基礎プログラミング+演習 ＃5 – 2次元配列+レコード+画像

久野 靖 (電気通信大学)

2017.12.11

今回の主な内容は次の通りです。

- 2次元配列・レコード型と画像の表現
- さまざまな形の描画

ここまでは数値の題材ばかりでしたが、コンピュータでは画像や音など任意のディジタル情報が扱えます。ここでは画像を通じて「自分が構想したものを作る」という経験を持って頂きます。

1 前回演習問題の解説

1.1 演習1 — 合計

一通り作りました (短いメソッドは1行に書いています)。引き算は簡単ですが、少しひねって負の数加えました。これまでの数値を覚えるためには$list という配列を用意し、数値をこの後ろに追加して覚えて行きます。reset の時は現在の値とこの$list を別の変数に退避しておき、undo では退避しておいたものを元に戻します。

```
$x = 0; $sx = 0; $list = []; $slist = [];
def sum(v) $x = $x + v; $list.push(v); return $x end
def dec(v) sum(-v) end
def list() p($list, $x) end
def reset() $sx = $x; $x = 0; $slist = $list; $list = [] end
def undo() $x,$sx = $sx,$x; $list,$slist = $slist,$list end
```

実行例を示します。

```
irb> sum 1
  => 1
irb> sum 2.5
  => 3.5       ←合計 3.5
irb> dec 1.2   ← 1.2 を引く
  => 2.3
irb> list      ←履歴表示
[1, 2.5, -1.2] ←履歴
2.3            ←現在値
  => nil
irb> reset     ←リセット
  => []
irb> sum 1
  => 1         ←また 0 からの和
irb> undo      ←リセットを戻す
  => [[1, 2.5, -1.2], [1]]
```

undo したときは過去の結果/リストと現在のものを交換するので、2回 undo すると元に戻ります。

1.2 演習 2 — RPN 電卓

これも一通り作りました。演算は add と同様で計算だけ変えます。交換は 2 つの値を取り出して逆に入れます。演算内容を覚えるために s というメソッドを用意し、この中で渡された値を文字列に変換して広域変数$str の後ろに連結して行きます。show はこの変数の内容を打ち出せばよいだけです (ついでに演算結果も打ち出します)。

```
$vals = []; $str = ''
def clear() $vals = []; $str = '' end
def s(x) $str = $str + ' ' + x.to_s end
def show() p($str, $vals[$vals.length-1]) end
def e(x) $vals.push(x); s(x); return $vals end
def add
  x = $vals.pop; $vals.push($vals.pop + x); s('+')
  return $vals
end
def sub
  x = $vals.pop; $vals.push($vals.pop - x); s('-')
  return $vals
end

def mul
  x = $vals.pop; $vals.push($vals.pop * x); s('*')
  return $vals
end
def div
  x = $vals.pop; $vals.push($vals.pop / x); s('/')
  return $vals
end
def exch
  x = $vals.pop; y = $vals.pop; s('x')
  $vals.push(x); $vals.push(y); return $vals
end
```

実行のようすを示します。

```
irb> e 1
  => [1]
irb> e 2
  => [1, 2]
irb> e 3
  => [1, 2, 3]  ← 1、2、3 を入れたところ
irb> mul        ←掛けたら 6
  => [1, 6]
irb> add        ←足したら 7
  => [7]
irb> show
" 1 2 3 * +"    ←履歴表示
7
```

```
  => nil
irb> e 4          ←さらに 7 を入れ
  => [7, 4]
irb> exch         ←交換
  => [4, 7]
irb> sub          ←引き算
  => [-3]
```

作ってみると、スタックを使った計算のようすがよく分かると思います。

1.3 演習 3 — 行列電卓

2 × 2 行列の電卓ですが、加減算は要素ごとに演算すればよいので簡単ですね。乗算とか逆行列とかはちょっとごちゃごちゃしますが、まあこれらも 2 × 2 であればひたすら書けばできるでしょう。

```
$vals = []
def e(m) $vals.push(m) end
def add
  m = $vals.pop; n = $vals.pop
  $vals.push([[n[0][0]+m[0][0], n[0][1]+m[0][1]],
              [n[1][0]+m[1][0], n[1][1]+m[1][1]]])
  return $vals
end
def sub
  m = $vals.pop; n = $vals.pop
  $vals.push([[n[0][0]-m[0][0], n[0][1]-m[0][1]],
              [n[1][0]-m[1][0], n[1][1]-m[1][1]]])
  return $vals
end
def mul
  m = $vals.pop; n = $vals.pop
  $vals.push([[n[0][0]*m[0][0] + n[0][1]*m[1][0],
               n[0][0]*m[0][1] + n[0][1]*m[1][1]],
              [n[1][0]*m[0][0] + n[1][1]*m[1][0],
               n[1][0]*m[0][1] + n[1][1]*m[1][1]]])
  return $vals
end
def trans
  m = $vals.pop;
  $vals.push([[m[0][0], m[1][0]],
              [m[0][1], m[1][1]]])
  return $vals
end
def inv
  m = $vals.pop
  d = (m[0][0]*m[1][1] - m[0][1]*m[1][0]).to_f
  $vals.push([[m[1][1]/d, -m[0][1]/d],
              [-m[1][0]/d, m[0][0]/d]])
  return $vals
end
```

3 × 3 以上になると、直接書くのではなくループを使った方が楽になりますが、そのあたりはこの科目では含みません (いずれどこかで習うと思いますが)。実行例をみてみましょう。

```
irb> e [[1, 2], [3, 4]]
  => [[[1, 2], [3, 4]]]
irb> e [[1, 1], [1, 1]]
  => [[[1, 2], [3, 4]], [[1, 1], [1, 1]]]
irb> sub
  => [[[0, 1], [2, 3]]]
```

引き算とかは問題ないですね。逆行列はどうでしょうか。

```
irb> e [[2, 1], [1, -1]]
  => [[[2, 1], [1, -1]]]
irb> inv
  => [[[0.3333333, 0.3333333], [0.3333333, -0.6666667]]]
irb> e [[1, 0], [5, 0]]
  => [[[0.3333333, 0.3333333], [0.3333333, -0.6666667]], [[1, 0], [5, 0]]]
irb> mul
  => [[[2.0, 0.0], [-3.0, 0.0]]]
```

これは何を計算しているかというと、次の連立方程式を解いています。

$$\begin{cases} 2x & + & y & = & 1 \\ x & - & y & = & 5 \end{cases}$$

これを行列の形に書くと次のようになります。

$$\begin{pmatrix} 2 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} x & 0 \\ y & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 5 & 0 \end{pmatrix}$$

係数行列 A =`[[2, 1], [1, -1]]` の逆行列を A^{-1} を上式両辺に左から掛けます。

$$A^{-1}\, A \begin{pmatrix} x & 0 \\ y & 0 \end{pmatrix} = A^{-1} \begin{pmatrix} 1 & 0 \\ 5 & 0 \end{pmatrix}$$

$A^{-1}A$ は単位行列なので消えますから、つまり右辺を計算すると x と y が求まるわけです。実際、$x = 2$、$y = -3$ を代入してみると元の連立方程式が成り立っていることが確認できます。

1.4 演習 4 — 再帰関数

これらは定義のとおり再帰関数にすればできるので、まずはコードを示します。

```
def fact(n)
  if n == 0 then return 1
  else           return n * fact(n-1)
  end
end
def fib(n)
  if n < 2 then return 1
  else          return fib(n-1) + fib(n-2)
  end
end
```

```
def comb(n, r)
  if r == 0 || r == n then return 1
  else                     return comb(n-1, r) + comb(n-1, r-1)
  end
end
def binary(n)
  if n == 0 then        return "0"
  elsif n == 1 then     return "1"
  elsif n % 2 == 0 then return binary(n / 2) + "0"
  else                  return binary((n-1) / 2) + "1"
  end
end
```

実行例も一応示しておきます。

```
irb> fact 4
  => 24
irb> fact 5
  => 120
irb> fib 2
  => 2
irb> fib 3
  => 3
irb> fib 4
  => 5
irb> comb 5, 2
  => 10
irb> comb 6, 2
  => 15
irb> binary 5
  => "101"
irb> binary 7
  => "111"
irb> binary 8
  => "1000"
```

1.5 演習 5 — 順列

問題のヒントに書いたように、配列から 1 つずつ要素を取って出力用の列に入れますが、その際取った要素の位置に nil を入れることで重複して取らないようにします。呼び出し方を覚えなくて済むように perm には配列だけ渡し、そこから再帰用メソッド perm1 を「残った長さ、元の配列、空の配列」をパラメタとして呼びます。

```
def perm(a) perm1(a.length, a, []) end
def perm1(n, a, b)
  if n == 0 then p(b); return end
  a.each_index do |i|
    if a[i] == nil then next end
    b.push(a[i]); a[i] = nil; perm1(n-1, a, b); a[i] = b.pop
```

```
    end
  end
```

perm1 では、残った長さが 0 なら出力して終わります。そうでない場合は a の各要素を順番に見ますが、その際入っていたのが nil なら「ループの次に進み」ます (next の機能)。そうでない場合は、配列 b にその要素を追加し、配列 a のその位置に nil を入れて自分自身を呼びます (もちろん n は 1 減らす)。戻ってきたら b の最後を取り除いてそれを a[i] に戻します。このように、それぞれが自分が変更したものを元に戻すことで、全体としてうまく動きます。実行例は次の通り。

```
irb> perm [1, 2, 3]
[1, 2, 3]
[1, 3, 2]
[2, 1, 3]
[2, 3, 1]
[3, 1, 2]
[3, 2, 1]
 => [1, 2, 3]
irb>
```

2　2 次元配列と画像の表現

2.1　2 次元配列の生成 exam

2 次元配列 (配列の配列) を用いた前回の演習問題では、直接すべての値を「[[a, b], [c, d]]」のように指定することで 2 次元配列を生成していました。しかしこの方法は大きさが大きくなるととても大変です。1 次元 (1 列) の配列を作る時に、ブロックを使って初期値を設定する方法がありました。

```
a = Array.new(100) do |i| 2*i end
```

これを応用することで大きな 2 次元配列を作ることができます。つまり、ブロックの中にさらに Array.new(...) を入れれば、「配列が並んだ配列」つまり 2 次元配列ができるからです。

```
a = Array.new(10) do Array.new(10, 1) end
  # 10 × 10 ですべて「1」の行列
a = Array.new(10) do |i| Array.new(10) do |j| i*j end end
  # 「九九の表」
```

「2 次元配列」は実際には図 1 のように配列の各要素が配列という構造です。でも普段は「縦横 2 次元に要素が並んでいる」とイメージして問題ありません。

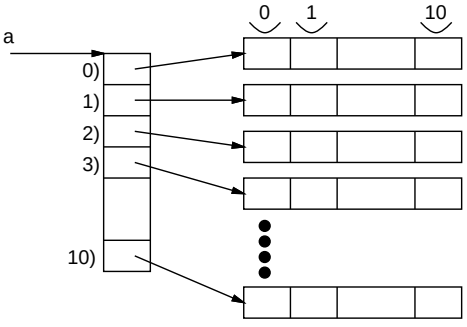


図 1: 2 次元配列

演習 1 5×5の2次元配列で、次の図の (a)〜(d) のような内容のものをブロックつきの `Array.new()` を使って生成してみなさい。

(a)

0	1	2	3	4
-1	0	1	2	3
-2	-1	0	1	2
-3	-2	-1	0	1
-4	-3	-2	-1	0

(b)

1	0	0	0	0
1	1	1	1	1
1	2	4	8	16
1	3	9	27	81
1	4	16	64	256

(c)

1	0	0	0	0
0	1	0	0	0
0	0	1	0	0
0	0	0	1	0
0	0	0	0	1

(d)

1	0	0	0	1
0	1	0	1	0
0	0	1	0	0
0	1	0	1	0
1	0	0	0	1

なお、2次元配列を irb で表示するときは「pp」というメソッドを使うと見やすくできます。ただしこれを使うには「`require 'pp'`」というおまじないを実行させておく必要があります。

```
irb> require 'pp'    ← pp を使うための準備
  => true
irb> a = Array.new(5) do |i| Array.new(5) do |j| i*j end end
  => [[0, 0, 0, 0, 0], [0, 1, 2, 3, 4], [0, 2, 4, 6, 8],
   [0, 3, 6, 9, 12], [0, 4, 8, 12, 16]]  ←見やすくない
irb> pp a            ← pp を使うと
[[0, 0, 0, 0, 0],    ←そろえてくれる
 [0, 1, 2, 3, 4],
 [0, 2, 4, 6, 8],
 [0, 3, 6, 9, 12],
 [0, 4, 8, 12, 16]]
  => nil
```

2.2 レコード型の利用 exam

配列が「同じ型 (種類) の値の並びで、添字 (番号) により要素を指定する」のに対し、**レコード**は「様々な型 (種類) の複数の値が並んだもので、どの値 (フィールド) かは名前で指定する」ものです。Ruby ではレコード型はまず `Struct.new` によりレコードクラスを定義し、その後レコードクラスを使って個々のレコード (データ) を作ります。具体的には、レコードクラスの定義は次のようにします (レコード名は大文字で始まる必要があります)。

```
レコード名 = Struct.new(:名前, :名前, ...)
```

ここで「:名前」は**記号型** (symbol type) の定数で、これによりフィールドの名前が指定できます。個々のレコードを作るのは次によります。

```
p = レコード名.new(値, 値, ...)
```

これによりレコード型の値が作られ、指定した値が各フィールドの初期値になります (順番はレコード定義の時に指定した順になります)。上の例ではそのレコードを変数 p に入れています。

コンピュータ上では画像は**ピクセル** (pixel、画面上の小さな点) の集まりとして扱い、各ピクセルの色は赤 (R)、緑 (G)、青 (B) の強さを 0〜255 の範囲の整数で表すことが普通です。ピクセルの情報をレコードとして定義し、それを用いてピクセルを生成します。

```
Pixel = Struct.new(:r, :g, :b)
p = Pixel.new(255, 255, 255)  # RGB とも 255 の値
```

Ruby では配列と同様、レコードも new を使って作り出す必要があります。作り出したあとは、「`p.r`」「`p.g`」「`p.b`」等、変数名の後にフィールド名を加えたものが通常の変数と同様に使えます。配列と似ていますが、レコードの場合はフィールドは「名前」である点が違います。

2.3 ピクセルの 2 次元配列による画像の表現 exam

さて、ピクセル 1 個の説明が終わったので、今度はこれを「2 次元に (縦横に) 並べて」画像を作ることを考えます (図 2 左)。これを Ruby で表現する場合、各 Pixel を上で説明したように Ruby のレコード型で表現し、それを縦横に並べるわけです。たとえば縦方向 (高さ) が 200 ピクセル、横方向 (幅) が 300 ピクセルの画像を作るとします。そのためには、先に学んだ 2 次元配列の初期化のとき、個々の要素をピクセルにすればよいのです。

```
$img = Array.new(200) do Array.new(300) do Pixel.new(255,255,255) end end
```

これによって作られるデータ構造は図 2 右のようになります。

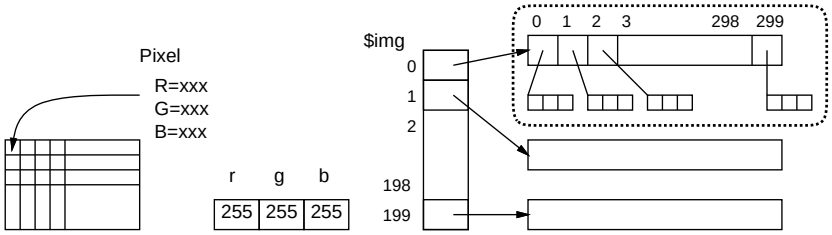


図 2: 画像のデータ構造とレコードの 2 次元配列

2.4 画像中の点の設定と書き出し

先に生成した画像は RGB 値が全て「255」なので「真っ白」です。そこで次に、座標 (x, y) のピクセルを指定した RGB 値に書き換えるメソッドを作ります。

```
def pset(x, y, r, g, b)
  if 0 <= x && x < 300 && 0 <= y && y < 200
    $img[y][x].r = r; $img[y][x].g = g; $img[y][x].b = b
  end
end
```

なぜ if 文があるかというと、X 座標と Y 座標が画像の範囲内 (ここでは 0〜299、0〜199) のときだけ書き込むためです。こうしておくと、呼び出す側で間違って (または簡単のため) 画像の範囲外に書き込もうとしても単に無視できます。

さて次に、こうして画像中に書き込むことができるようになりましたが、画像を実際に「見る」ためには何らかのファイル形式で書き出す必要があります。ここではできるだけ簡単な形式として **PPM 形式**を選び、その形式でファイルに書き出すメソッド `writeimage` を作りました。[1]

```
def writeimage(name)
  open(name, "wb") do |f|
    f.puts("P6\n300 200\n255")
    $img.each do |a| a.each do |p| f.write(p.to_a.pack('ccc')) end end
  end
end
```

メソッドの説明は次の通りです。

- `writeimage` は画像のファイル名 (文字列) を指定して呼び出す。

[1] 普段私たちが Web などで見ている画像形式は GIF、JPEG、PNG などで、ブラウザもこれらの画像を表示するようにできていますが、これらのファイル形式は圧縮などの機能が備わっているため、そんなに簡単なコードで書き出すことができないのです。

- open は指定した名前のファイルをバイナリ (binary) 形式で書き出す (write) 準備をして、その出力チャネル (データの通り道) をブロックに渡して呼び出す。
- ここではブロックでチャネルを f という名前で受け取る。
- まず、ファイルに「P6 300 200 255」と出力する。これは「PPM 画像のカラー形式で、幅 300 ×高さ 200、RGB 値の最大は 255」を表す指定になっている。[2]
- 続いて、画像の 2 次元配列の各行について、さらにその行の中の各ピクセルについて、(1) ピクセルを配列に変換し (p.to_a)、その配列を 3 バイトのバイナリデータに変換し (.pack('ccc'))、ファイルに書き込む (f.write(…))。

2.5 例題: 画像を生成し書き出す

ではいよいよ、画像を作って書き出すメインのメソッドまで含めた全体像を見て頂きましょう。

```
Pixel = Struct.new(:r, :g, :b)
$img = Array.new(200) do Array.new(300) do Pixel.new(255,255,255) end end
def pset(x, y, r, g, b)
  if 0 <= x && x < 300 && 0 <= y && y < 200
    $img[y][x].r = r; $img[y][x].g = g; $img[y][x].b = b
  end
end
def writeimage(name)
  open(name, "wb") do |f|
    f.puts("P6\n300 200\n255")
    $img.each do |a| a.each do |p| f.write(p.to_a.pack('ccc')) end end
  end
end
def mypicture
  pset(100, 80, 255, 0, 0)
  writeimage('t.ppm')
end
```

mypicture がメインになりますが、ここでは (100, 80) の位置に真っ赤な点 (RGB のうち R だけが最大なので) を打ち、t.ppm というファイルに書き出します。実際にこれを動かすには、ターミナルの窓を「2 つ」開いて次のようにします。

- 片方の窓ではこれまで通り irb を動かし、mypicture を実行させる。
- もう片方の窓では、できあがった t.ppm を gimp など表示できるツールを使って表示する (変換ツールで PNG など普通に見られる画像形式にしてもよいです)。

生成された画像を図 3 にお見せします (赤い点が小さすぎてほとんど分からないと思いますが…)。

演習 2 上の例題を打ち込み、そのまま動かしなさい (色の RGB 値は 0〜255 の範囲で適宜変えてみるとよいでしょう)。動いたら、次のように変更してみなさい。

a. 水平または垂直または斜め (右上がり) に線を引くようにしてみる。
b. 長方形または円形を描いてみる (輪郭だけ描くのでも内側を指定した色で塗りつぶすのでもよい)。

[2] 画像ファイルの先頭にはだいたい、このような形で画像の種別やサイズを記述したデータが置かれています。この部分のことを**ヘッダ** (header) などと呼びます。

図 3: 赤い点が 1 個

c. 三角形を描いてみる (塗りつぶすのは多少工夫が必要かと)。

d. その他、好きな図形や模様や色を表現してみる。

`mypicture` の中にコードを追加して `pset` を呼び出すことを想定していますが、場合によってはさらにメソッドを追加する方が作りやすいかも知れません。

先に示した `pset` は「Y 座標が大きいほど下」に点を打ちます。コンピュータ上の画像は伝統的にそうしていますが、皆様は「Y 軸が上向き」に慣れているので、`pset` をそのように直すことを勧めます。

2.6 計算により図形を塗りつぶす exam

先の練習問題はどうでしたか。グラフのように「x を変えながら y=f(x) を計算して `pset(x, y, ...)`」と考える人が多いと思いますが、実はこの方法で輪郭線を描くと細かったり途切れたりしてあんまりよくありません。[3] むしろ図 4 のように「塗りつぶす」方がきれいにできやすいです。

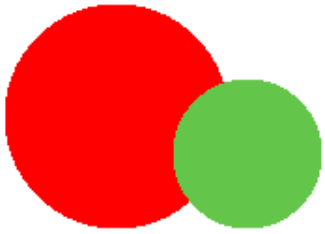


図 4: 2 つの内部まで色を塗った円

このような塗りつぶしをおこなうメソッド `fillcircle` を見てみましょう。

```
def fillcircle(x0, y0, rad, r, g, b)
  200.times do |y|
    300.times do |x|
      if (x-x0)**2 + (y-y0)**2 <= rad**2 then pset(x, y, r, g, b) end
    end
  end
end
```

[3] 途切れないように工夫したとしても、「1 ピクセル幅」というのは今日の画面解像度では「極めて細い」線になりますから。

このメソッド内では、timesの中にさらにtimes、つまりループの中にさらにループがあるので、このようなものを **2 重ループ**と呼びます。この内側での 2 つの変数の進み方は、次のようになります。

```
0,0  0,1  0,2,  0,3  0,4  ....  0,288 0,299
1,0  1,1  1,2,  1,3  1,4  ....  1,288 1,299
2,0  2,1  2,2,  2,3  2,4  ....  2,288 2,299
...
...
198,0  198,1  198,2,  198,3  198,4  ....  198,288 198,299
199,0  199,1  199,2,  199,3  199,4  ....  199,288 199,299
```

縦横に揃えて書いてありますが、要は外側ループで y の値を 0〜199 まで変化させ、**そのそれぞれの値について**内側の x の値を 0〜299 まで変化させます。そして、これで画像上のすべての点 (座標) を洩れなく列挙しているわけです。つまり、ここで列挙される (x, y) の集合は次のようになるわけです (これが画像の全範囲)。

$$\{\ (x, y) \mid 0 \leq x < 300,\ 0 \leq y < 200\ \}$$

円というのは中心 (x_0, y_0) からの距離が rad 以内の点の集合ですから、次のように表せます。

$$\{\ (x, y) \mid |(x, y) - (x_0, y_0)| \leq rad\ \}$$

これをプログラムで扱いやすいように、距離の 2 乗を使う形に直します。

$$\{\ (x, y) \mid (x - x_0)^2 + (y - y_0)^2\ \leq\ rad^2\ \}$$

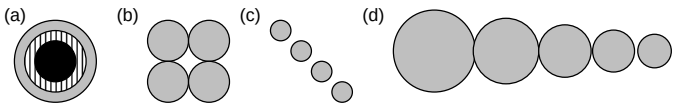
さて、先のコードでは 2 重ループの内側に if 文がありますが、その条件がまさにこの条件であり、従って「円に含まれるすべての点 (x, y) に対して色を設定する (塗る)」ことになるわけです。

実際にはこれを呼び出す必要があるので、円を 2 つ描き、ファイルに画像を書き出すメソッドを `mypicture1` という名前で用意しました (その結果が図 4 なのでした)。

```
def mypicture1
  fillcircle(110, 100, 60, 255, 0, 0)
  fillcircle(180, 120, 40, 100, 200, 80)
  writeimage("t.ppm")
end
```

長方形などさまざまな図形についても、このように「すべての点のうち、図形内部に含まれるという条件を満たす点のみに色を設定する」という形でコードを作ることができます

演習 3 円を塗るメソッドを先のプログラムに追加して動かせ。動いたら、`fillcircle` の呼び出し方を調節して、次の図のように円を配置してみよ。



演習 4 次のような手続きを追加して円以外の図形を塗ってみよ。

a. ドーナツ型を塗るメソッド。
b. 長方形または楕円を塗るメソッド。
c. 三角形を塗るメソッド。
d. その他、自分の好きな形を塗るメソッド。

演習 5 どの図形でもよいが、色を塗る際に、単色で単純に塗るのでなく、次のような塗り方ができるようにしてみよ。

a. 2 色を指定して、ストライプ、ボーダー、チェックなどで塗れるようにする。
b. 色を塗る際に、「重ね塗り」できるようにしする。つまり**透明度** (transparency)$0 \leq t < 1$ を指定し、各 R/G/B 値について単に新しい値で上書きする代わりに $t \times c_{old} + (1-t) \times c_{new}$ のように混ぜ合わせた値にする。
c. 徐々に色調が変わっていくようにする。(注意: RGB 値は 0〜255 の「整数」でなければならない! 実数で計算した場合はその値が x の場合、「`x.to_i`」で**小数部分を切り捨てて**整数にできる。)
d. ぼやけた形、ふわっとした形などを表現してみよ。
e. その他、美しい絵を描くのにあるとよい機能を工夫してみよ。

演習 6 何か好きな絵を生成してみなさい。

本日の課題 5A

「演習 2」または「演習 3」で動かしたプログラム (どれか 1 つでよい) を含むレポートを提出しなさい。プログラムと、簡単な説明が含まれること。アンケートの回答もおこなうこと。

Q1. 画像のデータ構造について学びましたが、納得しましたか。

Q2. どのような画像を生成してみたいと考えていますか。

Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

次回までの課題 5B

「演習 2」〜「演習 6」の (小) 課題から選択して 1 つ以上プログラムを作り、レポートを提出しなさい。プログラムと、課題に対する報告・考察 (やってみた結果・そこから分かったことの記述) が含まれること。アンケートの回答もおこなうこと。

Q1. 簡単なものなら自分が思った画像が作れますか。

Q2. うまく画像を作り出すコツは何だと思いますか。

Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。