu", "University", "School of Interdisciplinary Science and Innov
         for item in items
             println(item)
         end
```

```
Kyushu
University
School of Interdisciplinary Science and Innovation
```

```
In [ ]:
```

目次に戻る

6.2 while 文

for 文は指定した回数繰り返すような構文でした。while 文ではループ回数に上限はなく、ある条件が満たされるまでループが回り続けます。

While 文の基本構文

```
while 条件式
    処理
end
```

```
In [40]: i = 0
         while i < 10
             println(i)
             i += 1
         end
```

```
0
```

```
1
2
3
4
5
6
7
8
9
```

while 文では常に条件が true になると無限ループを起こし、その後の処理が実行されなくなってしまうので注意してください。無限ループに陥ってしまったら上の ■をクリックし、処理を中断させてください。

無限ループに陥る例

```
i = 0
while true
    i += 1
end
```

```
In [ ]:
```

目次に戻る

7 練習問題

7.1 1.

要素が [1, 2, 3, . . . , 1000] となるような配列を作成せよ

ヒント: collect 関数 または 内包表記 (comprehension)

```
In [ ]:
```

7.2 2.

スイス、エジプト、ホンジュラスの公用語を調べ、国名を key, 公用語を value とする辞書を作成せよ

ヒント: 公用語が複数ある場合は value を 配列または Tuple にする

```
In [ ]:
```

7.3 3.

身長・体重より BMI を計算し、計算結果より 痩せすぎ、標準、肥満を if 文と print 関数を使って出力せよ

BMI の計算方法:

$$\text{BMI} = \frac{\text{体重 (kg)}}{\text{身長 (m)}^2}$$

BMI による肥満判定

```
<th class="tg-031e">BMI</th>
<th class="tg-031e">判定</th>

<td class="tg-031e">〜18.5</td>
<td class="tg-031e">痩せている</td>

<td class="tg-031e">18.5〜25</td>
<td class="tg-031e">標準</td>

<td class="tg-yw4l">25〜</td>
<td class="tg-yw4l">肥満</td>
```

In [ ]:

7.4 4.

for 文を使って 1 から n までの整数を足し上げるプログラムを作成せよ。

In [ ]:

7.5 5.

(1) 要素が [1, 2, 3, . . . , 1000] となるような配列を作成せよ (問題 1 と同じ) (2) (1) で作成した配列の各要素を for 文を使って 2 倍せよ

In [ ]:

7.6 6.

for 文を使って配列

```
x = [3,4,4,4,2,1,4,5,2,3]
```

の要素を全て足し上げるプログラムを作成せよ。また、求めた結果を sum(x) と比較せよ
ヒント: 配列の要素数は length 関数で知ることができる。

In [ ]:

7.7 7.

for 文を使って九九表を作成せよ
出力例

```
1   2   3   4   5   6   7   8   9
2   4   6   8  10  12  14  16  18
3   6   9  12  15  18  21  24  27
4   8  12  16  20  24  28  32  36
5  10  15  20  25  30  35  40  45
```

```
6  12  18  24  30  36  42  48  54
7  14  21  28  35  42  49  56  63
8  16  24  32  40  48  56  64  72
9  18  27  36  45  54  63  72  81
```

In [ ]:

7.8 8.

フィボナッチ数列の第 n 項目を求めるプログラムを作成せよ

フィボナッチ数列 $F_0 = 0, F_1 = 1, F_{n+2} = F_{n+1} + F_n$

In [ ]:

7.9 9.

友人から 1000 円借りていたとする。10 日で 1 割の金利がついた場合、何日後に借金は 100 万円を超えるか求めよ

トイチ- Wikipedia

In [ ]:

目次に戻る