# 情報科学　2011　久野クラス　＃2

久野　靖*

2011.10.14

## はじめに

1 週間のごぶさたでしたが、演習とかやってみていかがでしたか? アンケートで色々感想を頂いていますが、まあそれなりという感じのようですので、今後とも同様に行きたいと思います。無理に難しい課題をやらなくても、自分の実力に合ったものでいいので、よろしくお願いします。さて、2 回目の目標は次の通り:

- 基本的な制御構造 (枝分かれ、繰り返し) について理解し、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/プログラムが考えられるようになる。

ところで、皆様が提出されたプログラムを拝見すると、

```
def test
.....
.....
.....
end
```

と、偏平足のように平らなのがありますが、これはやめた方がいいです。というのは、これから沢山 def とかその他の (今日出て来る) 構造が入ってくると、ちゃんと字下げしてないと「どこからどこまでの範囲か」が分からなくなりますから。ぜひこのように。

```
def test
  .....
  .....
  .....
end
```

何文字下げるとか、そのあたりの細かい流儀は自分で色々試して好みのものを見つけていただければ結構ですので。

## 1　整数と実数および結果の出力について

前回の課題の中に整数と実数の違いおよび実数の誤差に関するものがあったのですが、また桁数を多く表示するために printf を使うという説明もあったのですが、このあたり説明もれがあったりして (すみません)、あと誤解も生じやすいところなので、最初に補足説明をしておきます。

前回述べたように、コンピュータでは整数と実数はまったく別のものです。Ruby では整数は「誤差がなく」「桁数はいくらでも大きくなれる」(ただし桁数が大きくなると相応に演算が遅くなる) 形で表現されています。これを図 1 では六角形で表しました。

*筑波大学大学院経営システム科学専攻

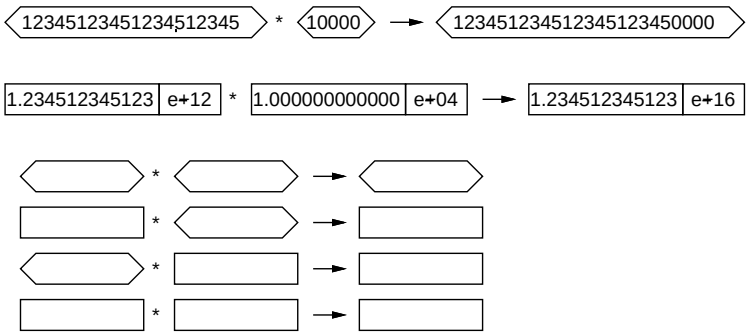


図 1: 整数と実数の違いおよび混合演算

一方、実数は仮数部と指数部に分けて表現されており、仮数部の桁数は一定なので、その桁数を超えた分は表現できません (つまり誤差がある)。これを図 1 では長方形で表しました。なお、図では十進表現で書かれていますが、実際には仮数も指数も 2 進表現されています。

そして、整数どうし、実数どうしの演算結果はそれぞれ整数、実数になりますが、整数と実数をまぜた場合は整数側が実数に変換されて実数計算され、結果は実数になります。

あと最後に 1 つ、「整数どうしの除算は切捨て除算」これも前回強調したところです。ところで誤差の出かたを調べるという課題に「切捨て除算による誤差」を挙げた人がいましたが、もともと切捨てるという演算を「あなたが指定」したわけですから、これは誤差とは言いませんし、この回答は題意に合ってないと考えます。

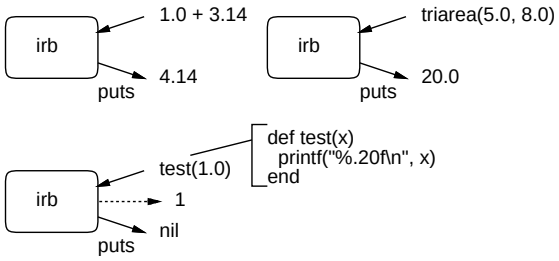


図 2: puts と printf による出力

次に、puts と printf による出力について改めて説明します。普通に irb を使って計算やメソッドの実行をした場合 (図 2 上)、計算した結果は「irb によって自動的に」出力されます。これは正確に言えば、irb が puts を使って計算した結果を画面に表示してくれているものです。

これに対し、プログラムの中に puts や printf を書いた場合は、プログラムの中でそこを「実行したときに」出力が起こります (図 2 下)。なお、メソッドの結果もこれまで通り irb によって表示されますが、今回の場合は出力が何もないことを示す値 nil が表示されるものと思います。

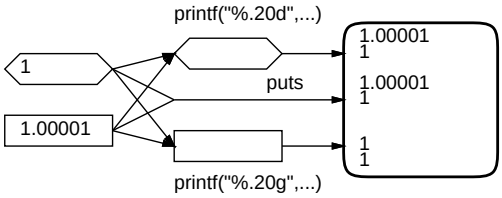


図 3: 内部表現と画面表示

では、puts と printf の違いは何かというと、puts ではさまざまなデータを Ruby に「おまかせ」で表示するものだということです。このため、小数点以下の桁数なども適宜丸められて表示

されます。それだと誤差の出かたを見るのには不便なので、実数を多くの桁数で表示するためには「表示形式を厳密に指定する機能」を持った printf を使うわけです。ちなみに「"%.20g\n"」というのは実数を 20 桁の精度で出力してから行かえをする」指定です。

ここで問題があったのですが、桁数の多い整数をそのまま打たせようとしてもこの指定だと 1 回実数に変換されてしまうので、誤差が入ってしまいます。桁数の多い整数を正確に打つには整数出力を指定した「"%.20d\n"」によるか、または単に puts を使うのがよかったわけです。

またまた最後に大事なことですが、「大事なのは内部のデータがどれだけの精度で計算されているか」であって、表示のときに何桁表示されてもそれは出力時の指定がたまたまそうだった、というだけです。たとえば、「3.140000000000」と表示されたからといって、円周率がこの値だとは思わないですよね。逆に内部で「3.1415926535897933」くらいの精度で計算されていても、おまかせで出力したら「3.1416」くらいしか表示されていないかも知れません。というわけで、内部の精度と表示されているものを区別して考えるようにしてください。

## 2 前回の演習問題の解答例 (一部)

### 演習 3a — 四則演算を試す

演習 3a はとりあえずは和の計算でした。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
```

動かしているところも見てみましょう:

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

四則つまり和、差、商、積の場合も上と同じにやればいいわけですが、和、差、商、積のために 4 つメソッドを作る代わりに 1 つで済ませるという方法を考えてみましょう (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
```

動かしているところは次のとおり:

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

なるほど 4 つの値が順次打ち出され、最後に `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** (character 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** (single quotation character) または**ダブルクォート** (double quotation character) 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
```

実行しているところは次のとおり:

```
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
```

実行しているところは次のとおり:

```
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。

### 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:

[1] ダブルクォートはその埋め込み機能のために**特殊文字** (special character) をさまざまに解釈しますから、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] さらに `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。

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

ちゃんとマイナスの時も試していただけたでしょうか? ここで「マイナスだとどうだろう」とか思うようになって頂きたいわけです。

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
```

こんどは 2 数ともマイナスの場合をまず考えて、それから等間隔で、というふうに考えるとよいかと思います。[3]

### 演習 3c — 円錐の体積

演習 3c 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 $\pi r^2$。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 (power) 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 (circle ratio) 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。[4]

### 演習 4 — 足し算の所要時間計測

演習 4 は「切捨て除算以外の」整数と実数の違いを試すというものでした。ヒントに載せたものをやってみます。足し算を直接書いてもいいのですが、定義したメソッド `add` を呼んでいます。

[3] 剰余演算の振舞いは、プログラミング言語によって多少の違いが見られる箇所です。

[4] 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 (base of natural logarithm) $e$ は `Math::E` で表せます。

```
irb> add(123451234512345, 1)
=> 123451234512346
irb> add(123451234512345.0, 1.0)
=> 1.23451234512346e+14
```

これはどちらも同じですね。実数は「おまかせ」だと指数記法になりますが、まあ値としては同じことです。では、12345 をもう 1 つ増やしてみます。

```
irb> add(12345123451234512345, 1)
=> 12345123451234512346
irb> add(12345123451234512345.0, 1.0)
=> 1.23451234512345e+19
```

整数は問題ないですが、実数はおまかせだと適当なところで丸められてしまっているので、違いが分かりません。そこで printf で桁数を増やして表示させてみます。

```
irb> printf("%.20g\n", add(12345123451234512345.0, 1.0))
12345123451234512896
=> nil
```

こうして見ると下 3 桁はまったく違うことが分かります。

で、理由ですが、皆様も色々試して頂いたと思いますが、実数の場合は仮数 (有効数字の部分) が 16 桁程度に固定されていますから、それより多い桁数の部分は無意味 (ゴミ) だ、ということです。

### 演習 5 — 実数計算の誤差

演習 4 は整数と実数の違いに関するものしたが、こちらは実数計算の誤差を観察する問題でした。当然、printf を使って十分な桁数を表示するようにします。いちいち関数を書かずに irb だけ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 桁程度だと分かる
=> 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 は 2 進表現では切りがよくないので
=> nil
irb> printf "%.20g\n", 7.0 * 0.1
0.70000000000000006661  ←値 0.1 も誤差を含むので上と違う値に
=> 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
```

### 演習 6 — 無限大と非数

演習 6 は±∞や NaN(非数) が表れる計算を作ってみなさい、と言うものでした。たとえば分母が 0 の数などはどうでしょう:

```
irb> 1/0
ZeroDivisionError: divided by 0
        from (irb):1:in `/'
        from (irb):1
irb>
```

おっと、整数どうしの除算では 0 で割るのはエラーですね。では実数にしてみます:

```
irb> 1.0/0.0
=> Infinity              ←確かに無限大
irb> -1.0/0.0
=> -Infinity             ←マイナスの無限大も作れる
irb> 1.0/0.0 + 1
=> Infinity              ←無限大に 1 足しても無限大
irb> 1.0 / (1.0/0.0)
=> 0.0                   ←有限の数を無限大で割ると 0
irb> 1.0 / (-1.0/0.0)
=> -0.0                  ←マイナス 0 というのもある!
irb> 1.0/0.0 - 1.0/0.0
=> NaN                   ←無限大どうしを引いたら NaN
 irb> 1.0/0.0 - 1.0/0.0 + 1
=> NaN                   ← NaN に何を演算しても NaN
```

「無限大ひく無限大」ですが、この無限大というのは特定の数があるわけではなく「大きすぎる」ということを表しているのであって、したがって「大きすぎる」ものどうしを引き算しても 0 ではなく「計算できない」ことを表す NaN になるわけです。-0.0 というのもヘンなようですが、「小さすぎて表せないが符号はマイナス」ということを表しているわけです。本文で説明したように、コンピュータ上の計算ではさまざまな限界がありますが、「限界を越えた」ことを表しつつできるだけ多くの情報を残すためにこのような特別な値や振舞いが用意されているわけです。

## 3　基本的な制御構造

前章で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。

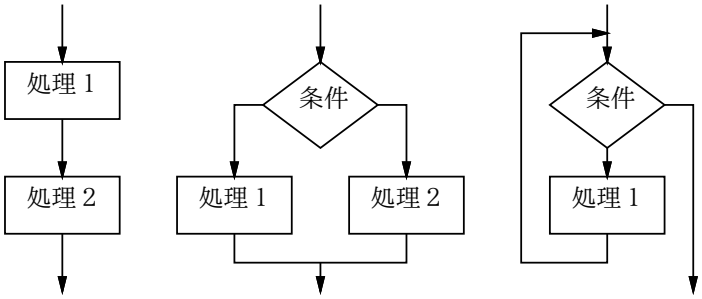


図 4: 3 つの基本的な制御構造

制御構造を表現する方法の 1 つに**流れ図** (flowchart) があります。流れ図では、図 4 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ることで、多様な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、不規則でごちゃごちゃの構造を作ってしまいやすい、という弱点があるため、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ています。

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 4 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** (sequenceing) — 動作を順番に実行していくこと。
- **枝分かれ**ないし**分岐** (branching) — 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** (repetition) — 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。[5]

なぜこの 3 つが基になるかというと、「どんなにごちゃごちゃの流れ図でも、その流れ図と同等の動作をするものを、この 3 つの組み合わせによって作り出すことができる」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。

## 4　枝分かれと if 文

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。

[5]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。

- 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) と呼ばれる文を使って表します (右側は「動作 2」のない場合です):

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

then は 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」のように 2 つの値を比べるもの。**比較演算子** (comparison operator) としては、>(より大)、>=(以上)、<(より小)、<=(以下)、==(等しい)、!=(等しくない) がある。[6]
- 条件の組み合わせ — **かつ** (and — ともに成り立つ) を表す「条件1 && 条件2」、**または** (or — 少なくとも片方は成り立つ) を表す「条件1 || 条件2」、**否定** (not — 〜でない) を表す「!条件1」が使える。[7]複数のかつ、または、否定を組み合わせたり、かっこでくくることもできる。

では具体的な例題として、「入力 $x$ の絶対値を計算する」ことを考えてみます。まず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):

[6]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。
[7]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。

```
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。

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。

「そうでなければ」の部分で何もすることがなければ「そうでなければ」以下を書かないでよいのでしたね。Ruby プログラムも示しておきます (`if` を 1 行に書いてみましたが、Ruby でもこのような時は `then` が必須です):

```
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。

## 5 繰り返しと while 文

ここまででは、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: [8]

- ～ 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このような繰り返しは while というキーワードを用いて表すので、このような条件を指定した繰り返しのことを **while ループ** (while loop) とも呼びます。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。

[8] 「～」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。

# 6　数値積分

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。

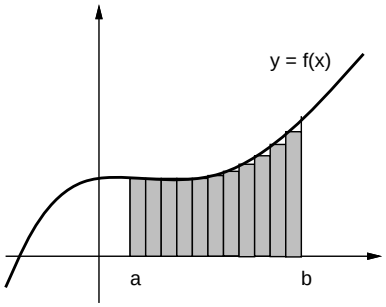


図 5: 数値積分の原理

関数 $y = f(x)$ の $x = a$ から $x = b$ までの定積分というのは図 5 のように、その関数のグラフを描いたとして、区間 $[a, b]$ の範囲における関数の下側の面積なのでしたね。そこで、図 5 にあるように、その部分に多数の細長い長方形を詰めてみて、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ることになります。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、その面積を計算するのは簡単です。

この方法であれば、$f(x)$ が数式の形で不定積分が求められないような 関数であっても、問題なく定積分が計算できることになります。(数式の形で一般的に問題の解を求めることを**解析的** (analytical) に解く、と言います。これに対して、数値で計算して特定の問題の答えを求めることを**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) / 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
```

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 1.0, 10.0, 100
=> 337.557149999999            ←ふーん?
irb> integ1 1.0, 10.0, 1000
=> 332.554621500007            ←小さい
irb> integ1 1.0, 10.0, 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: [9]

```
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。

[9] 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。

# 7　計数ループ

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、[10]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を数えながら繰り返していくような繰り返しを**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** (for loop) 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。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。これはたとえば次のようになります:

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

これまでの例ではメソッドは単独で出てきていましたが、この例のように「$x$`.`メソッド名」という形で呼び出すことで「値 $x$ に対して何かをする」ようなメソッドもこれから多数出てきます。数値に対して呼び出すメソッド `times` は、**ブロック**を引数として受け取り、そのブロックを数値の回数 (上の例では 100 回) 実行します。Ruby ではブロックは一群のコードを `do` … `end` で囲むことで指定します。ブロックの指定のための `do` は省略できません。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。

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:

[10]整数ならば、あふれない限り誤差はありません。

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

ところで、この times と同じようなブロックを受け取るメソッドを、自分でも定義できます。ブロックを受け取るメソッドを定義する時は、パラメタ列の最後に「&パラメタ名」という特別な指定を行うことで、このパラメタでブロックを受け取れます。メソッド内から渡されたブロックを実行させるには **yield 文** (yield statement) を使用します。この時必要ならブロックのパラメタ値を指定できます。times と同様の働きをするメソッドを自分で定義するとしたら、たとえば次のようになります:

```
def mytimes(count, &block)
  for i in 0..count-1 do
    yield i
  end
end
```

これを使ったループはたとえば次のようになります。書き方がさっきとちょっと違いますが、これは標準の times が整数にくっついたメソッドなのに対し、mytimes は自分が定義しているメソッドで整数にくっついていないので、パラメタをかっこ内に指定する必要があるからこうなるわけです。

```
mytimes(100) do |i|
  puts(i)
end
```

いろいろな話がありましたが、元に戻って以下の擬似コードでは、計数ループを次のように記します: [11]

- 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。

## 8 数値積分 (つづき)

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:

- integ1: 関数 $x^2$ の区間 $[a, b]$ の定積分を区間数 $n$ で計算
- $dx \leftarrow \frac{b-a}{n}$。
- $s \leftarrow 0$。
- 変数 $i$ を 0 から $n$ の手前まで変えながら繰り返し、
- $x \leftarrow a + \frac{i}{n}(b-a)$。
- $y \leftarrow x^2$。 # 関数 $f(x)$ の計算
- $s \leftarrow s + y \times dx$。
- 繰り返し終わり。
- $s$ を返す。

先のプログラムと違うのは、毎回 $x$ を $i$ から計算しているところです。では、これを Ruby にしたものを示します:

[11] 擬似コードはあくまでも「擬似」コードであり、これを Ruby で書く時に for 文にするか times にするかは特に指定しません。

```
def integ2(a, b, n)
  dx = (b - a) / n;
  s = 0.0
  n.times do |i|
    x = a + i * (b - a) / n;
    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?

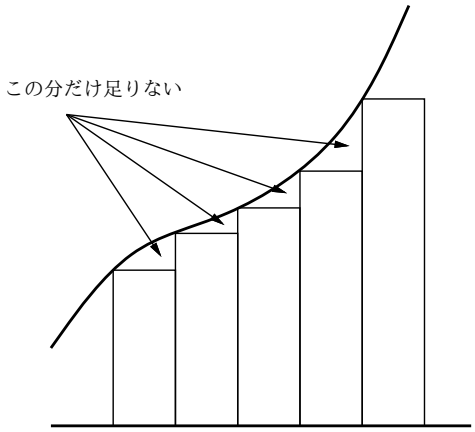


図 6: 区間の左端を使う場合の誤差

それは、長方形の面積を計算するのに微小区間の左端の $x$ を使って高さを求めているため、値が増大していく関数では図 6 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てしまうからです (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。

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. 整数 $n$ を受け取り、$2^n$ を計算する。

b. 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. 整数 $n$ と整数 $r(\leq n)$ 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?

## 9 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** 2 数 $a$、$b$ の最大公約数を $gcd(a,b)$ と記すことにする。正の整数 $x$、$y$ について $gcd(x,y)$ を求めることを考える。以下 (☆) に注意。

- $x = y$ 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 $x$、$y$ を読み込み、その最大公約数を出力するアルゴリズムの疑似コードを書き、それを実現した Ruby プログラムを作成せよ (☆の簡単な証明も書くこと)。

**演習 6** 「正の整数 $N$ を受け取り、$N$ が素数か否かを返すれる Ruby プログラム」を書け。まず疑似コードを書き、それから Ruby に直すこと。(ヒント: $N$ が素数ということは、$N$ を 2〜$N-1$ 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。たとえば「`7 % 4`」は 3。)

**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。)

## A 本日の課題 2A

今日は「演習 2」で動かしたプログラムを含む小レポートを久野まで電子メールで送ってください。具体的な内容は次の通り。

1. Subject: は「Report 2A」とする。
2. 学籍番号、氏名、ペアの**学籍番号**(または「個人作業」)、投稿日時を書く。
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 (簡単でよい)。

   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。

## B 次回までの課題 2B

次回までの課題は「演習 2」〜「演習 7」までの (小) 課題からプログラムを 2 つ以上作ること。ただし「演習 2」から選ぶのは最大 1 個とする。

レポートは授業開始 10 分前 (10:30) 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。具体的な内容は次の通り。

1. Subject: は「Report 2B」とする。
2. 学籍番号、氏名、ペアの**学籍番号**(または「個人作業」)、投稿日時を書く。
3. 選んだプログラム 1 つのソース。
4. 説明と考察。
5. 選んだプログラムもう 1 つのソース。
6. 説明と考察。
7. 下記のアンケートの回答。

   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. 課題に対する感想と今後の要望をお書きください。