Ruby 初級者向けレッスン 38 回
— イテレータを使う —

こなみひでお@Ruby 関西

2010 年 7 月 24 日

1　概要

定義っぽいもの

イテレータ:　繰り返し処理のためのブロックを伴うメソッド
　イテレータの他にもブロック付きメソッドをもつクラスはいくつもある (IO，File など)．つまり，イテレータはブロック付きメソッドの一部．

レシーバは：　Enumerable クラスのインスタンス．あるいは・・・・

Enumerable(数え上げ可能) なデータって何？

厳密には，Enumerable モジュールを mix-in したクラスのインスタンス，
具体的には，**Array**, **Hash**, **Range**,
Dir，**String**，**Integer** などのオブジェクトもラッパークラス Enumerable::Enumerator クラス[*1]で Enumerable の機能が与えられて，各種イテレータが使える．

[*1] Ruby 1.8.8 以降では Enumerator クラス．

2　いろいろなイテレータ

もっともよく使う each,map,inject

■each　レシーバから要素を 1 個ずつ取り出して，ブロック変数に渡して処理する．
Hash オブジェクトに対してはどのように振る舞うのか，ためしてみましょう．

■map　配列の要素を変換した新しい配列を返します． map! は破壊的なメソッドです．

■inject　リストのたたみこみ演算を行います。

```
[2, 3, 4, 5].inject {|result, item| result + item }

[2, 3, 4, 5].inject(0) {|result, item| result + item**2 }

[1, 2, 3, 4, 5].inject(:+)
```

どうして for ループを使わないの？

for ループは他の多くの言語で使われているので，「いちおう」 Ruby にもあったほうがいいという程度の存在[*2]．

配列をレシーバとするメソッド一覧

配列 [3,1,4,1,5,9] をレシーバとするメソッドの使用例[*3]

```
x.all? {|e| e > 0}      # => true  (要素がすべて正の数か？)
x.any? {|e| e > 7}      # => true  (7 より大きな要素があるか？)
x.map {|e| "%02d" % e}  # => ["03", "01", "04", "01", "05", "09"]
y = Hash.new{|h, k| h[k] = []}; x.each_with_index {|e, i| y[e] << i}; y
  # => {1=>[1, 3], 3=>[0], 4=>[2], 5=>[4], 9=>[5]}
x.find {|e| e > 3}      # => 4  (3 より大きな最初の要素)
```

[*2] for は Fortran, BASIC, C, AWK ,Perl, Postscript には装備されています．つまり小波が使って来た言語のすべてに for があるというわけ．LISP とかあまり知らないけど，ないよね？

[*3] http://www.okisoft.co.jp/esc/ruby/tut-06.html 『Ruby チュートリアル – 6. イテレータ』(沖ソフト)

```
x.select {|e| e > 3}    # => [4, 5, 9]  (3 より大きな要素を選択)
x.grep(Integer)         # => [3, 1, 4, 1, 5, 9]  (Integer === e の要素)
x.inject(1) {|a, b| a * b}
  # => 540  (要素の積 (((((1 * 3) * 1) * 4) * 1) * 5) * 9)
x.member?(4)            # => true  (4 を要素とするか？)
x.max                   # => 9 (最大値)
x.max {|a, b| b <=> a}  # => 1 (大小逆に比較した最大値)
x.min                   # => 1 (最小値)
x.min {|a, b| b <=> a}  # => 9 (大小逆に比較した最小値)
x.partition {|e| e > 3} # => [[4, 5, 9], [3, 1, 1]]  (真の群と偽の群)
x.reject {|e| e > 3}    # => [3, 1, 1]  (3 より大きな要素を排除)
x.sort                  # => [1, 1, 3, 4, 5, 9]
x.sort {|a, b| b <=> a} # => [9, 5, 4, 3, 1, 1]  (大小逆にソート)
x.sort_by {|e| e % 3}   # => [3, 9, 1, 4, 1, 5]  (3 の剰余系でソート)
x.to_a                  # => [3, 1, 4, 1, 5, 9]  (to array)
x.zip(('a'..'z').to_a)
  # => [[3, "a"], [1, "b"], [4, "c"], [1, "d"], [5, "e"], [9, "f"]]
y={}; x.zip(('a'..'z').to_a) {|d, c| y[c] = d}; y
  # => {"a"=>3, "b"=>1, "c"=>4, "d"=>1, "e"=>5, "f"=>9}
```

整数をレシーバとするイテレータ

```
2.upto(10){|e| p e}
10.downto(2){|e| p e}
10.times{|e| p e}
```

times は決まった回数の繰り返しを行うのに便利です.

文字列をレシーバとするイテレータ

```
str = "Abcd \nqrt"
jstr = "日本語の String "

str.each_byte { |e| p e}
jstr.each_byte { |e| p e}

str.each_char { |e| p e}
```

```
jstr.each_char { |e| p e}

str.each_line { |e| p e}
jstr.each_line { |e| p e}

# each は obsolete.  1.8.x では each_line のように振る舞う.
str.each { |e| p e}
jstr.each { |e| p e}
```

練習 2–1　正の整数 n の階乗 $n!$ を求める方法を何通りでも考えて下さい[*4].

練習 2–2　文字列 "**零 一 二 三 四 五 六 七 八 九 十 十一 十二 十三 十四 十五**" があります．これから次のような日本語の漢数字の読みを表すハッシュテーブルを作ってください．ただし，ハッシュは対応関係さえあっていればよいので，順序はそろっている必要はありません．

```
{0 => "零", 1 => "一", ...}
```

練習 2–3　シンボルの配列 syms があります．

```
syms = [:tokyo, :kyoto, :osaka, :kobe, :matsue, :sendai]
```

これを次のように，先頭だけ大文字化され，昇順にソートされた文字列の配列に変えて下さい．

```
["Kobe", "Kyoto", "Matsue", "Osaka", "Sendai", "Tokyo"]
```

練習 2–4　正の整数 n に対して，次の多重積を n の二重階乗といい， $n!!$ で表します．

$$\begin{aligned} n!! &= n \cdot (n-2) \cdots 3 \cdot 1 \,(n \text{ は奇数}) \\ &= n \cdot (n-2) \cdots 4 \cdot 2 \,(n \text{ は偶数}) \end{aligned} \tag{1}$$

$n!!$ を求めるメソッドを書いて下さい．

[*4] これはプログラミングの教科書では再帰的なアルゴリズムの定番ですが，実際には階乗を再帰で求めるのはぜんぜん実用的じゃありませんよね．

練習 2–5　n までの整数からすべての合成数を引き去って素数だけを残すやりかたとして,「エラトステネスのふるい」というアルゴリズムが有名です．このアルゴリズムを，イテレータを使ってうまく実装してください．

練習 2–6　次のメソッドは二項分布を生成します．これを改良してください．ちなみに二項分布は次のように試行回数 n と 確率 p をパラメータとする確率分布です．ここでは，二項係数 ${}_nC_x$ を計算するのにパスカルの三角形を構成して使っています．

$$f(x) =_n C_x p^x (1-p)^{n-x},\ {}_nC_x = \frac{n!}{x!(n-x)!}$$

```
# generate an array of accumulated binary distribution
# parameter = n,p
def binary_dist(n,p)
  ar = []
  v = [1]
  n.times do
    v2 = v.dup
    for i in 1 ... v.size
      v[i] += v2[i-1]
    end
    v<< 1
  end
  for i in 0 .. n
    ar << v[i] * p ** i * (1-p) ** (n-i)
  end
  ar
end
```