基礎プログラミング+演習 ＃2 – 分岐と反復+数値積分

久野 靖 (電気通信大学)

2017.12.11

前回はプログラミング入門なので一直線のコードで計算するだけでしたが、今回はいよいよ制御構造 (分岐や反復) を使っていただきます。今回の目標は次の通り。

- 基本的な制御構造 (分岐・反復) を理解し、これらを使ったプログラムが書けるようになる。
- 基本的な制御構造を用いた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。

以後毎回、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 3a — 四則演算を試す

演習 3a は和の計算でした。メソッド内の計算式を取り替えればいいだけなので簡単です。

```
def add(x, y)
  return x + y
end

irb> add 3.5, 6.8
=> 10.3
```

和、差、商、積の場合も同様でいいのですが、4 つメソッドを作る代わりに 1 つで済ませる方法を考えてみます (半分くらいは新しい内容の紹介を兼ねています)。まず先に説明したように、メソッドの最後に値を返す代わりに、puts などで順次画面に書き出す方法があります。

```
def shisoku0(x, y)
  puts(x+y)
  puts(x-y)
  puts(x*y)
  puts(x/y)
end

irb> shisoku0 3.3, 4.7
8.0
-1.4
15.51
0.702127659574468
=> nil
```

4 つの値が打ち出され、shisoku0 の結果としては **nil**(何もないことを示す値) が返されています。

上の方法だと「1 つの結果が返る」のでないのがちょっと、という気がするかもしれません。そこで次に、1 つの文字列を返し、その中に 4 つの数値が埋め込まれている、というふうにしてみましょう。

Ruby では**文字列** (string — 文字が並んだデータ) は「'...'」または「"..."」のように**シングルクォート**または**ダブルクォート**で囲んで表しますが、ダブルクォートのほうは内部に色々なものを埋め込む機能がついています。[1] 具体的には、文字列の中に「#{...}」という形のものがあると、中カッコ内の式を**評価** (evaluation — 値を計算すること) して、結果をそこに埋め込んでくれます。

これを利用した「四則演算」のメソッドを示します。

```
def shisoku1(x, y)
  return "#{x+y} #{x-y} #{x*y} #{x/y}"
end

irb> shisoku1 3.3, 4.7
=> "8.0 -1.4 15.51 0.702127659574468"
```

確かに、「文字列」が打ち出されていて、その中に 4 つの数値が埋め込まれています。

もう 1 つ、Ruby など多くの言語では値の並んだものを**配列** (array) という機能で扱います。Ruby では [...] の中に値をカンマで区切って並べることで配列を直接書けるので、これを使って 4 つの数値をまとめて返すことができます。[2]

```
def shisoku2(x, y)
  return [x+y, x-y, x*y, x/y]
end

irb> shisoku2 3.3, 4.7
=> [8.0, -1.4, 15.51, 0.702127659574468]
```

ここでは文字列の場合とあまり変わらない感じがするかもしれませんが、配列では返された値の中から「0 番目」「1 番目」など番号を指定して特定の要素を取り出せるので、より便利に使えます。

1.2 演習 3b — 剰余演算

演習 3b は剰余演算「%」を試すというものでした。演算子を取り換えるだけなので、プログラムは簡単ですね。

```
def jouyo(x, y)
  return x % y
end
```

実行してみましょう。

```
irb> jouyo 8, 5
=> 3
irb> jouyo 20, 5
=> 0
irb> jouyo -8, 5
=> 2
irb> jouyo -21, 5
=> 4
```

[1] ダブルクォートは埋め込み機能等のために**特殊文字** (special character — 英数字以外の文字) を様々に解釈します。そのようなことをせずに文字列をそのまま表示させたい場合はシングルクォートを使ってください。

[2] return の後だと囲んでいる [ ] を省略できますが、場所によっては省略できないので常に書くことを薦めます。

マイナスの時も試しましたか? 「ここでマイナスだとどうだろう」と気付くようになってください。

それで、マイナスの時も剰余は負にならず、つまり「5 間隔」というのがマイナスまでずっと続いている、というふうに考えるのでしょうね。では、割る数がマイナスだったらどうでしょうか?

```
irb> jouyo 8, -5
=> -2
irb> jouyo -8, -5
=> -3
```

剰余の符号は割る数の符号と一致するようになっているようです (剰余の振舞いは、プログラミング言語によって違いがあります)

演習 3c — 8 乗、6 乗、7 乗

演習 3c は 8 乗、6 乗、7 乗です。Ruby のべき乗演算子「**」を使えば次のように簡単です。

```
def x8a(x)
  return x**8
end
```

しかしこの演算を使わないとするとどうでしょうか。

```
def x8b(x)
  return x*x*x*x*x*x*x*x
end
```

もちろんこれでもいいのですが、乗算の数を減らす方法があります (「;」は 1 行に複数の文を書くときに区切りとして入れる必要があります)。

```
def x8c(x)
  x2 = x*x; x4 = x2*x2; return x4*x4
end
```

6 乗は「return x4*x2」、7 乗は「return x4*x2*x」ですね。「return x4*x4 / x」はどうでしょう? 除算は乗算より遅いので、無理のない範囲で少なくしておいた方がよいです。

演習 3d — 円錐の体積

演習 3d は円錐の体積でした。底面の半径 r、高さ h として、まず円錐の底面の面積は πr^2。体積はこれに高さを掛けて 3 で割ればできます。

```
def cornvol(r, h)
  return (r**2*3.1416*h) / 3.0
end
```

ちなみに「**」はべき乗の演算子です。もちろん 2 乗は「r*r」と書いても構いません。

```
irb> cornvol 3.0, 4.0
=> 37.6992
```

ところで「円周率が 3.1416 というのは不正確だ」と思う人もいそうですね。しかし、コンピュータ上の計算は「電卓での計算」と同様、有限の桁数でしか行えないのであり、自分で必要と思う適当な桁数を決めてその範囲でやるしかないので、有効数字 5 桁くらいでと思うならこれでよいわけです。[3]

[3] 3.141592653589793 くらいまでは扱える精度があるので、この定数をいちいち書くのは嫌だという人のために Math::PI と書いてもよいようになっています。同様に、自然対数の底 e は Math::E で表せます。

1.3 演習 3e — 平方根

平方根は Math.sqrt(x) で計算できるので、要するに何桁くらい精度があるか調べるわけです。

```
def sqrts
  printf("%.20g\n", Math.sqrt(2));
  printf("%.20g\n", Math.sqrt(3));
  printf("%.20g\n", Math.sqrt(5));
end
```

実行してみます。

```
irb> sqrts
1.4142135623730951455
1.7320508075688771932
2.2360679774997898051
=> nil
```

正確な平方根の値を掲げておきます (見比べると、精度としては 16〜17 桁程度であると言えます)。

$$\sqrt{2} \approx 1.4142135623730950488016887242096980785696$$
$$\sqrt{3} \approx 1.7320508075688772935274463415058723669428$$
$$\sqrt{5} \approx 2.2360679774997896964091736687312762354406$$

1.4 演習 4 — 2 次方程式の解の公式

まず素直に計算式通りコードを書きます。$\sqrt{D}$ を変数 rd に保存し、それを使って 2 つの解を計算しました。最後にその 2 つを配列として返すようにしました。

```
def solve1(a, b, c)
  rd = Math.sqrt(b**2 - 4*a*c)
  x1 = (-b - rd) / (2.0 * a)
  x2 = (-b + rd) / (2.0 * a)
  return [x1, x2]
end
```

では実行させてみます。

```
irb> solve1 1, -2, 1
=> [1.0, 1.0]
irb> solve1 1, 5, 6
=> [-3.0, -2.0]
```

とくに問題なさげです。

では設問 b に進んで、b と $\sqrt{D}$ が非常に近いものをやってみます。ヒントに書いたように、$(x+d)(x+1)$ で $d = 0.000000000012345$ をやってみましょう。

```
irb> solve1 1, 1.000000000012345, 0.000000000012345
=> [-1.0, -1.2345013900016966e-11]
```

「e-11」は「$\times 10^{-11}$」の意味でした。誤差はありますが、まあ計算できてます。ゼロを増やすと?

```
irb>  solve1 1, 1.000000000000012345, 0.000000000000012345
=> [-1.0, -1.2378986724570495e-14]
```

x2 の有効数字が 2 桁くらいに減ってしまった感じです。問題を克服するため、x2 の計算を解の公式ではなく c と x1 から求めるようにします (x1 については、b の符号が正なので $-b-rd$ は内容が足し算であり桁落ちは起きません)。

```
def solve2(a, b, c)
  rd = Math.sqrt(b**2 - 4*a*c)
  x1 = (-b - rd) / (2.0 * a)
  x2 = c / (a * x1)
  return [x1, x2]
end
```

先の値について実行してみると、完全にぴったりになると分かります。

```
irb> > solve2 1, 1.000000000000012345, 0.000000000000012345
=> [-1.0, -1.2345e-14]
```

1.5 演習 5 — 実数計算の誤差

実数計算の誤差を観察する問題なので、printf を用いて十分な桁数を表示します。まず a. から。

```
def kadai5a
  printf("%.20g\n", 1.12345 - 1.0)
  printf("%.20g\n", 1.1234512345 - 1.12345)
  printf("%.20g\n", 1.123451234512345 - 1.1234512345)
  printf("%.20g\n", 1.12345123451234512345 - 1.123451234512345)
end
```

```
irb> kadai5a
0.12345000000000005969      ←まあまあ
1.2345000000024697329e-06   ←誤差が
1.2345013900016965636e-11   ←増えて来て
0                           ←最後は近すぎるので 0
=> nil
```

値が近くなるにつれて桁落ちが現れてくることが分かります。次は b. です。

```
def kadai5b
  printf("%.20g\n", 1.0 + 0.0012345);
  printf("%.20g\n", 1.0 + 0.000000012345);
  printf("%.20g\n", 1.0 + 0.00000000000012345);
  printf("%.20g\n", 1.0 + 0.000000000000000012345);
end
```

```
irb> kadai5b
1.0012345123449999384  ← OK
1.000000012345123368   ←誤差が
1.0000000000001234568  ←増えて来て
1                      ←小さすぎると 1+ε =1 となる
=> nil
```

足す値が小さくなるほど下の桁は失われていき、最後はまったく値が増えなくなります。設問 c. はいろいろ試したいので irb で直接計算してみましょう。

```
irb> printf "%.20g\n", 1.0/3.0
0.33333333333333331483  ←有効桁数は 10 進で 16 桁程度
=> nil
irb> printf "%.20g\n", 1.0/3.0 * 3.0
1           ← 3 倍すると最後の桁が丸められて元に戻る
=> nil
irb> printf "%.20g\n", 7.0 / 10.0
0.69999999999999995559 ← 0.7 も 2 進で切りよくない
=> nil
irb> printf "%.20g\n", 7.0 * 0.1
0.70000000000000006661 ←値 0.1 も誤差を含むので
=> nil
```

このように、有限桁数の計算なので微妙に誤差が現れます。しかし一方で、2 進表現を使っているということは、2^N とか $\frac{1}{2^N}$ とかは非常に「切りのよい数」となり、あふれない限りは誤差が出ません。

```
irb> printf "%.40g\n", 7.0 / 16.0
0.4375
=> nil
irb> printf "%.40g\n", 7.0 * 0.0625
0.4375
=> nil
irb> printf "%.40g\n", 7.0 / (2**16)
0.0001068115234375
=> nil
irb> printf "%.40g\n", 7.0 * (0.5**16)
0.0001068115234375
=> nil
irb> printf "%.40g\n", 7.0 / (2**32)
1.629814505577087402343750e-09
=> nil
irb> printf "%.40g\n", 7.0 * (0.5**32)
1.629814505577087402343750e-09
=> nil
```

2 基本的な制御構造

2.1 実行の流れと制御構造

ここまでに出てきたアルゴリズムおよびプログラムはすべて「1 本道」、つまり上から順番に実行して一番下まで来たらおしまい、というものでした。単純な計算ならそれでも問題ありませんが、手順が複雑になってくると、実行の流れをさまざまに切り換えていくことが必要になります。この、実行の流れを切り換える仕組みのことを、一般に**制御構造** (control structure) と呼びます。

制御構造の表現方法の 1 つに**流れ図** (flowchart) があります。流れ図では、図 1 にあるような「処理を示す箱」「条件による枝分かれを示す箱」などを矢線でつなげて実行の流れを表現します。流れ図は一見分かりやすそうですが、作成に手間が掛かる、場所をとる、ごちゃごちゃの構造を作ってしまいがち、という弱点のため、今日のソフトウェア開発ではあまり使われません (このため本資料でも、流れ図の代わりに擬似コードを主に用います)。

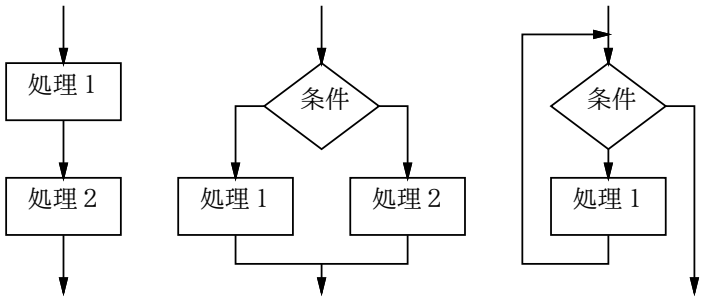

図 1: 3 つの基本的な制御構造

アルゴリズムを記述する時にはさまざまな実行の流れを組み立てますが、今日ではそれらの実行の流れは、図 1 に示す 3 つの制御構造を組み合せる形で作り出していくのが普通です。

- **順次実行**ないし連接— 動作を順番に実行していくこと。
- **枝分かれ**ないし分岐 — 条件に応じて 2 群の動作のうちから一方を選んで実行すること。
- **繰り返し**ないし反復 — 条件が成り立つ限り一群の動作を繰り返し実行すること。[4]

なぜこの 3 つが基かというと、「どんなにごちゃごちゃの流れ図でも、それと同等の動作を、この 3 つの組み合わせによって作り出せる」という定理があり、そのためにこの 3 つさえあればどのような処理の流れでも表現可能だからです。連接については単に動作を並べて書いたものは並べた順番に実行される、というだけなので、以下では残りの 2 つの制御構造をコード上で表現するやり方と、それらを組み合わせてアルゴリズムを組み立てていくやり方を学びます。

2.2 枝分かれと if 文 exam

上述のように、枝分かれとは、条件に応じて 2 群の動作のうちから一方を選んで実行するものです。擬似コードでは枝分かれを次のように書き表すものとします (「動作 2」が不要なら「そうでなければ」も書かなくてもかまいません)。

- もし ～ ならば、
- 　　動作 1。
- そうでなければ、
- 　　動作 2。
- 枝分かれ終わり。

Ruby ではこれを **if 文** (if statement) を使って表します (右側は「動作 2」のない場合です)。

```
if 条件 then
  ... 動作 1 ...
else
  ... 動作 2 ...
end
```

```
if 条件 then
   ... 動作 1 ...
end
```

`then` は Ruby では省略できますが、ただし「動作 1」を条件と同じ行に書く場合には省略できません。

「条件」については、当面は次の形のものがあると思っておいてください。

- 比較演算 — 「`x > 10`」等、2 値を比べるもの。**比較演算子**としては次の 6 種類がある。[5]

[4] 実行の流れを図示すると環状になるので、**ループ** (loop) とも呼びます。
[5] Ruby では「`!`」は「否定」を表すのに使っています。階乗の記号ではないので注意してください。

>	>=	<	<=	==	!=
より大	以上	より小	以下	等しい	等しくない

- 条件の組み合わせとして次が使える。[6] これらを「()」でくくったり複数組み合わせられる。

かつ (両方が成立)	または (最低限一方が成立)	否定 (〜でない)
条件1 && 条件2	条件1 || 条件2	! 条件1

では例として、「入力 x の絶対値を計算する」ことを考えます。擬似コードを示しましょう。

- abs1: 数値 x の絶対値を返す
- 　もし $x < 0$ ならば、
- 　　$result \leftarrow -x$。
- 　そうでなければ、
- 　　$result \leftarrow x$。
- 　枝分かれ終わり。
- 　$result$ を返す。

考え方としては簡単ですね? これを Ruby にしてみましょう。

```
def abs1(x)
  if x < 0
    result = -x
  else
    result = x
  end
  return result
end
```

実行の様子も示しておきます (0 もテストしていることに注意。作成したコードをテストするときには系統的に洩れなく試してみることが大切です)。

```
irb> abs1 8          ←正の数の絶対値は
=> 8                 ←元のまま
irb> abs1 -3         ←負の数であれば
=> 3                 ←正の数になる
irb> abs1 0          ← 0 の場合も
=> 0                 ←元のまま
irb>
```

ところで、同じ絶対値のプログラムを次のように書いたらどうでしょうか?

- abs2: 数値 x の絶対値を返す
- 　もし $x < 0$ ならば、
- 　　$-x$ を返す。
- 　そうでなければ、
- 　　x を返す。
- 　枝分かれ終わり。

[6] Ruby ではさらに演算子として and、or、not も使えますが、結合の強さが記号版と違っていて混乱しやすいので、本資料では使っていません。

Ruby 版は次のようになります。

```
def abs2(x)
  if x < 0
    return -x
  else
    return x
  end
end
```

先のとどちらが好みでしょうか? また、別のバージョンとして次のものはどうでしょうか?

- abs3: 数値 x の絶対値を返す
- 　$result \leftarrow x$。
- 　もし $x < 0$ ならば、
- 　　$result \leftarrow -x$。
- 　枝分かれ終わり。
- 　$result$ を返す。

Ruby プログラムも示します (`if` を 1 行に書いてみました。このような時は `then` が必須)。

```
def abs3(x)
  result = x
  if x < 0 then result = -x end
  return result
end
```

3 つのプログラムについて、あなたはどれが好みだったでしょうか?

一般に、プログラムの書き方は「どれが絶対正解」ということはなく、場面ごとに何がよいかが違ってきますし、人によっても基準が違うところがあります。ですから、皆様がこれからプログラミングを学習するに当たっては、自分なりの「よいと思う書き方」を発見していく、という側面が大いにあります。そのことを心に留めておいてください。

演習 1 絶対値計算プログラムの好きなバージョンを打ち込んで動かせ。

演習 2 枝分かれを用いて、次の動作をする Ruby プログラムを作成せよ。

a. 2 つの異なる実数 a、b を受け取り、より大きいほうを返す。

b. 3 つの異なる実数 a、b、c を受け取り、最大のものを返す。(やる気があったら 4 つでやってみてもよいでしょう。)

c. 実数を 1 つ受け取り、それが正なら「`'positive'`」、負なら「`'negative'`」、零なら「`'zero'`」という文字列を返す。

2.3 繰り返しと while 文 exam

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、繰り返しの最も一般的な形である、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すものとします。[7]

[7]「〜」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。

- ～ 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
  ... 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。

while を使った簡単な例として、1.0 を繰り返し 2 で割って行きその結果を表示する、というものを示しましょう。

```
def testdiv2
  x = 1.0
  while x > 0.0 do
    puts(x)
    x = x / 2.0
  end
end
```

無限に繰り返すような気がしますか? 次々に半分にしていくと、最後は浮動小数点表現で表せる限界の小さい数になって、さらに半分にすると近似値として「0.0」になるので止まるわけです。

```
irb> testdvi2
1.0
0.5
...
2.0e-323
1.0e-323
5.0e-324
=> nil
```

IEEE754 浮動小数点表現では、0 でない最も小さい数はおよそ「10^{-324}」くらいだと分かります。

3 数値積分

3.1 数値的に定積分を求める exam

もっと有用な繰り返しの具体的な題材として、**数値積分** (numerical integration — 定積分の値を数値を計算する方法で求めること) を取り上げてみましょう。皆様はこれまで、定積分を求めるのに、積分の公式を覚えたり、公式のあてはめや変形に苦労したりされてきたと思います。しかしプログラムを使えば、元の関数から直接定積分の値を計算してしまえるのです。

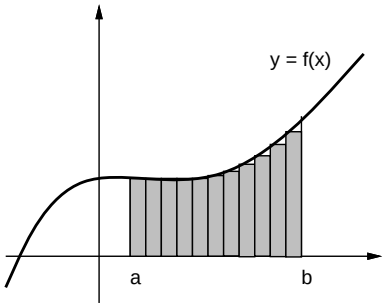

図 2: 数値積分の原理

関数 $y = f(x)$ の $x = a$ から $x = b$ までの定積分は図 2 のように、その関数のグラフを描くと、区間 $[a, b]$ の範囲における関数の下側の面積でした。そこで、図 2 にあるように、その部分に多数の細長い長方形を詰めて、その面積を合計すれば知りたい面積の値、つまり定積分の値が求まります。各長方形の幅は区間を n 等分した値 dx、高さは $f(x)$ の値なので、面積は容易に計算できます。

この方法であれば、$f(x)$ が数式として不定積分が求められなくても、定積分が計算できるます。数式の形で一般的に解を求めることを**解析的** (analytical) に解くと言い、これと対比して数値で計算して特定の問題の解を求めることを**数値的** (numerical) に解くと言います。

とはいえ、今は「正しい」値が求まるかどうかチェックしたいので、簡単な関数 $y = x^2$ でやってみます。不定積分は $\frac{1}{3}x^3$ ですから、区間 $[a, b]$ の定積分は $[\frac{1}{3}x^3]_a^b$ ということになります。たとえば $[1, 10]$ だったら $\frac{1000}{3} - \frac{1}{3} = \frac{999}{3} = 333$ となります。ではアルゴリズムを作ってみましょう。

- integ1: 関数 x^2 の区間 $[a, b]$ の定積分を区間数 n で計算
- $dx \leftarrow \frac{b-a}{n}$。
- $s \leftarrow 0$。
- $x \leftarrow a$。
- $x < b$ が成り立つ間繰り返し、
- $y \leftarrow x^2$。 # 関数 $f(x)$ の計算
- $s \leftarrow s + y \times dx$。
- $x \leftarrow x + dx$。
- 繰り返し終わり。
- s を返す。

すなわち、x にまず a を格納しておき、繰り返しの中で $x \leftarrow x + dx$、つまり x に dx を足した値を作ってそれを x に入れ直すことで x を徐々に (dx きざみで) 動かしていき、b まで来たら繰り返しを終わります。このように、繰り返しでは「こういう条件で変数を動かしていき、こうなったら終わる」という考え方が必要なのです。面積のほうは、s を最初 0 にしておき、繰り返しの中で細長い長方形の面積を繰り返し加えていくことで、合計を求めます。

では Ruby プログラムを示しましょう。「#」の右側に書かれている部分は注記ないし**コメント** (comment) と呼ばれ、Ruby ではこの書き方でプログラム中に覚え書きを入れておくことができます。コー

ドの意味が分かりづらい (何のためにこのような計算をしているのか読み取りにくい) 箇所には、その意図を注記しておくようにしてください。また、一時的に命令を実行しないようにするのにも、コメントが便利に使えます。これを**コメントアウト** (comment out) と呼びます。この例でも後で使うコードをコメントアウトしてあります。

```
def integ1(a, b, n)
  dx = (b - a).to_f / n
  s = 0.0
  x = a
# count = 0
  while x < b do
    y = x**2        # 関数 f(x) の計算
    s = s + y * dx
    x = x + dx
#   count = count + 1
#   puts("count=#{count} x=#{x}")
  end
  return s
end
```

2 行目の「(b - a).to_f」というのは、b - a を計算した後、その結果を実数に変換するメソッドです。a も b も n を整数で指定された場合、切り捨て除算されると dx が正しくならないので、このようにしました。それも含め、やっていることは先の擬似コードそのままだと分かるはずです。さて、333 が求まるでしょうか? 実行させてみます。

```
irb> integ1 1, 10, 100
=> 337.557149999999            ←ふーん?
irb> integ1 1, 10, 1000
=> 332.554621500007            ←小さい
irb> integ1 1, 10, 10000
=> 333.045451214912            ←大きい…
```

なんだかヘンですね。そこで、繰り返しの回数がいくつになっているかをチェックすることにして、上の行頭の「#」を削って動かし直してみました。[8]

```
irb> integ1 1.0, 10.0, 100
...
count=98 x=9.81999999999999  ←誤差が…
count=99 x=9.90999999999999
count=100 x=9.99999999999999
count=101 x=10.09            ← 101 回目が…
=> 337.557149999999          ←このため多い
```

区間数 100 個なのに長方形を 1 個余計に加えたため、値が大きすぎるわけです。なぜこんなことが起きるのでしょう? それは「$x \leftarrow x + dx$ で x を増やしていき b になったらやめる」というアルゴリズムの問題なのです。そもそもコンピュータでの浮動小数点計算は近似値の計算なので、dx を区間長の $\frac{1}{100}$ にしたとしても、そこに誤差があります。このため 100 回足してもわずかに b より小さい場合があり、その時は余分に繰り返しを実行してしまいます。

[8]つまり、変数 count に回数を数えつつ x を表示するようにするわけです。このように、コメントアウトしてあったコードを活かして動かすことを「コメントアウトを外す」とも言います。

3.2 計数ループ exam

ではどうすればよいでしょうか。繰り返し回数を 100 回と決めているのですから、回数を数えるのは整数型で行い、[9] それをもとに各回の x を計算するのがよいのです。つまり、次のようなループを書くことになります (**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。[10] これはたとえば次のようになります。

```
100.times do
  ...
end
```

この `times` も先の `to_f` などのように「値 x に対して何かをする」メソッドですが、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 100 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。[11]

ところで、計数ループの中でカウンタの値 $(0, 1, 2, \ldots)$ を使いたいこともあります。このため、`times` は各繰り返しごとにカウンタ値をブロックにパラメタとして渡してくれます。上の例ではそれを受け取っていませんでしたが、ブロックの冒頭に「`|`名前`|`」という書き方でパラメタ (の列) を指定することで、このパラメタを受け取ることができます。複数ある場合は`|x, y|`のようにカンマで区切って並べます。たとえば次のようにすると、0 から 99 までの数を次々に出力することができます。

```
100.times do |i|
  puts(i)
end
```

いろいろありましたが、元に戻って擬似コードでは、計数ループを次のように記します。[12]

[9] 整数ならば、あふれない限り誤差はありません。

[10] 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。

[11] 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。

[12] 擬似コードはあくまでも「擬似」コードであり、Ruby に直した時に for 文か `times` かは特に指定しません。

- 変数 i を 0 から n の手前まで変えながら繰り返し、
- 　　... # ループ内の動作
- 繰り返し終わり。

3.3　計数ループを用いた数値積分 exam

余談が終わったので、先の積分プログラムを計数ループを使うように直してみましょう。

- integ2: 関数 x^2 の区間 $[a, b]$ の定積分を区間数 n で計算
- 　　$dx \leftarrow \frac{b-a}{n}$。
- 　　$s \leftarrow 0$。
- 　　変数 i を 0 から n の手前まで変えながら繰り返し、
- 　　　　$x \leftarrow a + i \times dx$。
- 　　　　$y \leftarrow x^2$。 # 関数 $f(x)$ の計算
- 　　　　$s \leftarrow s + y \times dx$。
- 　　繰り返し終わり。
- 　　s を返す。

先の例との違いは、毎回 x を i から計算する点です。これを Ruby にしたものも示します。

```
def integ2(a, b, n)
  dx = (b - a).to_f / n
  s = 0.0
  n.times do |i|
    x = a + i * dx
    y = x**2        # 関数 f(x) の計算
    s = s + y * dx
  end
  return s
end
```

これを動かしてみましょう。

```
irb> integ2 1.0, 10.0, 100
=> 328.55715
irb> integ2 1.0, 10.0, 1000
=> 332.5546215
irb> integ2 1.0, 10.0, 10000
=> 332.955451215
```

こんどはきざみを小さくすると順当に誤差が減少していきます。

しかし、常に正しい面積である 333 より小さいようですが、これはなぜでしょうか? それは、長方形の面積を計算するのに微小区間の左端の x を使って高さを決めるため、増大関数では図 3 のように微小な三角形の分だけ面積が小さめに計算されるからです (逆に減少関数だと大きめに計算されます)。これをもうちょっと何とかする方法については、演習問題にしたので、やってみてください。

演習 3　上の演習問題のプログラムを打ち込んで動かせ。動いたら「減少する関数だと値が大き目に出る」ことも確認せよ。できれば、左端ではなく右端で計算するのもやってみるとよい。その後、次のような考え方で誤差が減少できるかどうか、実際にプログラムを書いて試してみよ。

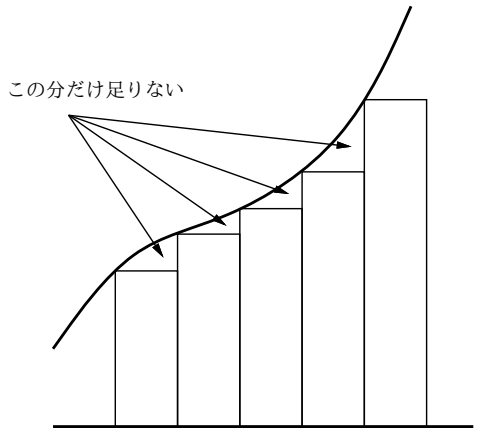

図 3: 区間の左端を使う場合の誤差

a. 左端の x だけでも右端の x だけでも弱点があるので、両方で計算して平均を取る。
b. 左端や右端だからよくないので、区間の中央の x を使う。
c. 上記 a と b をうまく組み合わせてみる。

演習 4 次のような、繰り返しを使ったプログラムを作成せよ。

a. 非負整数 n を受け取り、2^n を計算する。
b. 非負整数 n を受け取り、$n! = n \times (n-1) \times \cdots \times 2 \times 1$ を計算する。
c. 非負整数 n と $r(\leq n)$ を受け取り、${}_nC_r$ を計算する。

$$ {}_nC_r = \frac{n \times (n-1) \times \cdots \times (n-r+1)}{r \times (r-1) \times \cdots \times 1} $$

d. x と計算する項の数 n を与えて、次のテイラー展開を計算する。

$$ sinx = \frac{x^1}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots $$

$$ cosx = \frac{x^0}{0!} - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots $$

実際に値の分かる x を入れて精度を確認してみること。$\pm 10\pi$ とかだとどうか? n はいくつくらいが適切か?

4 制御構造の組み合わせ

少し込み入ったプログラムになると、ある制御構造 (枝分かれ、繰り返し) の内側にさらに制御構造を入れることになります。たとえば、

- もし〜であれば、
- 　条件〜が成り立つ間繰り返し、
- 　　○○をする
- 　以上を繰り返し。
- 枝分かれ終わり。

だと次のようになるわけです。

```
if ...
    while
        ...
    end
end
```

このように規則に従って要素を組み合わせて行くことで (単に並べるのも組み合わせ方のうち)、いくらでも複雑なプログラムが作成できます。これはちょうど、簡単な規則と単語からいくらでも複雑な文章が (日本語や英語で) 作れるのと同じです。

演習 5 a と b の最大公約数を $gcd(a,b)$ と記す。正の整数 x、y の $gcd(x,y)$ を求めることを考える。

- $x = y$ のとき、$gcd(x,y) = x = y$。
- $x > y$ のとき、$gcd(x,y) = gcd(x-y,y)$。
- $x < y$ のとき、$gcd(x,y) = gcd(x,y-x)$。

これを利用して、2 つの正の整数 x、y に対してその最大公約数を求めるアルゴリズムの疑似コードを書き、Ruby プログラムを作成せよ (なぜこれで求まるかも説明すること)。

演習 6 「正の整数 N を受け取り、N が素数なら `true`、そうでなければ `false` を返す Ruby プログラム」を書け。[13] まず疑似コードを書き、次に Ruby に直すこと。(ヒント: N が素数ということは、N を $2 \sim N-1$ のいずれで割っても余りが出るということ。剰余は演算子「`%`」で計算できる)。

演習 7 「正の整数 N を受け取り、N 以下の素数をすべて打ち出す Ruby プログラム」を書け。待ち時間 10 秒以内でいくつの N まで処理できるか調べて報告せよ。(もちろん N が大きくなるように工夫してくれるとなおよい。)

本日の課題 2A

「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. 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 2B

「演習 2」、「演習 4〜演習 7」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. 枝分かれや繰り返しの動き方が納得できましたか?

Q2. 枝分かれと繰り返しのどっちが難しいですか? それはなぜ?

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。

[13] `true`/`false` は「はい/いいえ」を表す値である。