基礎プログラミングII

第3回　配列とハッシュの応用

今回学ぶこと

配列処理メソッド
様々な条件での並べ替え
データの格納と検索・絞り込み

今回学ぶこと

配列処理メソッド

様々な条件での並べ替え

データの格納と検索・絞り込み

配列処理メソッド

1. 集合的な処理を行うメソッド
2. 配列用のメソッド
3. 文字列固有のメソッド

配列処理メソッド

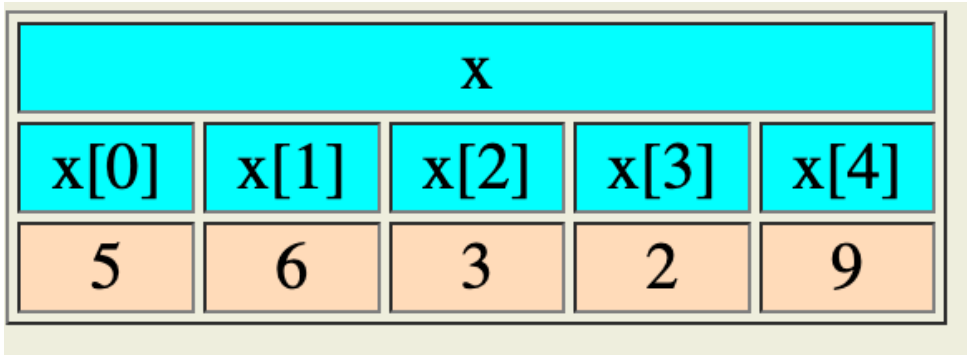
1. 集合的な処理を行うメソッド
2. 配列用のメソッド
3. 文字列固有のメソッド

配列と文字列

配列と文字列はどちらも要素の並びと捉えられる。

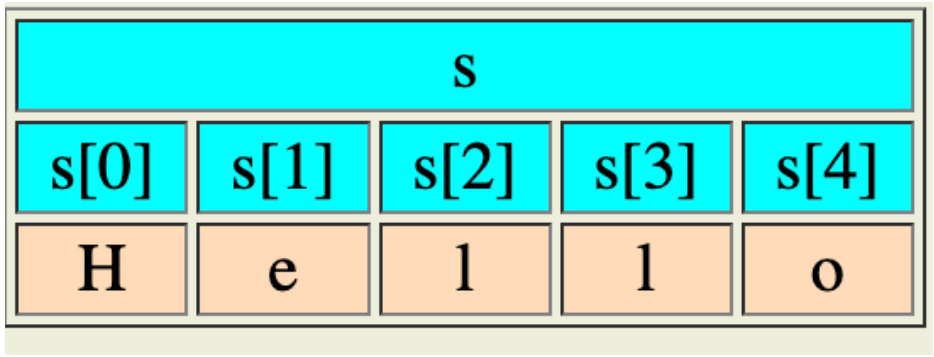
配列

x = [5, 6, 3, 2, 9]の格納イメージ



文字列

s = “Hello”の格納イメージ



メソッド“+”: 連結・結合

メソッド“+”は2つのオブジェクトを繋げた新しい配列/文字列を返す。

配列

元の配列と別の配列を繋げた新しい配列を返す。元の配列は破壊されない。

文字列

元の文字列と別の文字列を繋げた新しいも文字列を返す。元の文字列は破壊されない。

メソッド“+”: 連結・結合

メソッド“+”は2つのオブジェクトを繋げた新しい配列/文字列を返す。

配列

x = [3, 4, 1, 2]
y = [9, 8, 7]

x + y
=> [3, 4, 1, 2, 9, 8, 7]

x => [3, 4, 1, 2]
y => [9, 8, 7]

文字列

s = “abc”
t = “hoge”

s + t
=> “abchoge”

s => “abc”
t => “hoge”

メソッド“<<”: 末尾への破壊的要素追加

末尾に新しい要素を追加する。元の配列や文字列自体が変更される。

配列

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

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

文字列

```
s << “hoge”
=> “abchoge”

s
=> “abchoge”
```

元の値を直接書き換える操作を**破壊的操作**という。

concat: 配列の破壊的結合

引数に指定した配列を元の配列の末尾に繋げる。

```
y = [9, 8, 7]
x.concat(y)
=> [3, 4, 1, 2, 9, 8, 7]
```

xの値も変わる。

```
x
⇒[3, 4, 1, 2, 9, 8, 7]
```

yの値は変わらない。

```
y
⇒[9, 8, 7]
```

each: 配列/ハッシュの各要素に対するブロックの評価

配列/ハッシュの各要素に対して後続するブロックを繰り返し評価する。

配列

要素を受ける1つのブロック変数で繰り返す。

```
x = [3, 4, 1, 2, 5]
x.each do |i|
  #i= 3, 4, 1, 2, 5で5回繰り返す
end
```

forループで書き換えられる。

```
for i in x
  #i= 3, 4, 1, 2, 5で5回繰り返す
end
```

ハッシュ

keyとvalueを受ける2つのブロック変数で繰り返す。

```
h = {“りんご” => 150, “みかん” => 30}
h.each do |item, price|
  #「item = “りんご”, price = 150」と「item = “みかん”, price = 30」の組で2回繰り返す
end
```

forループで書き換えられる。

```
for item, price in h
  #「item = “りんご”, price = 150」と「item = “みかん”, price = 30」の組で2回繰り返す
end
```

[N]: N番目の取り出し

配列/文字列の添字Nの位置の値を取り出す。

配列

```
x = [3, 4, 1, 2, 5]
x[0]
⇒3
x[-2]
⇒2
x[99]
⇒nil
```

文字列

```
s = “hoge”
s[0]
⇒“h”
s[-2]
⇒“g”
s[99]
⇒nil
```

[start..last], [start, len]: 部分の取り出し

配列/文字列の部分集合を得る。開始添字位置 (start) と終了位置 (last)または長さ (len)で指定する。

配列

z = [1, 3, 5, 6, 2, 10, 3, 5]
z[2..5]
⇒[5, 6, 2, 10]
z[2..−1]
⇒[5, 6, 2, 10, 3, 5]
z[−2..−1]
⇒[3, 5]
z[0, 7]
⇒[1, 3, 5, 6, 2, 10, 3]

文字列

u = “C1254320”
u[2..5]
⇒“2543”
u[2..−1]
⇒“254320”
u[−2..−1]
⇒“20”
u[0, 7]
⇒“C125432”

index(val): 指定要素の発見

配列/文字列から特定の値を探し位置を返す。

配列

配列の中に指定した値valに等しいものがあるか調べ、最初に見つかった位置（添字）を返す。見つからなかった場合はnilを返す。

```
x = [3, 1, 2, 4]
x.index(3)
⇒ 0
x.index(4)
⇒ 3
x.index(5)
⇒ nil
```

文字列

文字列の中に引数に指定した部分文字列valに等しいものがあるか調べ、最初に見つかった位置（添字）を返す。見つからなかった場合はnilを返す。

```
m = “koekitaro@e.koeki-u.ac.jp”
m.index(“@”)
⇒ 9
m.index(“koeki-u.ac.jp”)
⇒ 12
m.index(“f.koeki-u.ac.jp”)
⇒ nil
```

length, size: 配列/文字列の長さ(要素数)を返す

配列

配列の要素数を返す。

```
x = [3, 1, 2, 4]
x.size
⇒ 4
x.length
⇒ 4
```

文字列

文字列の長さを返す（バイト数ではない）。バイト数はbytesizeメソッドで得られる。

```
“abcde”.length
⇒ 5
“あいうえお”.length
⇒ 5
“あいうえお”.bytesize
⇒ 15 (UTF-8の場合)
```

reverse, reverse!: 逆順にする

配列/文字列の中身を全て逆順に並べたものを返す。

配列

```
x = [3, 1, 2, 4]
x.reverse
⇒ [4, 2, 1, 3]
```

xの値は変わらない。

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

reverse!は破壊的に変更する。

```
x.reverse!
⇒ [4, 2, 1, 3]
x => [4, 2, 1, 3]
```

文字列

```
s = “Hello”
s.reverse
⇒ “olleH”
```

xの値は変わらない。

```
s => “Hello”
```

reverse!は破壊的に変更する。

```
s.reverse!
⇒ “olleH”
s => “olleH”
```

配列処理メソッド

1. 集合的な処理を行うメソッド
2. **配列用のメソッド**
3. 文字列固有のメソッド

join (sep): 配列を結合して文字列化

配列の各要素の間に文字列sepを挟んで連結した文字列を返す。sepを省略した場合は、配列の要素をそのまま文字列化する。

```
x = [3, 1, 2, 4]
x.join("/")
⇒ "3/1/2/4"
```

```
x.join(", ")
⇒ "3, 1, 2, 4"
x.join()
⇒ "3124"
```

shift: 先頭要素の取り出し

配列の先頭要素を取り出し、その値を返す。配列は一つずつ前に詰められる。

```
x = [3, 1, 2, 4]
x.shift
⇒ 3
x => [1, 2, 4]
```

shift: 先頭要素の取り出し

これを繰り返すと、配列の要素の先頭から取り出すループが、

```
while i = x.shift
  printf(“%d\n”, i)
end
```

のように作成でき、以下の出力が得られる。

```
3
1
2
4
```

ループから抜けたときのxの値は[]となる。

unshift(val): 先頭への要素の追加

配列の先頭にvalを追加し、その配列を返す。各要素は一つずつ後にずれる。

```
x = [3, 1, 2, 4]
x.unshift(20)
⇒ [20, 3, 1, 2, 4]
x
⇒ [20, 3, 1, 2, 4]
```

uniq: 重複要素の削除

配列から重複した要素を取り除き、取り除いた部分を前に詰めた配列を返す。

```
z = [1, 0, 2, 0, 3, 9, 7, 7, 1, 3, 2, 6, 5, 2]
z.uniq
⇒ [1, 0, 2, 3, 9, 7, 6, 5]
z
⇒ [1, 0, 2, 0, 3, 9, 7, 7, 1, 3, 2, 6, 5, 2]
```

uniq!: 重複要素の削除

元の配列を破壊的に書き換えて、重複したものを取り除いた配列を返す。

```
z = [1, 0, 2, 0, 3, 9, 7, 7, 1, 3, 2, 6, 5, 2]
z.uniq!
⇒ [1, 0, 2, 3, 9, 7, 6, 5]
z
⇒ [1, 0, 2, 3, 9, 7, 6, 5]
```

select{ブロック}: 条件を満たす要素の選択

配列の要素を一つずつ取り出しブロックに渡し、ブロックが真を返す要素のみからなる新規配列を返す。

```
z = [1, 0, 2, 3, 9, 7, 6, 5]
even = z.select {|i| i%2 == 0}
⇒ [0, 2, 6]
```

select{ブロック}: 条件を満たす要素の選択

配列の要素を一つずつ取り出しブロックに渡し、ブロックが真を返す要素のみからなる新規配列を返す。

```
f = [“柿”, “りんご”, “サクランボ”, “桃”, “いちご”]
hira = f.select {|el| /[ぁ-ん]/ =~ el}
⇒ [“りんご”, “いちご”]
```

reject{ブロック}: 条件を満たす要素の削除

selectメソッドの逆。

```
f = [“柿”, “りんご”, “サクランボ”, “桃”, “いちご”]
f.reject {|i| /[あ-ん]/ =~ i}
⇒ [“柿”, “サクランボ”, “桃”]
f
⇒ [“柿”, “りんご”, “サクランボ”, “桃”, “いちご”]
```

reject!{ブロック}: 条件を満たす要素の削除

selectメソッドの逆。破壊的メソッド

```
f = [“柿”, “りんご”, “サクランボ”, “桃”, “いちご”]
f.reject! {|i| /[あ-ん]/ =~ i}
⇒ [“柿”, “サクランボ”, “桃”]
f
⇒ [“柿”, “サクランボ”, “桃”]
```

sort {ブロック}: 配列の内容のソート

配列の内容を並べ換える。ブロックを指定しない場合は各要素を演算子<=>で比較して小さい順（昇順）に並べ換える。

```
z = x.sort { |x, y| x <=> y}
```

大きい順（降順）に並べ換えたい場合

```
z = x.sort { |x, y| y <=> x}
```

sort_by{ブロック}: 配列の内容の効率的ソート

ソート基準とする値を取り出す式のみ指定。ハッシュのvalueの大小関係でソートしたい時に有用

商品名vs.単価リストを持つハッシュの商品名を単価の小さい（安い）順に並べ換える場合：

```
item = {
  “アルカリ異音水” => 150,
  “おでんつゆ” => 50,
  “マグロの煮こごり” => 500
}

item.keys.sort_by{ |x| item[x]}
```

配列処理メソッド

1. 集合的な処理を行うメソッド
2. 配列用のメソッド
3. 文字列固有のメソッド

chop, chop!: 文字列の末尾削除

元の文字列の最後の一文字を取り除いたものを返す。

“abc\n”.chop
⇒ “abc”
“abc\n”.chop.chop
⇒ “ab”

chomp(rs), chomp!(rs): 文字列の末尾削除 (条件付)

文字列末尾から調べてrsという文字列があればそれを取り除く。rsを指定しない場合は改行文字が取り除かれる。

```
x = “abc\n”
x.chomp
⇒ “abc”
```

downcase, downcase!: 文字列の小文字化

```
x = “Hello, World!!”
x.downcase
⇒ “hello, world!!”
```

upcase, upcase!: 文字列の大文字化

```
x = “Hello, World!!”
x.upcase
⇒ “HELLO, WORLD!!”
```

capitalize, capitalize!: 先頭文字の大文字化

```
x = “Hello, World!!”
x.capitalize
⇒ “Hello, world!!”
```

gsub(pattern){replace}, gsub!(pattern){re[lace}: 文字列の置き換え

元の文字列のうちpatternにマッチするものを全てreplaceに置き換え、その結果の文字列を返す。

```
x = “This is a pen”
y = x.gsub(/is/){“IS”}
y
⇒ “ThIS IS a pen”
```

gsub(pattern){replace},
gsub!(pattern){replace}: 文字列の置き換え

パターンを指定して、マッチするもの全てを置換できる。

x = “This is a pen. That is a pencil.”

x.gsub(/is/){“IS”}

⇒ “ThIS IS a pen. That IS a pencil.”

x.gsub(/\bis\b/){“IS”} #\bは単語の境界を意味

⇒ “This IS a pen. That IS a pencil.”

sub(pattern){replace},
sub!(pattern){replace}: 文字列の置き換え

最初の一つだけを置き換える。

x = “This is a pen. That is a pencil.”
x.sub(/is/){“IS”}
⇒ “ThIS is a pen. That is a pencil.”

split(sep): 文字列の分割

元の文字列を、指定したパターンsepを区切りと見なして分割し、分割された文字列を配列に格納して返す。sepを省略した時は空白を区切りとみなす。

各項目が「カンマ(,)スペース(が0個以上)」で区切られたデータの場合：

中町太郎, 90,　70,20

正規表現は**/,\s*/**なので、

“中町太郎, 90,　70,20”.split(/,\s*/)
⇒ [“中町太郎”, “90”, “70”, “20”]

今回学ぶこと

配列処理メソッド

様々な条件での並べ替え

データの格納と検索・絞り込み

一般的なソート（昇順ソート）

```
x = [9, 20, 10, 15, 8]
y = x.sort
y
⇒ [8, 9, 10, 15, 20]
```

一般的なソート（昇順ソート）

```
x = [“orange”, “apple”, “strawberry”, “pear”]
y = x.sort
y
⇒ [“apple”, “orange”, “pear”, “strawberry”]
```

一般的なソート（降順ソート）

```
x = [9, 20, 10, 15, 8]
y = x.sort.reverse
y
⇒ [20, 15, 10, 9, 8]
```

一般的なソート（降順ソート）

```
x = [“orange”, “apple”, “strawberry”, “pear”]
y = x.sort.reverse
y
⇒ [“strawberry”, “pear”, “orange”, “apple”]
```

独自基準のソート

以下のCSVデータ (ja-math.csv) の要素を国語の点が高い順に並べ換える。

```
氏名,国語,数学
山田太郎,50,40
中町太郎,90,70
飯森花子,91,90
鶴岡一人,60,50
酒田三吉,52,80
三川一二三,12,96
```

国語の点が高い順に並べ替え

このCSVファイルを

```
score = CSV.read(“ja-math.csv”, headers:true)
```

として読み込むと、

国語の点が高い順に並べ替え

概念的に以下のような構造で読み込まれる（実際には
CSV::Tableの値だが、ハッシュの配列と同じ性質を持つ）。

```
score = [
  {“氏名” => “山田太郎”, “国語” => 50, “数学” => 40},
  {“氏名” => “中町太郎”, “国語” => 90, “数学” => 70},
  {“氏名” => “飯森花子”, “国語” => 91, “数学” => 90},
  {“氏名” => “鶴岡一人”, “国語” => 60, “数学” => 50},
  {“氏名” => “酒田三吉”, “国語” => 52, “数学” => 80},
  {“氏名” => “三川一二三”, “国語” => 12, “数学” => 96},
]
score.length
=> 6
```

国語の点が高い順に並べ替え

国語の点数を取り出すには、

```
x[“国語”].to_i
```

とすれば良いので、

```
score.sort_by{|x| x[“国語”].to_i}
```

で国語の点数で昇順にした配列が得られる。

国語の点が高い順に並べ替え

これをreverseメソッドに渡すと降順の結果が得られる。

```
score.sort_by{|x| x[“国語”].to_i}.reverse
=>
[ #<CSV::Row “氏名” : “飯森花子” “国語”: “91” “数学”: “90”>,
  #<CSV::Row “氏名” : “中町太郎” “国語”: “90” “数学”: “70”>,
  #<CSV::Row “氏名” : “鶴岡一人” “国語”: “60” “数学”: “50”>,
  #<CSV::Row “氏名” : “酒田三吉” “国語”: “52” “数学”: “80”>,
  #<CSV::Row “氏名” : “山田太郎” “国語”: “50” “数学”: “40”>,
  #<CSV::Row “氏名” : “三川一二三” “国語”: “12” “数学”: “96”>]
```

国語の点が高い順に並べ替え

このソート結果を順序よく出力するには、eachで1つずつ取り出して処理する。

```
score.sort_by{|x| x[“国語”].to_i}.reverse.each do |i|
  ……出力処理……
end
```

今回学ぶこと

配列処理メソッド
様々な条件での並べ替え
データの格納と検索・絞り込み

シラバスの表現例

データベース

値

ハッシュ

科目名	基礎プログラミング
科目コード	154
担当教員	鳥海三輔
開講時期	春学期
概要	プログラミングを通じて問題解決能力の向上を図る。

ハッシュ

科目名	応用プログラミング
科目コード	254
担当教員	飯森大和
開講時期	秋学期
概要	C言語を通じて問題解決に最適なアルゴリズムを適用する方策を学ぶ。

ハッシュ

科目名	応用もっけ論
科目コード	39
担当教員	酒田育造
開講時期	春学期
概要	その地域の「もっけ」を理解し、深い交流のきっかけとする。

数値項目の処理

CSV.readでは、各項目を**文字列**として代入する。CSVデータの項目中に数値がある場合は、**数値に変換する処理**が必要。

文字列を数値に変換する方法

- 要素を取り出してから数値に変換(to_i, to_f)
- CSV読み取り時に数値変換

要素取り出してから数値化

```
csv[“科目コード”].to_i
```

CSV読み取り時に数値変換

```
csv = CSV.read(“syllabus.csv”, headers: true,
converters: :numeric)
```

まとめ

- Rubyでデータの集合的な処理を行うには配列処理メソッドを用いる。
- データを並べ替えるにはsort (昇順)、reverse (降順)を用いる。
- CSV.readでは各項目を文字列として代入するので、項目中に数値がある場合は、数値に変換する必要がある。