プログラミングプロジェクト＃3 – 反復と込み入った制御構造

久野　靖 (電気通信大学)

2021.3.19

前回は始めてのプログラミングなので一直線のコードで計算するだけでしたが、今回はいよいよ手続きや反復をやります。今回の目標は次の通り。

- 基本的なループ (反復) を含むアルゴリズムやプログラムについて考えられるようになる。
- 枝分かれとループの組み合わせが扱えるようになる。

まず前回の演習問題から抜粋して解説します (自分で課題をやってから読むことを勧めます)。

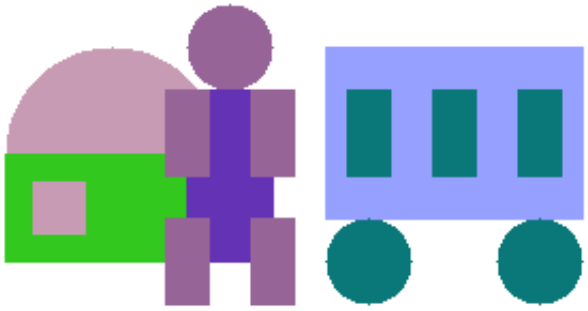
1　前回演習問題の解説

1.1　演習 1a — 図形の絵

設計図の 1 目盛を u として 3 つの図形の絵を 3 つの手続きとして作りました。課題としてはどれか 1 つでもかまいません。プログラムはライブラリを省略しました。

```
def prac1a(u, x, y, r1, g1, b1, r2, g2, b2)
  fillcircle(x, y, 2*u, r1, g1, b1)
  fillrect(x, y+u, 4*u, 2*u, r2, g2, b2)
  fillrect(x-u, y+u, u, u, r1, g1, b1)
end
def prac1b(u, x, y, r1, g1, b1, r2, g2, b2)
  fillcircle(x, y, u, r1, g1, b1)
  fillrect(x, y+3*u, 2*u, 4*u, r2, g2, b2)
  fillrect(x-u, y+2*u, u, 2*u, r1, g1, b1)
  fillrect(x+u, y+2*u, u, 2*u, r1, g1, b1)
  fillrect(x-u, y+5*u, u, 2*u, r1, g1, b1)
  fillrect(x+u, y+5*u, u, 2*u, r1, g1, b1)
end
def prac1c(u, x, y, r1, g1, b1, r2, g2, b2)
  fillrect(x, y, u*6, u*4, r1, g1, b1)
  fillrect(x-2*u, y, u, 2*u, r2, g2, b2)
  fillrect(x, y, u, 2*u, r2, g2, b2)
  fillrect(x+2*u, y, u, 2*u, r2, g2, b2)
  fillcircle(x-2*u, y+3*u, u, r2, g2, b2)
  fillcircle(x+2*u, y+3*u, u, r2, g2, b2)
end
def prac21
  prac1a(25, 70, 90, 200, 155, 180, 50, 200, 30)
  prac1b(20, 125, 40, 150, 100, 150, 100, 50, 180)
  prac1c(20, 230, 80, 150, 160, 255, 10, 120, 120)
```

```
  writeimage('prac.ppm')
end
```



1.2 演習 2 — 枝分かれ

演習 2a は例題とほとんど同じです。まず擬似コードを見てみましょう。

- max2: 数 a、b の大きいほうを返す
- 　もし a > b であれば、
- 　　$result \leftarrow a$。
- 　そうでなければ、
- 　　$result \leftarrow b$。
- 　枝分かれ終わり。
- 　$result$ を返す。

Ruby では次のとおり。

```
def max2(a, b)
  if a > b
    result = a
  else
    result = b
  end
  return result
end
```

これも、次のような「別解」があり得ます。

- max2x: 数 a、b の大きい方を返す
- 　$result \leftarrow a$。
- 　もし $b > result$ であれば、
- 　　$result \leftarrow b$。
- 　枝分かれ終わり。
- 　$result$ を返す。

これの Ruby 版は次のとおり。

```
def max2x(a, b)
  result = a
  if b > result then result = b end
  return result
end
```

どちらが好みですか? これもどちらが正解ということはありません。

ところで、「2 数が等しい場合はどうするのか」について皆様の中には迷った人がいると思います。問題には「異なる数」と書いてあるので考えなくてもよいのですが、仮にそれが書いていなかったとします。そうなると、等しい場合について何らかの指示が本来あるべきですよね。たとえば次のものがあり得ます。

- 「等しい場合はその等しい数を返す」
- 「等しい場合は何が返るかは分からない」
- 「等しい数を渡してはならない」

上 2 つの場合は例解のままで OK です (2 番目では何が返ってもよいので、等しい数でもよい)。最後の場合はどうでしょう。次の考え方があり得ます。

(a) 「渡してはならない」以上、渡されることはないのだから、例解のままでよい

(b) 「渡してはならない」値が渡されたのだから、エラーを表示するなどして警告するべき

どちらにも (互いに裏返しの) 利点と弱点があります。(a) の方が簡潔で短く間違いが起きにくいですが、(b) の方が起きるべきでないことが起きていることが分かるので対処が必要な場合には有用です。

で、あなたは発注者 (教員) の注文を受けてこの課題をやっているわけですから、正解は発注者に「どうしますか」と確認することです。そうすれば、どちらにするかは決められるでしょう。勝手に (b) を選んでプログラムを複雑で間違いやすいものにするのはいかがかと思いますし、発注者が「等しい場合はその等しい数を返す」と書き忘れただけだったら目もあてられませんね。

1.3 演習 2b — 枝分かれの入れ子

演習 2b はもう少し複雑です。まず考えつくのは、a と b の大きいほうはどちらかを判断し、それぞれの場合についてそれを c と比べるというものでしょうか。

- max3: 数 a、b、c で最大のものを返す
- もし $a > b$ であれば、
- もし $a > c$ であれば、
- $result \leftarrow a$。
- そうでなければ、
- $result \leftarrow c$。
- 枝分かれ終わり。
- そうでなければ、
- もし $b > c$ であれば、
- $result \leftarrow b$。
- そうでなければ、
- $result \leftarrow c$。
- 枝分かれ終わり。
- 枝分かれ終わり。
- $result$ を返す。

かなり大変ですね。これを Ruby にしたものは次のとおり。

```
def max3(a, b, c)
  if a > b
    if a > c
      result = a
    else
      result = c
```

```
    end
  else
    if b > c
      result = b
    else
      result = c
    end
  end
  return result
end
```

こうなると字下げしてないとごちゃごちゃになるでしょう? しかし字下げしてあってもこれはかなり苦しいですね。一般に、if の中に if を入れると非常に分かりづらくなるので、できるだけ避けたほうがよいのです。

ところで、先の別解から発展させるとどうなるでしょう?

- max3x: 数 a、b、c で最大のものを返す
- result ← a
- もし $b > result$ であれば、$result \leftarrow b$。
- もし $c > result$ であれば、$result \leftarrow c$。
- $result$ を返す。

「もし」の擬似コードが 1 行に書かれていますが、この場合はこちらののほうが見やすいと思ったのでそうしてみました。Ruby でも次のとおり (こんどはどちらが好みですか?)。

```
def max3x(a, b, c)
  result = a
  if b > result then result = b end
  if c > result then result = c end
  return result
end
```

一般には、枝分かれの中に枝分かれを入れるよりは、枝分かれを並べるだけで済ませられればそのほうが分かりやすいと言えます。また、この方法では入力の数 N がいくつになっても簡単に対処できるという利点があります。

実は、さらなる別解があります。それは、既に `max2` を作ったわけですから、それを利用するというものです。

```
def max3xx(a, b, c)
  return max2(a, max2(b, c))
end
```

このように、一度作って完成したものは後から別のものを作る時の「部品」として使える、というのは重要な考え方です。このことも覚えておいてください。

1.4 演習 2c — 多方向の枝分かれ

演習 2c は 3 通りに分かれるので、if の中にまた if が入るのはやむをえないはずです。Ruby コードを見てみましょう。

```
def sign1(x)
  if x > 0
```

```
    return "positive."
  else
    if x < 0
      return "negative."
    else
      return "zero."
    end
  end
end
```

このような「複数の条件判断」はよく使うので、実はこれは if の入れ子にしなくても書けるようになっています。具体的には、if 文には「elsif 条件 then 動作」という部分を途中に何回でも入れられ、それを使うと次のようになります。

```
def sign2(x)
  if x > 0
    return "positive."
  elsif x < 0
    return "negative."
  else
    return "zero."
  end
end
```

順序が前後しましたが、擬似コードだと次のようになります。

- 実数 x を入力する。
- もし $x > 0$ ならば、
- 　　「positive.」を返す。
- そうでなくて $x < 0$ ならば、
- 　　「negative.」を返す。
- そうでなければ、
- 　　「zero.」を返す。
- 枝分かれ終わり。

「そうでなくて〜ならば、」は何回現われても構いません。また、そのどれもが成り立たない場合は「そうでなければ」に来るわけですが、この部分は不要なら無くても構いません。

1.5　演習 3 — 枝分かれのある絵

演習 3a は、u > 10 のとき車を 1 つ余計に描いています。演習 3b は、「はい」と「いいえ」で色を入れ換えなければならないので、ほとんど同じだけれど色指定を入れ換えた車をそれぞれの場合に描いています。

```
def prac3a(u, x, y, r1, g1, b1, r2, g2, b2)
  fillcircle(x-3*u, y+2*u, 2*u, r2, g2, b2)
  fillcircle(x+3*u, y+2*u, 2*u, r2, g2, b2)
  if u > 10
    fillcircle(x, y+2*u, 2*u, r2, g2, b2)
  end
  fillrect(x, y, 6*u, 4*u, r1, g1, b1)
end
```

```
def prac3b(u, x, y, r1, g1, b1, r2, g2, b2, rect)
  if rect
    fillcircle(x-3*u, y+2*u, 2*u, r2, g2, b2)
    fillcircle(x+3*u, y+2*u, 2*u, r2, g2, b2)
    fillrect(x, y, 6*u, 4*u, r1, g1, b1)
  else
    fillcircle(x-3*u, y+2*u, 2*u, r1, g1, b1)
    fillcircle(x+3*u, y+2*u, 2*u, r1, g1, b1)
    fillrect(x, y, 6*u, 4*u, r2, g2, b2)
  end
end
def prac23
  prac3a(10, 240, 30, 20, 40, 80, 250, 200, 200)
  prac3a(11, 180, 50, 200, 250, 40, 0, 150, 100)
  prac3a(14, 80, 50, 150, 150, 150, 255, 0, 0)
  prac3b(10, 240, 140, 20, 40, 80, 250, 200, 200, false)
  prac3b(11, 180, 150, 200, 250, 40, 0, 150, 100, false)
  prac3b(14, 80, 140, 150, 150, 150, 255, 0, 0, true)
  writeimage('pict.ppm')
end
```

2　繰り返し

2.1　while 文

ここまでは、プログラム上に書かれた命令はせいぜい 1 回実行されるだけでしたから、プログラムが行う計算の量はプログラムの長さ程度しかありませんでした。しかし、繰り返しがあれば、その範囲内の命令は何回も反復して実行されますから、短いプログラムでも大量の計算を行わせられます。

まず繰り返しの基本となる、条件を指定した繰り返しの擬似コードは次のように書き表します。[1]

- ～ である間繰り返し、
- 　　動作 1。
- 繰り返し終わり。

この形の繰り返しは、Ruby では **while 文** (while statement) として記述します。

```
while 条件 do
```

[1]「～」のところには条件を記述しますが、ここに書けるものは if 文の条件とまったく同じです。

```
  ... 動作 1 ...
end
```

条件の次にある do も、Ruby では省略することができます。ただし、「動作 1」を条件と同じ行に書く場合は省略できません。本資料では do は省略しないことにします。

多くのプログラミング言語では、このような条件を指定した繰り返しは while というキーワードを用いて表すので、**while ループ**と呼びます。while ループは形だけなら if 文より簡単ですが、慣れるまではどのように実行されるかイメージが湧かない人が多いと思います。while ループの実行のされ方は、次のようなものだと考えてください。

- 「〜」を調べる (成立)。
- 動作 1 を実行。
- 「〜」を調べる (成立)。
- 動作 1 を実行。
- 「〜」を調べる (成立)。
- 動作 1 を実行。
- …
- 「〜」を調べる (不成立)。
- 繰り返しを終わる。

つまり、条件を調べ、成り立てば動作 1 を実行し、また条件を調べ、…のように繰り返していき、条件が成り立たなくなると繰り返しを終わります。

2.2 例題: 車を繰り返し描く

それでは早速、繰り返しを使った絵を作って見ましょう。このプログラムは x、y を変数で持っています。

- repeatcars: 車を画面一杯にならべる
- x ← 30。 y ← 100。
- x ≤ 300 である間繰り返し、
- 位置 (x,y) に車を描く。
- x ← x + 80。
- ここまで繰り返し。
- 画像をファイルに出力。

車を描くのはいつもの car を使います。繰り返しの使い方ですが、まず x は 30 になっているのでこの場所に車を描き、そのあと x = x + 80 つまり x と 80 を足したものを計算して x に入れ直します。x が 300 以下なら (きっとそうです)、また新しい位置に車を描いて、また x を 80 増やします。何回もこれを繰り返すと、x が 300 を超えるので、ループをおわります。

これを Ruby に直したものを示します。絵のライブラリや car については、これまでとかわりません。なお、Ruby では 1 行に複数の文を書くときは「;」で区切る必要があります。

```
$img = Array.new(200) do Array.new(300) do [255,255,255] end end
def pset(x, y, r, g, b)
  if 0 <= x && x < 300 && 0 <= y && y < 200
    $img[y][x][0] = r; $img[y][x][1] = g; $img[y][x][2] = b
  end
end
def writeimage(name)
  open(name, "wb") do |f|
    f.puts("P6"); f.puts("300 200"); f.puts("255")
```

図 1: 車の繰り返し

```
    $img.each do|a| a.each do |p| f.write(p.pack("ccc")) end end
  end
end
def fillrect(x0, y0, w, h, r, g, b)
  (y0-h/2).step(y0+h/2) do |y|
    (x0-w/2).step(x0+w/2) do |x| pset(x, y, r, g, b) end
  end
end
def fillcircle(x0, y0, rad, r, g, b)
  (y0-rad).step(y0+rad) do |y|
    (x0-rad).step(x0+rad) do |x|
      if (x-x0)**2 + (y-y0)**2 <= rad**2 then pset(x, y, r, g, b) end
    end
  end
end

def car(u, x, y, r1, g1, b1, r2, g2, b2)
  fillcircle(x-3*u, y+2*u, 2*u, r2, g2, b2)
  fillcircle(x+3*u, y+2*u, 2*u, r2, g2, b2)
  fillrect(x, y, 6*u, 4*u, r1, g1, b1)
end
def repeatcars
  x = 30; y = 100
  while x <= 300 do
    car(8, x, y, 120, 40, 180, 250, 200, 20)
    x = x + 80
  end
  writeimage('pict.ppm')
end
```

このように、繰り返しを使うことで、短かいプログラムでも多数の絵を描いたり多くの計算をさせることができます。

演習 1 「車を繰り返し描く」の例題をそのまま動かしなさい。動いたら、次のように直してみなさい。

a. 例題では車が水平に動いていたが、斜めに動くようにしてみなさい。

ヒント: y も決まった値ずつ変化させる。

b. 車が 2 台動くようにしてみなさい。

c. その他自分がやってみたいと思う好きなものを動かす。

2.3 計数ループ

先のプログラムでは「x が一定値より小さい間だけ」という条件で繰り返しましたが、多くの場合は「10 回」のように回数指定で繰り返します。このような場合はどうしたらいいでしょう。一般に回数を n として、次のようなループを書けばよいのです。(**カウンタ** (counter) とは「数を数える」ために使う変数のことを言います)。

```
i = 0           # i はカウンタ
while i < n do # 「n 未満の間」繰り返し
  ...           #  ここでループ内側の動作
  i = i + 1     # カウンタを 1 増やす
end
```

このように指定した上限まで数えながら反復する繰り返しを**計数ループ** (counting loop) と呼びます。計数ループはプログラムで頻繁に使われるため、ほとんどのプログラミング言語は計数ループのための専用の機能や構文を持っています (while 文でも計数ループは書けますが、専用の構文のほうが書きやすくて読みやすいからです)。

Ruby では計数ループ用の構文として **for 文** (for statement) を用意しています。これを使って上の while 文による計数ループと同等のものを書くと次のようになります。

```
for i in 0..n-1 do
  ...
end
```

これは、カウンタ変数 i を 0 から初めて 1 つずつ増やしながら n-1 まで繰り返していくループとなります (多くのプログラミング言語では、計数ループを表すのに for というキーワードを使うので、計数ループのことを **for ループ**と呼ぶこともあります)。

せっかく for 文を説明しておきながら恐縮ですが、以下では計数ループを整数値が持つメソッド times を使って書くことにします。[2] これはたとえば次のようになります。

```
10.times do       10.times do |i|
  ...               ...
end               end
```

この times というのは、整数値 (上の場合は 10 です) に「対して呼び出せる」メソッドであり、さらにブロック (コードの並び、do〜end の部分) を受け取るようになっています。そしてそのブロックを数値の回数 (上の例では 10 回) 実行します。ブロックの指定のための do は省略できません。[3] さらに、右のように「|変数名|」という指定 (ブロックパラメタ) をつけると、ブロックの中で「何回目の繰り返しか」を受け取ることができます (最初が 0 回目になります)。これらを疑似コードでは次のように書くことにします。

n 回繰り返し、	i を 0 から $n-1$ まで変えながら繰り返し、
(繰り返す内容)	(i を参照する繰り返し内容)
ここまで繰り返し。	ここまで繰り返し。

[2] なぜ for 文でなく times を使うかというと、ブロックを受け取るメソッドは Ruby でさまざまな用途に使える便利な仕組みなので、そちらに慣れたほうがよいと思うからです。

[3] それで混乱しやすいので、while でも do を省略しないことにしたわけです。

絵を描くような場合は、この受け取ったカウンタの値をもとに (たとえば定数倍して) 座標を決めることで、「規則的な」並びを作ることができます。たとえば先の車の場合、回数は「4 回」と分かっていますから、次のようにしてもよいのです (以下、下請けコードは省略)。

```
def repeatcars1
  4.times do |i|
    car(8, 30+(i-1)*80, 100, 120, 40, 180, 250, 200, 20)
  end
  writeimage('pict.ppm')
end
```

しかし、30+(i-1)*80 とは? この式は i が 0、1、2、3 の時にそれぞれ 30、110、190、270 になるので、先の絵と同じものができます。

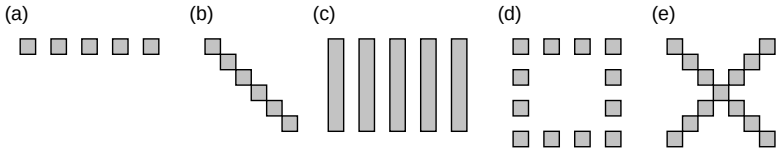
図 2: 固定回数の反復

絵をもっと簡単な丸にして、繰り返しの数を増やしてみます。図 2 のような並びを作る例題を見てみましょう。

```
def fixedrepeat
  10.times do |i|
    fillcircle(20*(i+1), 40, 10, 255, 0, 0);
  end
  writeimage("pict2.ppm")
end
```

今度は x は 20、40、60、80、、、と変化しますね。

演習 2 上の例題のプログラムを動かせ。動いたら、図のようなものを作ってみよ。色や繰返し回数などは好きにしてよい。



演習 3 手続きを使った込み入った絵をループにより繰り返して描いてみなさい。

演習 4 枝分かれにより変化する絵の手続きをループの中から呼び出して、並んだ絵が一部違うようにしてみなさい。

最後の問題をやるには、ループのなかで「はい」になったり「いいえ」になったりする計算式があるといいですね。そのような例を挙げて置きましょう。

- 1 回だけ (この場合は i が 2 のとき)「はい」 — `i == 2`
- 途中から (この場合は i が 2 以上)「はい」 — `i >= 2`
- i が偶数のとき「はい」 — `i % 2 == 0`
- i を 3 で割った余りが 1 以上なら「はい」 — `i % 3 >= 1`

本日の課題 **3A**

「演習 1」「演習 2」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. プログラムを打ち込んで動かすのに慣れましたか?

Q2. 自分にとって次の「難しいポイント」は何だと思いますか?

Q3. 本日の全体的な感想と今後の要望をお書きください。

次回までの課題 **3B**

「演習 3」「演習 4」の (小) 課題から選択して 2 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. 手続きの「抽象化」の役割が納得できましたか。

Q2. 繰り返しは納得できましたか。

Q3. 課題に対する感想と今後の要望をお書きください。