情報科学 2011　久野クラス　＃ 4

久野　靖*

2011.10.28

はじめに

今回の主な内容は次の通りです。

- 手続き (メソッド) による抽象化と再帰関数
- レコード型、画像の生成

手続きが使えると自分のプログラムを見通しよく書けるようになりますので、意識して活用してください。そして、画像が生成できるとこれまでの数字ばっかりとは違ったものが扱えて楽しめると思いますので、それを B 課題にします。「自分が構想したものを作る」という経験をぜひ積んで頂きたいと思います。

1　演習問題解説: 制御構造と配列の活用

1.1　演習 1 — fizzbuzz

演習 1 は繰り返しと枝分かれの基本的な組み合わせなので、さっさと Ruby コードを示します。 まず 2 の倍数と 3 の倍数以外を打ち出すものから:

```
def fizz2
  100.times do |i|
    if i % 2 != 0 && i % 3 != 0
      puts(i)
    end
  end
end
```

条件が読みにくいかもしれませんが、「2 の倍数でなく、3 の倍数でもないもの」を打ち出すと考えれば、これでよいと分かります。

次に演習 1b ですが、これは if-else の連鎖で「3 の倍数」「5 の倍数」「3 と 5 の公倍数 (15 の倍数)」「それ以外」の 4 つに場合分けするのが一番素直です:

```
def fizzbuzz1
  100.times do |i|
    if i % 15 == 0
      puts('fizzbuzz')
    elsif i % 3 == 0
      puts('fizz')
    elsif i % 5 == 0
      puts('buzz')
    else
```

*筑波大学大学院経営システム科学専攻

```
      puts(i)
    end
  end
end
```

なぜ 15 の倍数を最初に調べているのか分かりますか。それは、else-if の連鎖では上から順に条件を調べていくので、先に「3 の倍数」や「5 の倍数」を調べてしまうと、15 の倍数は 3 や 5 の倍数でもあるので条件が成り立ってその枝が選ばれてしまい、15 の倍数だけの枝には決して来なくなってしまうからです。

ところで、せっかく「3 の倍数なら fizz」「5 の倍数なら buzz」「両方の倍数なら fizzbuzz」となっているのだから、両者をうまく組み合わせたいと思う人もいるかもしれません。そのようなコードも作ってみましょう: [1]

```
def fizzbuzz2
  100.times do |i|
    num = i
    if i % 3 == 0
      print('fizz'); num = ''
    end
    if i % 5 == 0
      print('buzz'); num = ''
    end
    print(num); puts
  end
end
```

考え方としては、変数 num に打ち出す数を入れておきますが、3 の倍数なら fizz、5 の倍数なら buzz を打ち出すとともに num には空っぽの文字列を入れ直すので、最後の print で何も出力しなくなります。3 や 5 の倍数でないなら num には i が入っているので、そのまま出力されます。パラメタなしの puts は最後に行換えをするためです。

ところで、このコードはよくできている、と思いますか? 個人的には、このコードは先のコードより分かりにくいので、好きではありません。2 つの if の切れ目のところに合流がありますし、最後の数の出力を抑制するために変数 num を使ったりして、流れを追うのが難しくなっています。そんなことをするより、4 方向に枝分かれしてそれぞれの場合を明快に処理する先のコードのほうがずっと読みやすくてスマートだと思うのですが、どうでしょうか?

最後は「世界のナベアツ」ですが、「3 がつく数」はどうしましょうか。それは、対象とする数が 1 桁または 2 桁なので、「3 がつく」というのは 1 桁目が 3 か、2 桁目が 3 という意味になります。1 桁目が 3 というのは、10 で割った余りが 3 ということですし、2 桁目が 3 というのは、10 で切捨て除算した結果が 3 ということですね。ここまで分かればあとは書くだけです:

```
def fizz3
  100.times do |i|
    if i % 3 == 0 || i % 10 == 3 || i / 10 == 3
      puts('aho')
    else
      puts(i)
    end
  end
end
```

1.2 演習 2 — 最大公約数

課題の擬似コードを Ruby に直したものは次のとおり:

[1] print は puts と同様に文字列や数値を打ち出すけれども、行換えはしないメソッドです。なぜこれを使うかというと、fizz と buzz をくっつけて 1 行に打ち出すためです。

```
def gcd1(x, y)
  while x != y
    if x > y
      x = x - y
    else
      y = y - x
    end
  end
  return x
end
```

なぜこれで最大公約数が求まるのでしょうか? 次のように考えてください (x と y は正の整数であるものとします):

- $x = y$ であれば、最大公約数は x そのもの。当然ですね。
- $x > y$ であれば、x と y の最大公約数は $x - y$ と y の最大公約数に等しい。[2]
- したがって、$x - y$ を改めて x と置いて、x と y の最大公約数を求めればよい。
- $x < y$ の場合も同様。
- この手順の反復ごとに、x または y のどちらかがより小さくなるが、0 以下にはならない (大きい方から小さい方を引くから)。
- ということは、この反復は有限回で止まる。
- ということは、そのとき $x = y$ が成り立ち、x が一番最初の x と y の最大公約数に等しい。

繰り返しを使うときは「必ず止まって、止まった時には求める状況が成り立っている」ように設計する、という感じがお分かりになりましたか?

1.3 演習 3 — 素数判定

素数の判定ですが、擬似コードは次のとおり:

- isprime1: N が素数か否かを返す
- 　sosu ← 「真」。
- 　i を 2 から $N - 1$ まで変化させながら繰り返し:
- 　　もし N が i で割り切れるならば、sosu ←「偽」
- 　繰り返し終わり。
- 　sosu を返す。

変数 sosu は先に「はい」を表す true を入れておきます。そして素数でないと思ったら「いいえ」を表す false 入れ、最後に結果がどちらか見ます。このような使い方の変数のことを**旗** (flag) と呼びます。最初「旗」が立っていて、後で見たら「旗」が降りていたとすれば、誰が降ろしたかは分からないとしても、少なくとも誰かが旗を降ろしたことは確実に分かるわけです。では Ruby コードを見てみましょう:

```
def isprime1(n)
  sosu = true
  2.step(n-1) do |i|
    if n % i == 0 then sosu = false end
  end
  return sosu
end
```

[2]証明: 最大公約数を G と置くと、x も y も G の整数倍なのだから、$x - y$ もまた G の整数倍です。ということは、G は $x - y$ と y の公約数です (最大かどうかはまだ分かりません)。ところで、もし最大公約数で「なかった」とすると、最大公約数 $H(> G)$ が別にあるわけで、H は y の約数かつ $x - y$ の約数になります。ということは、H は $x - y + y = x$ の約数でもあります。これは G が x と y の最大公約数であるということに矛盾します。したがって G は $x - y$ と y の最大公約数でもあることになります。

ところでこの step とは…これも新しい計数ループで「整数.step(上限, 増分) do |i| ... end」で指定した整数から初めて 上限まで増分ずつの等差数列を i に入れながら回るループです (増分は省略すると 1 です)。そんなの習っていないからずるい、ですか? times でも次のようにすればできますね。

```
(n-2).times do |i|
  if n % (i+2) == 0 then sosu = false end
end
```

同じだといっても間違えやすいでしょうから、こういう場合は step を使いましょう。

1.4 演習 4 — 素数列挙とその改良

もっとも素朴な素数列挙プログラムは、上の素数判定を利用すれば簡単にできます。Ruby のコードを直接示しましょう:

```
def primes1(n)
  2.step(n-1) do |i|
    if isprime1(i) then puts(i) end
  end
end
```

これを手もとのマシンで動かしてみましたところ、10 秒間でおよそ 5,500 まで調べられましたね。これはあまり速くはないです。ところで、先の素数判定 isprime は「割り切れる」と分かってもそこでやめないで n の手前までずっと割っていきますから、早い段階で割り切れた数に対しては非常に無駄が大きいと思われます。そこで、改良版を作ってみました:

```
def isprime2(n)
  2.step(n-1) do |i|
    if n % i == 0 then return false end
  end
  return true
end
```

こちらは割り切れると分かったら直ちに return で「いいえ」を返しますから、無駄な割り算はしなくてすみます。上の primes1 をこちらを使うように直したところ、10 秒間で 17,000 くらいまで調べられました。つまり速度が倍以上になったわけです。

さらに考えると、割り算は $N-1$ までやる必要はなく、$\sqrt{N}$ まで調べて割り切れなければそれ以上やっても割り切れないと分かります ($\sqrt{N}$ よりも大きい因数があるなら、小さい因数もあるはずですから)。そこで素数判定を次のように改良します:

```
def isprime3(n)
  2.step(Math.sqrt(n)) do |i|
    if n % i == 0 then return false end
  end
  return true
end
```

これで試してみると、10 秒間で 140,000 くらいまで調べられました。

次に、2 は素数であり、2 の倍数は素数でないことが分かり切っているので調べる必要がない、ということを利用しましょう。このため、3 以上の奇数だけで割ってみる「改編版」の素数判定を作って使います。

```
def isprime4(n)
  3.step(Math.sqrt(n), 2) do |i|
    if n % i == 0 then return false end
  end
```

```
    return true
  end
  def primes4(n)
    puts(2)
    3.step(n-1) do |i|
      if isprime4(i) then puts(i) end
    end
  end
```

こんどは 10 秒間で 220,000 くらいまで調べられました。

もうちょっとだけ頑張って、では 2 と 3 より大きい素数は 6 の倍数± 1 だけ (それ以外は 2 と 3 の倍数になってしまう)、ということを利用してさらに調べる数を減らしてみましょう。

```
  def isprime5(n)
    6.step(Math.sqrt(n), 6) do |i|
      if n % (i-1) == 0 then return false end
      if n % (i+1) == 0 then return false end
    end
    return true
  end
  def primes5(n)
    puts(2)
    puts(3)
    6.step(n-1, 6) do |i|
      if isprime5(i-1) then puts(i-1) end
      if isprime5(i+1) then puts(i+1) end
    end
  end
```

こんどは 10 秒間で 300,000 くらいまで調べられました。コンピュータが高速だといっても、大量に計算する場合にはやはり工夫する価値はあるわけです。

1.5 演習 6 — 区間 2 分法とニュートン法

区間 2 分法とニュートン法は擬似コードまで説明してあったので、簡単に Ruby コードを示すだけにします。まず区間 2 分法:

```
  def sqrt2bun(n, e)
    a = 0.0
    b = n
    while (a-b).abs > e do
      c = 0.5 * (a+b)
      if c**2 > n then b = c else a = c end
    end
    return a
  end
```

そしてニュートン法:

```
  def sqrtnewton(n, e)
    r = 0.0
    r1 = n
```

```
  while (r1-r).abs > e do
    r = r1
    r1 = 0.5*r + n/(2.0*r)
  end
  return r
end
```

こちらのほうが理屈は面倒ですが、コードは枝分かれが無くて済むので簡単ですね。

演習 7 — 配列の演習

演習 7 も擬似コード略して Ruby のコードだけ記します。まず最大:

```
def arraymax(a)
  max = a[0]
  a.each do |x|
    if x > max then max = x end
  end
  return max
end
```

このような形で配列を使う場合は、ふつうは「とりあえず max に最初の値を入れておき、より大きい値が出てきたら入れ換える」方法になります。each は配列の各要素を順に取り出して来るメソッドでした。

次は最大の値が何番目に出てくるかなので、普通の計数ループにします。また、「何番目か」も変数に記録し、最大を更新した時にに同時に更新します:

```
def arraymaxno(a)
  max = a[0]
  pos = 0
  a.each_index do |i|
    if a[i] > max then max = a[i]; pos = i end
  end
  return pos
end
```

配列の各添字を列挙するには a.length.times を使えばよいのですが、ここに示したように配列のメソッド a.each_index を使うこともできます。

最大を 1 箇所だけ記録するのは変数でもできましたが、最大が複数あった時にその位置を全部打ち出すのには、(1) まず最大を求め、(2) その最大と等しいものがあったら位置を打ち出す、という形で 2 回ループを使う必要があります:

```
def arraymaxno2(a)
  max = a[0]
  a.each_index do |i|
    if a[i] > max then max = a[i] end
  end
  a.each_index do |i|
    if a[i] == max then puts(i) end
  end
end
```

平均より大きい値を打ち出すのもこれと同様です:

```
def arrayavglarger(a)
  sum = 0
  a.each do |x| sum = sum + x end
  avg = sum.to_f / a.length
  a.each do |x|
    if x > avg then puts(x) end
  end
end
```

ところで、この「sum.to_f」というのは? これは、合計を実数に変換するメソッドを呼んで、その結果に対して割り算をするようにしているものです。そうしないと、たまたま配列の中身が全部整数だった場合、a.length も整数ですから切捨て除算になってしまって、平均の計算が正しくならないからです。

演習 8 — 配列を使った素数列挙

まず最初は、これまでに見つかった素数を配列に覚えておく方法です: [3]

```
def isprime8(a, n)
  limit = Math.sqrt(n).to_i
  a.each do |i|
    if i >= limit then a.push(n); return true end
    if n % i == 0 then return false end
  end
  a.push(n); return true
end
def primes8(n)
  a = []
  2.step(n-1) do |i|
    if isprime8(a, i) then puts(i) end
  end
end
```

素数判定メソッドは素数の入った配列を受け取り、そこから順に素数を調べて候補の数に対して割り切れるかどうか調べていきます。ただし、候補の数の平方根まで来たらそれ以上やっても割り切れないことが分かるので「素数である」という答えを返します。なお、素数だった時は後に備えて配列にその素数を追加しておきます。この方法だと、「2 や 3 の倍数を除外」などのワザを使っていないのにもかかわらず、手元のマシンで 10 秒間で 600,000 くらいまでの素数が調べられましした。

ただし、このあたりをやっていると分かりますが (もっと早く気づいた人もいることでしょう)、実は数値を表示するという処理にもかなり手間が掛かっています。計測するという観点からは、表示を省略して内部のチェックだけの時間を計った方がいいでしょう。でも、速さの違いを実感していただくには、画面に出力が出た方が分かりやすいので、課題としては画面に出力するようにしてあります。

もう 1 つ (エラトステネスのふるい) はこれまでと大幅に違う方法です:

```
def primes9(n)
  a = Array.new(n, true);
  2.step(n-1) do |i|
    if a[i] then
      puts(i)
      i.step(n-1, i) do |k| a[k] = false end
    end
```

[3]配列のメソッド push は配列の末尾に新たな値を追加します。

```
    end
  end
```

まず最初に、添字が 0〜$N-1$ の配列 a を作り、各要素の値は true としておきます。次に、2 から始めて各候補の数値 i について、a[i] が true ならそれは素数なので打ち出すとともに、その素数の倍数 k について a[k] を false にします。そうすると、調べて行ってまだ true の要素が素数として次々に拾われていくわけです。この方法は非常に高速で、10 秒間で 3,000,000 くらいの素数がチェックできました。

2 手続き/関数と抽象化

2.1 手続き/関数が持つ意味

手続きないし**サブルーチン**とは、ひとまとまりの動作に名前をつけ、他の箇所からの**呼び出し** (call) により実行できるようなもののことです。

多くのプログラミング言語では、手続きから値を返すことができ、「ある決まった手順で値を計算するもの」とも捉えられます。このため、手続きのことを**関数** (function) と呼ぶ言語もあります。そして既に学んできたように、Ruby では**メソッド**が手続きに相当します。

また、既にたくさん使ってきたように、手続き呼び出し時に**パラメタ**を渡すことで、その渡した値に応じた動作や処理を行わせることができます。

たとえば前節では「整数 n が素数かどうかを調べる」というメソッドを作り、「素数を列挙する」メソッドからはそれを呼び出していました。そのコードを少し手直して再掲します:

```
def isprime(n)
  2.step(Math.sqrt(n)) do |i|
    if n % i == 0 then return false end
  end
  return true
end
def primes(n)
  2.step(n-1) do |i|
    if isprime(i) then puts(i) end
  end
end
```

2 つのメソッドに分けると、何がよいのでしょうか? それは、2 番目のメソッド中で「もし i が素数なら」とひとことで書けるようになることです。

別の例として「n 未満の数で、2 つ違いの素数になっているものを打ち出す」という作業を考えてみます。これも上のメソッドがあれば、次のように書けます:

```
def adjacentprimes(n)
  2.step(n-3) do |i|
   if isprime(i) && isprime(i+2) then puts "#{i} #{i+2}" end
  end
end
```

つまり「もし i が素数で、かつ、i+2 が素数 なら」とひとことで言えます。中では複雑な計算が必要な手順であったとしても、まとめて名前をつけることによって、必要なら何箇所からでも呼び出せ、コードも分かりやすくなるわけです。これをもっと一般的に言うと、手続きによって**抽象化**が行える、ということになります。

抽象化とは、不要な細部を省いて問題の検討に必要なことがらだけを残すことです。たとえば、「n が素数かどうか」を調べる方法が一度分かれば、あとはそれを参照すればよいのであって、その中でどのように処理しているかは「不要な細部」として見ないで済むことが利点なのです。

2.2 手続き/関数と副作用

関数という言葉は数学でも使われますが、数学で言う関数は「入力空間ないし**定義域** (domain) から出力空間ないし**値域** (range) への**写像** (mapping)」であって、同じ入力 (パラメタ) を与えた場合は同じ結果を返します。

たとえば $f(x) = x^2$ であれば、$f(2)$ の値は 4 であり、計算するたびに違うということはあり得ません。ですから、関数の値を 1 回計算して取っておき、2 回目は取っておいた値を利用するのでも、2 回とも計算するのとで、結果は一緒です。

これに対し、プログラムにおける関数は「単なる計算手順」ですから、その計算のやり方によっては、毎回違う値を返すこともありますし、引数以外の部分で変化を及ぼすこともあります。これを一般に**副作用** (side effect) と呼びます。

たとえば一番簡単な例として、`puts` は呼び出すたびに画面に文字が出力されますから、1 回呼び出すのと 2 回呼び出すのでは結果が違います。つまり、**入出力** (input/output — キーボードや画面やファイルなどとの間でのデータのやりとり) は副作用の形で扱われます。

また、関数や手続きの中で外部の (関数や手続きの外で定義された) 変数を書き換える場合も副作用になります。これまで使って来た変数は**局所変数** (local variable) と呼ばれ、そのメソッドが実行されている間だけ存在していて、実行が終わると消滅します (メソッドのパラメタも局所変数の一種と考えられます)。

Ruby では先頭が英字で始まる名前の変数 (これまで普通につかってきた変数) はすべて局所変数 (またはパラメタ) であり、局所変数でない変数は先頭に`$`や`@`などの記号をつけて区別します。一方で、このような見た目で変数種別の区別をしない言語も多くあります。

これに対し、プログラムの実行中ずっと存在し続け、さまざまなメソッド中から参照できる変数を**広域変数** (global variable) と呼びます。Ruby では先頭に`$`のついた名前の変数が広域変数です。簡単な例を見てみましょう:

```
$x = 0
def myfunc(n)
  $x = $x + 1
  return n + $x
end
```

上の例では `myfunc` はパラメタ `n` に変数`$x` の値を足した値を返す関数ですが、呼ばれるたびに`$x` を増やしているので (副作用)、`n` の値が同じでも返す結果は毎回変わってきます。

手続きが副作用を持つのは、広域変数に対する書き換えだけとは限りません。先に出てきた `bubblesort` などは、パラメタとして渡された配列の中身を書き換えていますが、これも副作用の一種です。

2.3 再帰手続き/関数

関数や手続きの興味深い用法として、ある関数の中から直接または間接に自分自身を呼び出す、というものがあります。これを**再帰** (recursion) と呼びます。たとえば、前章でやった内容から、正の整数 x、y について、その最大公約数は次のように定義できます:

$$gcd(x,y) = \begin{cases} x & (x = y) \\ gcd(x-y,\ y) & (x > y) \\ gcd(x,\ y-x) & (x < y) \end{cases}$$

これにそのまま従って Ruby のメソッドを書くことができます:

```
def gcd(x, y)
  if x == y
    return x
  elsif x > y
    return gcd(x-y, y)
  else
    return gcd(x, y-x)
  end
end
```

プログラムそのものは大変分かりやすいですが、なぜ「堂々めぐり」にならずに計算が終わるのでしょうか。それは、図 1 を見れば分かります。

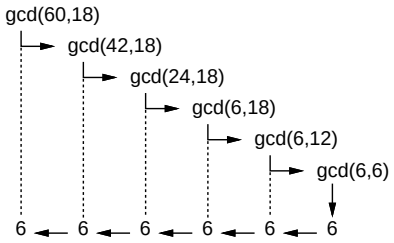


図 1: 再帰関数による最大公約数の計算

再帰関数 (再帰手続き) を作る時は、必ず次の原則に従います:

- 問題の「簡単な場合」は、すぐに答えを返す (上の例では $x = y$ の場合)。
- それ以外は問題を「少し簡単な問題に変形した上で」自分自身を呼び出す (上の例では、少し小さい数の最大公約数問題に変形)。

これがうまくできていれば、堂々めぐりにならずに正しく実行できるわけです。

演習 1 上の例題をそのまま打ち込んで動かせ。また、x や y に 0 や負の整数を入れるとどうなるかまず予測し、その後実際にやってみて確認せよ。

演習 2 次のような再帰的定義に従った計算を再帰関数として書いて動かせ。また、典型的な実行の様子を表す、図 1 のような図を描いてみよ。

a. 階乗の計算。

$$fact(n) = \begin{cases} 1 & (n = 0) \\ n \times fact(n-1) & (otherwise) \end{cases}$$

b. フィボナッチ数。

$$fib(n) = \begin{cases} 1 & (n = 0 \ or \ n = 1) \\ fib(n-1) + fib(n-2) & (otherwise) \end{cases}$$

c. 組み合わせの数の計算。

$$comb(n, r) = \begin{cases} 1 & (r = 0 \ or \ r = n) \\ comb(n-1,\ r) + comb(n-1,\ r-1) & (otherwise) \end{cases}$$

d. 正の整数 n の 2 進表現。[4]

$$binary(n) = \begin{cases} \text{“0”} & (n = 0) \\ \text{“1”} & (n = 1) \\ binary(n \div 2) + \text{“0”} & (n \text{ が } 2 \text{ 以上の偶数}) \\ binary(n \div 2) + \text{“1”} & (n \text{ が } 2 \text{ 以上の奇数}) \end{cases}$$

3 レコード型と画像

3.1 レコード型の利用

前章で説明したように、配列が「同じ型 (種類) の値が並んだもので、添字 (番号) により要素を指定する」のに対し、**レコード**は「違う型 (種類) の値でもよい、複数の値が組み合わさったもので、どの値 (フィールド) かは名前で指定する」ものでした。Ruby ではレコード型は `Struct.new` を使ってまずレコードクラスを定義し、その後その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: [5]

[4] この場合、関数の返す値は文字列であることと、+ は文字列の連結演算、÷は整数の除算 (切捨て除算) を表していることに注意してください。Ruby では整数どうしの「/」は自動的に切捨て除算になるのでしたね。

[5] レコード名は大文字で始まらなければなりません。

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

ここで「:名前」は先に説明した **記号** (symbol) リテラルで、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。

たとえば、コンピュータ上で画像を扱う時は、多数の**ピクセル** (pixel — 画素とも呼び、画面上の小さな点に対応) として扱うこと、そして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方法が多く使われることはご存じと思います。このピクセルの情報を表すレコードを定義してみましょう:

```
Pixel = Struct.new(:r, :g, :b)
```

実際にこのレコードを使う時は、次のようになります:

```
p = Pixel.new(255, 255, 255)  # RGB とも 255 の値
```

Ruby では配列の時と同様、レコードも new を使って作り出さないと使えないのに注意してください。

さらにレコードの場合も配列と同様、レコード本体はどこか「別の場所」に取られ、変数はそこを「指している」ことに注意してください。つまり、変数 p は最初は何も指していない状態 (図 2 上) で、代入によってレコードを指します (図 2 下)。「何も指していない状態」はレコード、配列、オブジェクト型の変数すべてに存在しえる状態で、その時はそれらの変数には **nil** という特別な値が入っています。

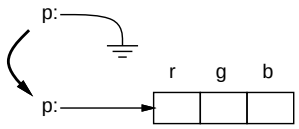


図 2: レコードの割り当て

3.2 2 次元配列

ピクセル 1 個ではつまらないので、ピクセルが 2 次元に (縦横に) 並んだ配列、つまり **2 次元配列** (two-dimensional array) [6] を作って、200 × 300 の画像を表すことにします。まず次のコードで、要素数 200 の 1 次元配列ができ、その配列を変数$img に入れます:

```
$img = Array.new(200)
```

配列の添字としては$img[0]〜$img[199] が使えるわけです。2 次元配列ということは、この各要素がさらに配列であればよいことになります (図 3)。このような構造を作るには、最初の 1 次元配列の各要素に大きさ 300 の配列を格納すればよいので、引き続いて次のようにします:

```
200.times do |j|
  $img[j] = Array.new(300)
  (以下略)
```

この 2 次元配列 (正確に言えば「配列の配列」) の各要素を指定するのには、$img[10][25] のように 2 つ添字を指定することになります (2 次元目の添字の範囲は 0〜299 ということになります)。これで数学でいう x_{ij} のような 2 つの添字つき変数ができたわけです。

ところでここでレコードの話題に戻りますが、今はこの 2 次元配列の各要素が RGB 値を持ったピクセルなので、2 次元配列を作ったらすぐにその各要素にレコード値を入れることにします:

[6] 添字が 2 つあるような配列。数学でいえば x_{ij} のようなものに相当します。

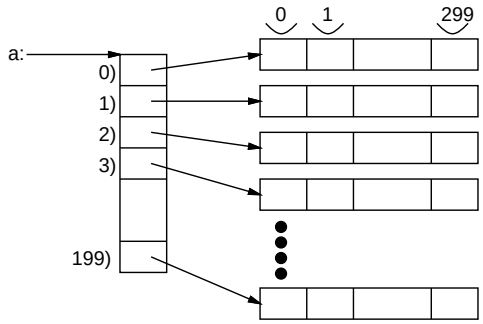


図 3: 2 次元配列

```
$img = Array.new(200)
200.times do |j|
  $img[j] = Array.new(300)
  300.times do |i| $img[j][i] = Pixel.new(255,255,255) end
end
```

(255,255,255) というのは赤緑青とも最大に光った色、つまり「真っ白」を意味しています。

ここまでは多くの言語にある普通のやり方について説明しましたが、Ruby には「配列の生成時にブロックを実行して初期値を決める」という機能があり、それを使うと同じことがもっと簡潔に書けます。この方法は次の例のような書き方によります: [7]

```
a = Array.new(10) do |i| i*i end  # 10 要素の配列を作成 (i 番目の初期値は i*i)
```

このブロックの中がさらに `Array.new` でもよいので、上に示した 200 × 300 の画像を作るのと同じことは次のようにも書けるのです:

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

3.3 例題: 画像を生成し書き出す

以下に示す例題は、色のついた円が 2 個重なっている画像を生成し書き出すものです。説明の都合上、4 つに分けて示していますが、もちろんこれらをくっつけた形でファイルに書いて読み込めばよいのです。まず最初に、レコード `Pixel` の定義があります。次は `initimage` ですが、これは前節で説明したとおり、2 次元配列を用意して、その各要素にピクセルを表すレコード (真っ白に対応) を入れています:

```
Pixel = Struct.new(:r, :g, :b)
def initimage
  $img = Array.new(200) do Array.new(300) do Pixel.new(255,255,255) end end
end
```

続いて、画像をファイルに書き出すメソッド `writeimage` を説明します。画像ファイルの形式には色々なものがありますが、ここでは、できるだけ出力が簡単な形式として **PPM**(Portable PixMap) 形式の画像を出力するようにしました。`writeimage` の引数とは作成するファイルの名前で、それをファイル読み書きの準備を行うメソッド `open` に渡します。`open` は指定された名前のファイルを用意し、その後ろのブロック内でそのファイルの読み (モード`"r"`の時)/書き (モード`"w"`の時) をできるようにしてくれます。さらにバイナリデータ (文字でないデータ) の場合は「`"rb"`/`"wb"`」を指定します。ブロック内では、`open` からパラメタ `f` として渡されてくる**ファイルオブジェクト** (file object — ファイルを読み書きする機能を提供するオブジェクト) を使用すればよいのです。:

[7] ブロックパラメタ (例では `i`) は添字番号を参照しないなら指定しなくても構いません。

```
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
```

ファイルに書くのは 2 箇所です。1 つ目は puts を使って、PPM ファイルの**ヘッダ** (header) を書き出します。ヘッダというのは音声や画像などのデータファイルの冒頭部分にある、その画像の種別やサイズなどの情報を格納した部分です。ここでは「色つき PPM で、サイズ 300 × 200、各色の最大値は 255」という情報を PPM で定めた形式で書いています。

2 つ目はループの中で、レコードの RGB 値を書いていきます。外側の each で各行の配列を取り出し、内側の each でその各ピクセルを取り出しています。各ピクセルのレコードについて、その 3 つのフィールドを to_a で 3 要素の配列に変換し、メソッド pack で「'ccc'(バイト 3 つ)」を指定することで各要素を 1 バイトずつのデータに変換し、それを write でファイルに書いています。

さて、初期化していきなりファイルに書いたら「真っ白」な画像ができるだけでつまらないので、何らかの「絵」を作成することにします。ここでは簡単な例として、「円形に色を塗る」メソッド fillcircle を作ります:

```
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
        $img[y][x].r = r; $img[y][x].g = g; $img[y][x].b = b
      end
    end
  end
end
```

この本体では、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 まで変化させる、ということです。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。

さて、最内側に if 文がありますが、その条件は「$(x-x_0)^2+(y-y_0) < rad^2$」ですから「点 (x,y) が中心 (x_0,y_0)、半径 rad の円内にある」ということですね。

そして、この条件が成り立つ時だけ、if の内側で$img[y][x] のレコードにパラメタとして渡された r、g、b 値を入れています。これを言い直すと、「画像上のあらゆる点 (x,y) のうちで、円内にあるものについてだけ、渡された色を設定する」つまり「円内を指定の色で塗る」ことになるわけです。

なお、なぜ y が先かというと、ここで使っている表現は「配列の 2 つの添字のうち 1 番目が Y 座標、2 番目が X 座標で、しかも画像の下のほうほど Y が大きい」という奇妙なものだからですが、コンピュータグラフィクスではなぜかこうすることが多いようです (図 3 もこれに合わせて描いてあります)。

以上で「材料」が揃ったので、画像を初期化し、円を 2 つ描き、ファイルに画像を書き出す」最後にメソッド mypicture を用意しました。打ち込む時に分かりやすいように、全体をまとめて掲載します:

```
Pixel = Struct.new(:r, :g, :b)
def initimage
  $img = Array.new(200) do Array.new(300) do Pixel.new(255,255,255) end 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 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
        $img[y][x].r = r; $img[y][x].g = g; $img[y][x].b = b
      end
    end
  end
end
def mypicture
  initimage
  fillcircle(110, 100, 60, 255, 0, 0)
  fillcircle(180, 120, 40, 100, 200, 80)
  writeimage("t.ppm")
end
```

これを実行するとファイル `t.ppm` に PPM 形式の画像ファイルが作成されます。この形式はあまり扱える**ビューア** (viewer — 画像表示ソフトウェア) がないので、次のコマンドで `PNG` 形式に変換してください。

```
convert t.ppn t.png
```

変換した後表示すると図 4 のように見えるはずです。なお、この `convert` というプログラムは「ImageMagick」というフリーのパッケージに入っているので、自宅などでやりたい人はこれを入れてください。または直接 PPM を表示できるソフトとして、Macintosh では **ToyViewer**、Windows では **PhotoFiltre** というビューアがあります。これらを入れるのでも構いません。

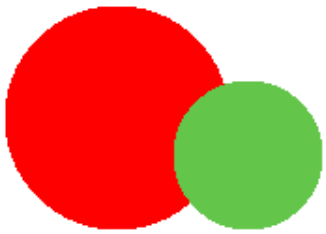
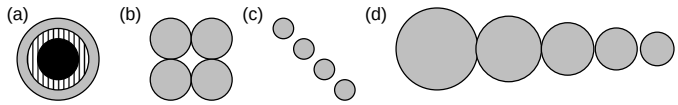


図 4: プログラムで生成した画像

演習 3 画像ファイルを生成する例題を打ち込んでそのまま動かせ。動いたら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 4 円を塗るメソッドを何回か呼び出すことで、次の図のように円を配置してみよ。



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)$0 \leq p < 1$ 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 $p\, r_{old} + (1-p)\, r_{new}$ のように混ぜ合わせた値にする。
c. 全体を均一に塗る代わりに、徐々に色調が変わっていくようにする。(注意: RGB 値は 0〜225 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 「連番の名前になった複数のファイルを生成」することで複数の画像を用いたアニメーションを生成してみよ (下記参照)。

演習 7 「美しい絵」を生成するプログラムを作れ。何が美しいかの定義は各自に任されるものとします (どーしてもアニメーションにしたければしてもよいです)。

3.4 おまけ 1: さまざまな図形を描くには

ここでもう 1 度、円を表示する部分がなぜそれでいいのか考えてみましょう。

```
200.times do |y|
  300.times do |x|
    …
  end
end
```

この 2 重ループの内側は結局、「すべての (x, y) について」実行されるのでしたね。そして円というのはどのように定義されるかというと、

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

ですから、この条件を満たすという if 文の中で、満たす場合だけ「その点の色を設定」していたわけです。では、長方形だったらどうでしょう? XY 軸に沿ったものなら

$$\{\ (x,y) \mid x_0 \leq x \leq x_1 \wedge y_0 \leq y \leq y_1\ \}$$

ですね。三角形は? それは、三角形は 3 つの辺で囲まれた領域ですよね (当り前だ)。1 つの直線は、平面を 2 つの半平面に分けます。直線だと指定しにくいので、直線に含まれる線分 $(x_0, y_0) - (x_1, y_1)$ で指定することにして、この半平面の点の集合は次の式で表されます。

$$\{\ (x,y) \mid (x_1-x_0)(y-y_0) - (y_1-y_0)(x-x_0) \geq 0\ \}$$

で、3 つの半平面に「ともに」含まれる点 (つまり共通集合)、というのが三角形なわけです (図 5)。

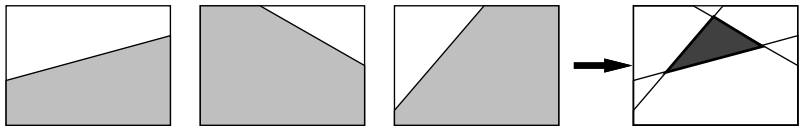


図 5: 三角形は 3 つの半平面の共通集合

3.5 おまけ 2: アニメーションを作るには

演習 6 のためにアニメーションの例を簡単に説明しましょう。以下は「赤い円が下に動いて行き、青に変わって小さくなる」というアニメーションを作成しています。

```
def myanim
  count = 100
  20.step(80, 4) do |y|
    initimage; fillcircle(110, y, 30, 255, 0, 0)
    writeimage("a#{count}.ppm"); count += 1
  end
  30.step(10, -1) do |r|
    initimage; fillcircle(110, 80, r, 0, 0, 255)
    writeimage("a#{count}.ppm"); count += 1
  end
end
```

ファイル名は a100.ppm、a101.ppm、…という連番になります (ファイル名の長さが変わって欲しくないので 100 からにしています)。かなり時間が掛かりますが、生成が終わったらこれらの PPM ファイルをたばねてアニメーション GIF にします。

```
convert -set delay 5 a*.ppm result.gif
```

これをブラウザなどで見ればよいわけです。

3.6 おまけ 3: フラクタル

フラクタルな図形というのは、再帰性のある図形 (その図形の一部分が、全体と相似になっているような図形) を言います。たとえば、図 6 を見ると、「正方形の 4 隅に小さい正方形がくっついている」という構造が繰り返されています。

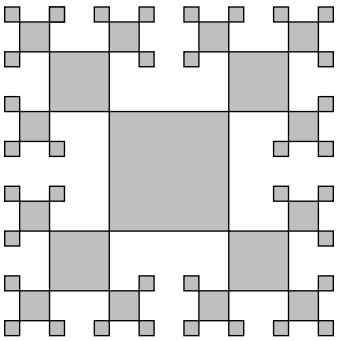


図 6: フラクタルな図形の例

こういうのは、再帰的なメソッドで作れます。どのみち、図形が小さくなりすぎたら描けないので、それが終了条件となります。たとえば次のような擬似コードを考えてみてください (正方形を描くメソッド fillsquare は別にあるものとします。)

- squares: 中心 x, y, 1 辺 2 × len の正方形フラクタルを描く
- 　もし `len < 1` ならば 戻る。
- 　fillsquare(x, y, len)。
- 　half ← len / 2。
- 　squares(x+len+half, y+len+half, half)。
- 　squares(x+len+half, y-len-half, half)。
- 　squares(x-len-half, y+len+half, half)。
- 　squares(x-len-half, y-len-half, half)。

なお、このようにきっちり機械的にやると人工物ぽくなりますが、乱数を使って「大きさがランダムに変動したり」「子供が確率的にできたりできなかったり」すると、自然物ぽくなります (自然界はフラクタルだと言われている)。Ruby では乱数は次の 2 つの方法で使えます。

- `rand()` — 0 以上 1 未満の実数値の一様乱数が得られる。
- `rand(`N`)` — N は整数として、0 から $N-1$ までの整数の一様乱数が得られる。

A　本日の課題 4A

「演習 2」で動かしたプログラムを含む小レポートを今日中に久野までメールで送ってください。

1. Subject: は「Report 4A」とする。
2. 学籍番号、氏名、ペアの学籍番号 (または「個人作業」)、投稿日時。
3. 「演習 2」で動かしたプログラムどれか 1 つのソース。
4. 以下のアンケートの回答。

　Q1. 手続き/関数について学びましたが、納得しましたか。

　Q2. 画像の生成について学びましたが、使えそうですか。

　Q3. 本日の全体的な感想と今後の要望をお書きください。

B　次回までの課題 4B

次回までの課題は「演習 2〜6」の (小) 課題 (4A で提出したものは除く) から 1 つと、「演習 7」との合計 2 つとします。演習 7 のプログラムが生成する画像についてはこのクラスの Web に掲示して皆様で相互に観賞して頂こうと思いますので、公序良俗に反する (ネットに掲示できない) 画像を生成するのはやめてください。そして、アイデア次第で前向きに「◎」をつけようと思いますので、すこしひねってみてください。**今回に限り、ペアで同じ絵にしないでください。図柄が同じで色違いは認めますので、それほど厳しい条件ではないと思います。**

各課題のために作成したプログラムはすべてレポートに掲載してください。また、今回は特に「どういう考えでこの画像を作ったか」の考察が重視されますのでよろしく。レポートは次回授業の **10 分前**までに、上記と同様に久野までメールで送付してください。

1. Subject: は「Report 4B」とする。
2. 学籍番号、氏名、投稿日時を書く。
3. 演習 1〜7 で作成しだプログラムのソース。
4. その説明と分析/考察。
5. 演習 7 で作成したプログラムのソース。
6. その説明と分析/考察。
7. 下記のアンケートの回答。

　Q1. 手続きが使いこなせるようになりましたか。

　Q2. 思ったような画像を生成できましたか。できた/できなかったとすれば、どこがポイントでしたか。

　Q3. 課題に対する感想と今後の要望をお書きください。