4_array_programming

2017 年 3 月 16 日

1 Introduction to Data Science with Julia

2 目次

3 配列操作

今回は配列操作について学びます。データ分析をする際には配列単位で操作をすることが多くなるのでしっかりと身につけてください。

まず初めに復習として、配列は [ ] で配列の要素にしたいものを囲んで作るのでした。

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

```
Out[1]: 5-element Array{Int64,1}:
          1
          2
          3
          4
          5
```

今度は、2 次元配列 (行列) を作ってみます。2 次元配列を作るには各行の要素を**スペース区切り**で書き、行の終わりをセミコロン; で区切ります。

```
In [2]: [1 2 3; 4 5 6 ; 7 8 9]
```

```
Out[2]: 3 × 3 Array{Int64,2}:
          1  2  3
          4  5  6
          7  8  9
```

各行の区切りは改行でも大丈夫です。改行を使って書くとより行列に近い形に書けるので見やすくなります。

```
In [3]: [1 2 3
         4 5 6
         7 8 9]
```

```
Out[3]: 3 × 3 Array{Int64,2}:
         1  2  3
         4  5  6
         7  8  9
```

要素を 0 に初期化した $M \times N$ 行列を作るときは zeros を使うと便利です。

```
In [4]: M = 5
        N = 3
        zeros(M, N) # zeros(Int, M, N) とすると Int 型になる
```

```
Out[4]: 5 × 3 Array{Float64,2}:
         0.0  0.0  0.0
         0.0  0.0  0.0
         0.0  0.0  0.0
         0.0  0.0  0.0
         0.0  0.0  0.0
```

全要素を 1 に初期化する場合は ones を使います。

```
In [5]: ones(M, N) # ones(Int, M, N) とすると Int 型になる
```

```
Out[5]: 5 × 3 Array{Float64,2}:
         1.0  1.0  1.0
         1.0  1.0  1.0
         1.0  1.0  1.0
         1.0  1.0  1.0
         1.0  1.0  1.0
```

任意の数字や文字列で初期化した配列を作るには fill を使います。

```
In [6]: fill(3, 5, 2)
```

```
Out[6]: 5 × 2 Array{Int64,2}:
         3  3
         3  3
         3  3
         3  3
         3  3
```

```
In [7]: fill("hoge", 5, 2)
```

```
Out[7]: 5 × 2 Array{String,2}:
```

```
          "hoge"  "hoge"
          "hoge"  "hoge"
          "hoge"  "hoge"
          "hoge"  "hoge"
          "hoge"  "hoge"
```

```
In [ ]:
```

複数の配列を [ ] で囲むことで配列を結合することが出来ます。

```
In [8]: x = [1, 2, 3]
        y = [4, 5, 6]
        [x y] # 縦ベクトル２つを結合。hcat 関数を使っても同様のことが出来る
```

```
Out[8]: 3×2 Array{Int64,2}:
         1  4
         2  5
         3  6
```

```
In [9]: x = [1 2 3]
        y = [4 5 6]
        [x y] # 横ベクトル２つを結合
```

```
Out[9]: 1×6 Array{Int64,2}:
         1  2  3  4  5  6
```

```
In [10]: x = [1,2,3]
         y = [4,5,6]
         [x; y] # vcat 関数を使っても同様のことが出来る
```

```
Out[10]: 6-element Array{Int64,1}:
          1
          2
          3
          4
          5
          6
```

```
In [ ]:
```

push! を使うと一次元配列の末尾に要素を追加することが出来ます。

```
In [11]: x = [1,2,3]
         @show x
         push!(x, 10) # push!(x, 10, 11) などとすれば複数の要素を一度についかできる
         @show x
```

```
x = [1,2,3]
x = [1,2,3,10]
```

```
Out[11]: 4-element Array{Int64,1}:
           1
           2
           3
          10
```

push! とは逆に末尾の要素を取り出すには pop! を使います

```
In [12]: @show x
         pop!(x)
         @show x
```

```
x = [1,2,3,10]
x = [1,2,3]
```

```
Out[12]: 3-element Array{Int64,1}:
          1
          2
          3
```

unshift! を使うと先頭に要素を追加することが出来ます

```
In [13]: unshift!(x, 5) # 複数の要素を一度に追加も出来る
```

```
Out[13]: 4-element Array{Int64,1}:
          5
          1
          2
          3
```

shift! を使うと先頭に要素を取り出すことが出来ます

```
In [14]: @show x
         shift!(x)
         @show x
```

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

```
Out[14]: 3-element Array{Int64,1}:
          1
          2
          3
```

```
In [ ]:
```

2 次元配列の (i, j) 成分へのアクセスは $A[i, j]$ のように指定します。

```
In [15]: A = [1 2 3
              4 5 6
              7 8 9]
         A[1, 3]
```

```
Out[15]: 3
```

3 次元以上は

```
Array{Any}(3,3,3) # 3 x 3 x 3 配列
```

と雛形を作ってから、後で値を代入する必要が有ります。ただし、3 次元以上の配列はあまり使うことはないでしょう。

```
In [16]: X = Array{Int}(3,3,3) # 初期値はコンピュータが勝手に決める
```

```
Out[16]: 3 × 3 × 3 Array{Int64,3}:
         [:, :, 1] =
          140052001667728  140052001886512  140052001887792
          140052001667824  140052001887952  140052001895024
          140052001667888  140052001887312  140052001895088

         [:, :, 2] =
          140052001896720  140052001896912  140052002092656
          140052001896784  140052001994800  140052002092720
          140052001896848  140052001994864  140052001667312

         [:, :, 3] =
          140052001667344  140052001667440  140052001667536
          140052001667376  140052001667472  140052001667568
          140052001667408  140052001667504  140052001886192
```

```
In [ ]:
```

目次に戻る

3.1 要素を抜き出す

配列の要素をある区間抜き出すには range を使います。range とは for 文のところでも使っていた 1:10 などです。

range の書き方は

```
start:step:stop
```

の様に書きます。step を省略すると 1 刻みになります。

この range を使って

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

から奇数番目の要素だけを抜き出す場合には以下のようにします。

```
In [17]: x = [1, 2, 3, 4, 5]
         x[1:2:5] # 1 〜 5 までの数を 2 刻み
```

```
Out[17]: 3-element Array{Int64,1}:
          1
          3
          5
```

添字に : を使うと全要素という意味になります。

```
In [18]: x[:]
```

```
Out[18]: 5-element Array{Int64,1}:
          1
          2
          3
          4
          5
```

end を使うと要素の最後までを表すことが出来ます。

```
In [19]: x[1:2:end]
```

```
Out[19]: 3-element Array{Int64,1}:
          1
          3
          5
```

末端 -1 の要素までなどとする場合は end -1 とすれば大丈夫です。

```
In [20]: x[1:2:end-1]
```

```
Out[20]: 2-element Array{Int64,1}:
          1
          3
```

```
In [ ]:
```

この書き方は行列に関しても同じです。
3 x 3 行列から 1 列目を抜き出す

```
In [21]: A = [1 2 3
              4 5 6
              7 8 9]
         A[:, 1]
```

```
Out[21]: 3-element Array{Int64,1}:
          1
          4
```

```
 7
```

3 x 3 行列から 1 行目を抜き出す

```
In [22]: A[1,:]
```

```
Out[22]: 3-element Array{Int64,1}:
          1
          2
          3
```

3 x 3 行列から 2 x 2 行列を抜き出す

```
In [23]: A[1:2, 1:2]
```

```
Out[23]: 2 × 2 Array{Int64,2}:
          1  2
          4  5
```

```
In [ ]:
```

要素の指定には true, false も使うことが出来ます。抜き出す必要のあるところを true 、抜き出す必要のないところを false にした配列を添え字として使うと、true に対応した要素だけを抜き出すことが出来ます。

```
In [24]: x = collect(1:5) # collect は range を配列に変換する
         truth = [true, false, true, true, false]
         x[truth]
```

```
Out[24]: 3-element Array{Int64,1}:
          1
          3
          4
```

多次元配列に関しても同様にできます。

```
In [25]: x = [1 2
              3 4]
         truth = [true false
                  false true]
         x[truth]
```

```
Out[25]: 2-element Array{Int64,1}:
          1
          4
```

```
In [ ]:
```

目次に戻る

3.2 値の更新

上記の要素を抜き出し方を覚えると、配列の要素を一度に更新することが出来ます。例として配列の奇数番目の要素を全て 100 にするには以下のようにします。

```
In [26]: x = collect(1:5)
         x[1:2:end] = 100
         x
```

```
Out[26]: 5-element Array{Int64,1}:
          100
            2
          100
            4
          100
```

奇数番目の数だけ 10 倍にするには次のようになります。

```
In [27]: x = collect(1:5)
         x[1:2:end] *= 10
         x
```

```
Out[27]: 5-element Array{Int64,1}:
          10
           2
          30
           4
          50
```

要素を抜き出して出来た配列と同じサイズの配列を代入すると対応する要素を更新することが出来ます。

```
In [28]: x = collect(1:5)
         x[1:2:end] = [100, 200, 300]
         x
```

```
Out[28]: 5-element Array{Int64,1}:
          100
            2
          200
            4
          300
```

```
In [ ]:
```

目次に戻る

3.3 配列演算

配列演算ではある 2 つの配列の対応する要素ごとに計算が行われます。配列演算をするにはいままで使ってきた演算子にドットをつけるだけです。

```
In [29]: x = [1, 2, 3];
         y = [4, 5, 6];
```

$$\begin{bmatrix} a_1 \\ a_2 \end{bmatrix} .+ \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} a_1 + b_1 \\ a_2 + b_2 \end{bmatrix}$$

```
In [30]: x .+ y # 加算と減算ではドットはつけてもつけなくてもどちらでも構いません。
```

```
Out[30]: 3-element Array{Int64,1}:
          5
          7
          9
```

$$\begin{bmatrix} a_1 \\ a_2 \end{bmatrix} .- \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} a_1 - b_1 \\ a_2 - b_2 \end{bmatrix}$$

```
In [31]: x .- y
```

```
Out[31]: 3-element Array{Int64,1}:
          -3
          -3
          -3
```

$$\begin{bmatrix} a_1 \\ a_2 \end{bmatrix} .* \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} a_1 * b_1 \\ a_2 * b_2 \end{bmatrix}$$

```
In [32]: x .* y
```

```
Out[32]: 3-element Array{Int64,1}:
           4
          10
          18
```

$$\begin{bmatrix} a_1 \\ a_2 \end{bmatrix} ./ \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} a_1/b_1 \\ a_2/b_2 \end{bmatrix}$$

```
In [33]: x ./ y
```

```
Out[33]: 3-element Array{Float64,1}:
          0.25
          0.4
          0.5
```

$$\begin{bmatrix} a_1 \\ a_2 \end{bmatrix} .\hat{} n = \begin{bmatrix} a_1^n \\ a_2^n \end{bmatrix}$$

```
In [34]: x.^2 # x の各要素を 2 乗
```

```
Out[34]: 3-element Array{Int64,1}:
          1
          4
          9
```

大きさが違いものどうしでは配列演算は出来ません。

```
In [35]: [1, 2, 3] + [4, 5, 6, 7]


        DimensionMismatch("dimensions must match")



         in _elementwise(::Base.#+, ::Type{Int64}, ::Array{Int64,1}, ::Array{Int64,1}) a

         in +(::Array{Int64,1}, ::Array{Int64,1}) at ./arraymath.jl:63

         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
```

しかし、定数を加減乗除することは出来ます。

```
In [36]: x = [1, 2, 3]
         x + 1

Out[36]: 3-element Array{Int64,1}:
          2
          3
          4

In [37]: x - 1

Out[37]: 3-element Array{Int64,1}:
          0
          1
          2

In [38]: x * 10

Out[38]: 3-element Array{Int64,1}:
          10
          20
          30
```

```
In [39]: x / 10
```

```
Out[39]: 3-element Array{Float64,1}:
          0.1
          0.2
          0.3
```

定数 / 配列のときはドットをつけないとエラーが出ます。ドットをつけ忘れないように気をつけましょう。

```
In [40]: 10 / x # エラーが出る
```

```
        MethodError: no method matching /(::Int64, ::Array{Int64,1})
    Closest candidates are:
      /(::Integer, ::Integer) at int.jl:35
      /(::Real, ::Complex{T<:Real}) at complex.jl:182
      /(::Union{Int16,Int32,Int64,Int8}, ::BigFloat) at mpfr.jl:272
      ...



         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 [41]: 10 ./ x
```

```
Out[41]: 3-element Array{Float64,1}:
          10.0
           5.0
           3.33333
```

```
In [ ]:
```

このドット ". " を使った演算は四則演算だけに限りません。関数に関しても ‘julia　　関数名.(collection) # collection: 配列や Tuple などと関数とカッコの間にドットを入れることで collection の各要素に関数を作用させる意味になります。得られる結果としては map 関数を使った時と同じです。

```
In [42]: x = linspace(0, 2π, 100) # linspace(a,b,n):  a:(b-a)/n:b となる range。n を指定し
         y = sin.(x)
```

```
Out[42]: 100-element Array{Float64,1}:
           0.0
           0.0634239
```

```
 0.126592
 0.189251
 0.251148
 0.312033
 0.371662
 0.429795
 0.486197
 0.540641
 0.592908
 0.642788
 0.690079
 ⋮
-0.642788
-0.592908
-0.540641
-0.486197
-0.429795
-0.371662
-0.312033
-0.251148
-0.189251
-0.126592
-0.0634239
 6.43249e-16
```

```
In [43]: map(sin, x) # map(func, collection): collection の各要素に func を作用させる
```

```
Out[43]: 100-element Array{Float64,1}:
 0.0
 0.0634239
 0.126592
 0.189251
 0.251148
 0.312033
 0.371662
 0.429795
 0.486197
 0.540641
 0.592908
 0.642788
 0.690079
 ⋮
-0.642788
-0.592908
```

```
 -0.540641
 -0.486197
 -0.429795
 -0.371662
 -0.312033
 -0.251148
 -0.189251
 -0.126592
 -0.0634239
  6.43249e-16
```

```
In [ ]:
```

目次に戻る

3.4 要素の比較

配列の要素同士を比較する場合もドットを使って .< などとします。返り値として true, false が入った配列が返ってきます。

```
In [44]: x = [1, 2, 3]
         x .<= 2
```

```
Out[44]: 3-element BitArray{1}:
          true
          true
         false
```

```
In [45]: x = [1, 2, 3]
         y = [1, 100, 3]
         x .== y
```

```
Out[45]: 3-element BitArray{1}:
          true
         false
          true
```

```
In [46]: x .< y
```

```
Out[46]: 3-element BitArray{1}:
         false
          true
         false
```

返り値の true, false が入った配列を元の配列の添字として使うことで、ある条件を満たす要素を抜き出すことが出来ます。

```
In [47]: # 1 〜 10 が入った配列から 5 未満の数を抜き出す。
         x = collect(1:10)
```

```
          x[x .< 5]
```

```
Out[47]: 4-element Array{Int64,1}:
          1
          2
          3
          4
```

```
In [ ]:
```

```
    x = [67,57,69,64,54,13,78,26,29,37]
```

という配列の中から奇数を抜き出すには

```
    x[x .% 2 .!= 0]
```

と書くことも出来ますが、引数が奇数か否かを判定する関数 isodd を使うことで次のようにも書けます。

```
In [48]: x = [67,57,69,64,54,13,78,26,29,37]
         x[isodd.(x)]
```

```
Out[48]: 6-element Array{Int64,1}:
          67
          57
          69
          13
          29
          37
```

```
In [ ]:
```

次のように 1 列目に学生の名前、2, 3 列目に試験の点数が入った配列を考えます。

```
score = [
    "九州太郎" 60 88
    "九州花子" 70 43
    "博多勉" 90 37
]
```

この配列から 九州花子 の行を抜き出すには以下のようになります。
score[:,1] .== “九州花子” で抜き出す行を指定し、: で列全体を抜き出しています。

```
In [49]: score = [
             "九州太郎" 60 88
             "九州花子" 70 43
             "博多勉" 90 37
         ]
         score[score[:,1] .== "九州花子", :]
```

```
Out[49]: 1 × 3 Array{Any,2}:
```

```
            "九州花子"  70  43
```

```
In [ ]:
```

目次に戻る

3.5 配列を扱う上での注意

配列をコピーする際には次のように注意が必要です。

```
In [50]: x = [1, 2, 3]
         y = x
         x[1] = 10
         @show x, y
```

```
(x,y) = ([10,2,3],[10,2,3])
```

```
Out[50]: ([10,2,3],[10,2,3])
```

上の例では 1. 変数 x に配列 [1, 2, 3] を代入する 1. y という変数に x を代入 1. 配列 x の第 1 要素を更新という流れになっていますが、x の値だけを更新したはずなのに、何故か y の値までの変わってしまっています。

y に影響を与えないようにするには、y へ x をコピーするときに

```
y = x[:]
```

とするか、または

```
y = copy(x)

or

y = deepcopy(x)
```

とします。

```
In [51]: x = [1, 2, 3]
         y = copy(x)
         x[1] = 10
         @show x, y
```

```
(x,y) = ([10,2,3],[1,2,3])
```

```
Out[51]: ([10,2,3],[1,2,3])
```

copy と deepcopy は引数の配列の中身が数値だけの時はどちらも動作としては変わりませんが、配列の中に配列を含んでいるような場合では動作が異なります。

```
In [52]: x = [[1, 2, 3], 4, 5]
```

```
y = copy(x)
y[1][1] = 100
@show x, y
```

```
(x,y) = (Any[[100,2,3],4,5],Any[[100,2,3],4,5])
```

```
Out[52]: (Any[[100,2,3],4,5],Any[[100,2,3],4,5])
```

```
In [53]: x = [[1, 2, 3], 4, 5]
y = deepcopy(x)
y[1][1] = 100
@show x, y
```

```
(x,y) = (Any[[1,2,3],4,5],Any[[100,2,3],4,5])
```

```
Out[53]: (Any[[1,2,3],4,5],Any[[100,2,3],4,5])
```

```
In [ ]:
```

目次に戻る

4 練習問題

4.1 1.

ビンゴカードを作成せよ。ここで各列の数字は以下の規則に従う 1 列目: 1〜15 2 列目: 16〜30 3 列目: 31〜45 4 列目: 46〜60 5 列目: 61〜75

中央は 0 をする。

ヒント: shuffle 関数

出力例

```
 7  20  34  59  64
 4  23  43  60  74
 2  25   0  46  62
 1  22  39  49  70
13  30  31  50  75
```

```
In [ ]:
```

4.2 2.

配列

```
x = collect(1:100)
```

から 3 の倍数番目の要素を抜き出せ。

```
In [ ]:
```

4.3 3.

配列

```
x = collect(Float64, 1:100)
```

の奇数番目の項を 100 倍に、偶数番目の項を 1/100 倍に更新せよ。

In [ ]:

4.4 4.

配列

```
srand(1) # 乱数のシード
x = rand(10)
```

から値が 0.5 未満の項を抜き出せ。

In [ ]:

目次に戻る