情報科学　2011　久野クラス　＃3

久野　靖*

2011.10.21

はじめに

今回はまずメソッドと return の話を少し説明してから、前回の課題の解説と併せて、数値積分や制御構造などで追加すべき点を説明します。制御構造の組み合わせは重要なので、演習して頂くようにします。その後本日の新しい内容としては、次のものを取り上げます。

- $f(x) = 0$ の求解
- 基本データ型、配列型とその利用

とくに配列を学ぶとデータを沢山保持しておけるようになるので、実際に「お仕事で計算する」時に役立つと思います。

1　メソッドと return

皆様の演習している様子を拝見していると、return、def、end などが「極めて特別」であることを意識されていない方が多いようです (あまり説明してなかったので当然ですね)。少しこのあたりに絞って復習しておきましょう。

まず、私たちが「Ruby プログラムを書く」ときは、いくつかの動作をまとめて「名前をつけ」ますよね。これが「メソッド」です。

```
def 名前(パラメタ, ...)
  ...
  ...  ←この部分に動作の中身(アルゴリズム)を書く
  ...
end
```

メソッド定義はこのように「def」(definition の意味) から「end」までの範囲として書くので、このどちらかでも忘れていると (または 1 文字でも綴り間違うと) うまく行きません。

我々は自然言語に慣れているので「def でも define でも definition でも同じじゃん」という感覚がありますが、Ruby ではこれらは「まったく別」です。たとえば、define という名前の変数を作ることはできますが、def という名前の変数は作れません。こういう特別扱いな語のことを**予約語** (reserved words) と言います。

あと、end は if や while や do の終わりにも使うので、「end の数が足りない (多い)」という間違いもよくあります。もちろん、ちゃんと対応が合っていないと思った通りになりません。このために、字下げをきちんと書くことを勧めています。

さて次に、定義したメソッドは次のように名前とパラメタをつけて「呼び出す」ことができます。パラメタを指定できるようにすることで、さまざまな値について計算できるようになる、ということは説明しましたね。図 1 のように、パラメタを指定すると、それは def のところで書い

*筑波大学大学院経営システム科学専攻

たパラメタにそれぞれ対応づけられて、定義本体の中で使われます。この場合は、8 と 3 を渡したので、x が 8、y が 3 になるわけです。

では、`return` は? これは、呼び出されたメソッドから値を返すのに使います。[1]

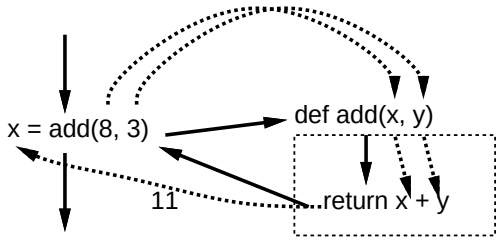

図 1: メソッドと return

そして、`return` の役割は「このメソッドの計算を終了させて、結果を呼び出したところに返す」ことです。図 1 の場合、結果 11 が呼び出し元に返されて、それが呼び出し元の動作として変数 x に代入されます (この x とパラメタの x は別の x だということにも注意)。そういうわけで `return` を複数回書くことはできないわけです。[2]

2 前回演習問題の解説

2.1 演習 2a — 枝分かれの復習

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

[1] Ruby では `return` を書かなかった場合、「一番最後に計算した値が返される」ことになっています。個人的にはそれだと何が返るかがはっきりしないので、きちんと `return` を書いてもらうようにしています。

[2] では、複数の値を表示させたい場合は? それは前にやったように、`puts` などの出力命令を使えばそれを実行したところで指定した値が画面に表示されます。

- $result \leftarrow a$。
- 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 (久野) 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。

2.2 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。
- $result$ を出力する。

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。

2.3 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
```

しかし、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.」と出力。
- そうでなくて $x < 0$ ならば、
- 　　「negative.」と出力。

- そうでなければ、
- 　　「zero.」と出力。
- 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。

ところで、最大値の問題にちょっと戻ると、複合条件を使えば「`a > b && a > c`」なら`a`が最大だと分かりますから、これを利用した 3 方向枝分かれで書くこともできます (変数を使わず値を返すスタイルにしてみました):

```
def max3y(a, b, c)
  if a > b && a > c
    return a
  elsif b > c
    return b
  else
    return c
  end
end
```

ただし、この方法でも N が 4、5 と増えてくると条件の中の比較演算が増えて、一般に N^2 に比例してしまいます。だからいけないというわけではなく、N の個数が多くなければ、このやり方を使ってもよいかも知れません。

2.4　演習 3a〜3c — 数値積分

長方形の高さとして区間の左端の $f(x)$ を使うと増大関数で値が小さくなり、右端の $f(x)$ を使うと大きくなるので、「左端と右端の平均を取って」みるという課題でした (減少関数だと逆に大きく/小さくなります)。これは考えてみると、面積を計算するのにその区間の関数を直線で補間した「台形」を考え、その面積を求めているのと同等です。このため、これを数値積分の**台形公式** (trapezoid rule) と呼びます。

台形公式の計算内容は次のようになります (区間の幅を d で表す):

$$s = \sum \frac{1}{2}\{f(x) + f(x+d)\}d$$

これを計算する Ruby プログラムを示しておきます:

```
def integ3(a, b, n)
  dx = (b - a) / n
  s = 0.0
  n.times do |i|
    x = a + i * (b - a) / n
    y0 = x**2
    y1 = (x+dx)**2
    s = s + 0.5*(y0+y1) * dx
  end
  return s
end
```

台形公式は直線による補間なので、曲線が上に凸だと値は小さく、下に凸だと値は大きくなります。一方、これも演習にありましたが、区間の中央の x を使って長方形で計算すると (これを**中点**

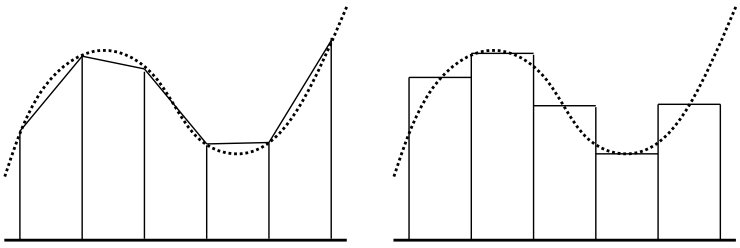

図 2: 台形公式と中点公式

公式と言います)、逆に上に凸だと大きく、下に凸だと小さくなります (図 2)。だからこれをちょうどよく混ぜたらよいのでは、というのが演習 3c になっていたわけです。実は、左端:中央:右端を 1:4:1 で混ぜると (つまり台形:中点を 1:2 で混ぜると) よい結果が得られます。プログラムも示しておきます:

```
def integ4(a, b, n)
  dx = (b - a) / n
  s = 0.0
  n.times do |i|
    x = a + i * (b - a) / n
    y0 = x**2
    y1 = (x+0.5*dx)**2
    y2 = (x+dx)**2
    s = s + (y0+4*y1+y2) * dx / 6.0
  end
  return s
end
```

実際に計算させてみましょう (「正解」は 333 だったことに注意):

```
irb> integ4 1.0, 10.0, 100
=> 333.0                ←ぴったり? 本当か?
irb> printf "%.20g\n", integ4(1.0, 10.0, 100)
332.99999999999994316  ← 20 桁表示---確かにすごくよい値
=> nil
irb> printf "%.20g\n", integ4(1.0, 10.0, 10)
333                     ←分割数を 10 に減らしたら逆にぴったり?
=> nil
```

この計算式はシンプソンの公式 (Sympson rule) と言われ、数値積分では標準的な方法です: [3]

$$s = \sum \frac{1}{3}\{f(x) + 4f(x+d) + f(x+2d)\}d$$

なぜこれがよいかというと、当該区間を 2 次曲線で補間することになるからです。だから積分しようとしている関数が 2 次以下の多項式だと「ぴったし」になり、そのため上の例では区間数が少ないほど (誤差が出ないため) よかったわけです。実際、分割数 1 でもぴったりなので、もはや数値積分と言えないような…

2 次式の補間になる理由を示しておきます。当該区間の曲線を 2 次式

$$y = ax^2 + bx + c$$

[3] この式では見やすくするため区間の半分を d としていて、そのためプログラムの 6 で割る代わりに 3 で割っています

で表せるものとします。また区間の幅を $2d$、左端を x_0、中央を $x_1 = x_0 + d$、右端を $x_0 + 2d$、対応する関数値を y_0、y_1、y_2 とおきます。上の 2 次式の不定積分は $\frac{1}{3}ax^3 + \frac{1}{2}bx^2 + cx$ ですから、面積 (定積分) は次のようになります:

$$s = \left[\frac{1}{3}ax^3 + \frac{1}{2}bx^2 + cx\right]_{x_0}^{x_0+2d}$$

これを整理すると次のようになります:

$$3s = \{a(6{x_0}^2 + 12x_0d + 8d^2) + b(6x_0 + 6d) + 6c\}d$$

ところで

$$y_0 = a{x_0}^2 + bx_0 + c$$

$$y_1 = a(x_0 + d)^2 + b(x_0 + d) + c$$

$$y_2 = a(x_0 + 2d)^2 + b(x_0 + 2d) + c$$

なので、見比べると次の式が成り立つと分かります:

$$3s = (y_0 + 4y_1 + y_2)d$$

というわけで、上の式が出て来るわけです。

数値積分にはシンプソンの公式が一番よいのかというと、必ずしもそうとは言えません。たとえば、ある細かさで積分を計算して、もっと細かくするために d を半分にしたいと思ったとすると、台形公式では既に計算した値をとっておいて、新たに加えた半分ずつの点についての計算を追加すれば済みます。このような計算方法を**漸近的**と言います。漸近的に計算していき、値の変化がなくなったらこれ以上細かさを増やしても意味がないと判断してやめるというのは 1 つの方法です。

2.5 演習 4a〜4c — 繰り返し

この辺は簡単なのでプログラムだけ示します (べき乗は計算するだけなら `2**n` でよいのですが、繰り返しを使うという前提なのでループを使います):

```
def pow2(n)
  result = 1
  n.times do result = result * 2 end
  return result
end

def fact(n)
  result = 1
  n.times do |i| result = result * (i+1) end
  return result
end
```

組み合わせの数を整数で計算できるようにするためには「小さい側から」掛けて・割って・掛けて・割ってのようにしないとうまくいきません。$\frac{4\times5\times6\times7}{1\times2\times3\times4}$ のように並べて左から 1 列ずつ乗算・除算の順で計算するわけです。この順序でやれば、除算が常に割り切れるので、誤差なしで計算できます (浮動小数点で計算してしまうと、誤差が現れるのでいまいちだと思います):

```
def comb(n, r)
  result = 1
  r.times do |i|
    result = result * (n - r + i + 1) / (i + 1)
  end
  return result
end
```

2.6 演習 4d — テイラー級数で sin と cos を計算

これは「階乗や x^n を計算しつつ」足していくのでちょっと面倒ですね。しかも交互に+/-が変わることも扱う必要があります。

```
def sincos(x, n)
  sign = 1; pow = 1.0; fact = 1; sin = 0.0; cos = 0.0
  n.times do |i|
    cos = cos + sign * pow / fact
    pow = pow * x
    fact = fact * (2*i+1)
    sin = sin + sign * pow / fact
    pow = pow * x
    fact = fact * (2*i+2)
    sign = -sign
  end
  return [sin, cos]
end
```

このプログラムでは、べき乗の数を 2 ずつ増やしながら sin と cos のテイラー展開を並行して計算しています。計算式を再録しておきましょう:

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

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

「何項まで計算するか」によって精度が変わって来ますが、言い換えれば実用のプログラムでは項を無限に計算することはできず、どこかで打ち切る必要があります。ということは、打ち切ったその先の項の値のぶんは無視されて誤差となわけです。これを**打ち切り誤差** (cutoff error) といい、既に学んだ丸め誤差、情報落ち、桁落ちと並んで数値計算における誤差の要因の 1 つです。

では実際に計算してみます:

```
irb> sincos 1.04719755112, 5                  ← π/3 (60度)
=> [0.866025445061482, 0.50000043349925]      ← 確かに cos が 0.5
irb> sincos 3.14159265359, 5                  ← π
=> [0.00692527070730282, -0.976022212623592] ←いまいち…
irb> sincos 3.14159265359, 10                 ← n を増やすと
=> [-5.2912555176166e-10, -1.00000000352908] ←まあ OK
irb> sincos 314.159265369, 10                 ← 10 π
=> [-2.2869271590877e+30, -1.38360262196954e+29] ←デタラメ…
```

何が問題なのでしょう? それは、x が大きくなるほどテイラー級数の収束が遅くなるためです。これに対処するため、sin とか cos が周期関数であることを利用し、この方法で計算するのは絶対値

の小さい $0 \leq x \leq \frac{\pi}{4}$ の範囲だけにすべきでしょう (この範囲の sin と cos があれば残りの範囲は全部これらをもとに計算できますから)。

そして、x の範囲をこのように限定するなら、テイラー級数の項の数は 8 つくらいあれば十分と分かります (その先の項は分子の絶対値が 1 より小さく、分母は 10^{10} 以上になるので、そこで打ち切っても精度は十分です)。その場合、加えていく順序をテイラー級数の後ろの項から順にしたほうがよいのです。と言うのは、後ろのほうほど絶対値が小さくなるので、前から順に足すと情報落ちしやすくなります。項の数を決めておけば、後ろから足すように書くのも簡単です。

3 制御構造の組み合わせ (ふたたび)

簡単なプログラムでは制御構造として「枝分かれ」「繰り返し」のどちらかを 1 つだけを使えば済みますが、もう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、「0〜99 の数を順に打ち出すが、ただし 3 の倍数の時だけは fizz と打ち出す」という例を考えてみます: [4]

- fizz1: 3 の倍数の時だけ fizz
- 　変数 i を 0 から 100 の手前まで変えながら繰り返し、
- 　　もし i が 3 の倍数ならば、
- 　　「fizz」と出力。
- 　　そうでなければ、
- 　　i を出力。
- 　　枝分かれ終わり。
- 　以上を繰り返し。

これを Ruby に直したものは次のようになります (「3 の倍数かどうか」は「3 で割った余りが 0 かどうか」を調べれば分かりますね):

```
def fizz1
  100.times do |i|
    if i % 3 == 0
      puts('fizz')
    else
      puts(i)
    end
  end
end
```

動かしてみましょう:

```
irb> fizz1
fizz
1
2
fizz
(途中略)
97
```

[4] 海外で古くからある言葉遊びに **fizzbuzz** というのがあります。これは輪になって「$1, 2, \ldots$」と順に数を唱えますが、ただし数が 3 の倍数なら「fizz」、5 の倍数なら「buzz」、3 と 5 の公倍数なら「fizzbuzz」と (数の代わりに) 言わなければならず、間違えたりつっかえたりしたら負けで輪から抜ける、というものです。日本で有名なのは世界のナベアツの「3 の倍数と 3 がつく数字の時だけアホになります」というネタですが、ナベアツも fizzbuzz をヒントにこのネタを考案したという説があります。

```
98
fizz
=> 100
irb>
```

このように、基本的な制御構造を組み合わせていけば、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) 作れるのと同じだと考えてください。

演習 1 上の fizz プログラムを打ち込んでそのまま動かせ。動いたら、繰り返しと枝分かれを組み合わせて次の動作をする Ruby プログラムを作成せよ。

a. 0 から 99 までの数のうち、2 の倍数でも 3 の倍数でもないものだけを順に打ち出す。
b. 0 から 99 までの数を順に打ち出すが、ただし 3 の倍数の時は fizz、5 の倍数の時は buzz、3 の倍数かつ 5 の倍数の時は fizzbuzz と (いずれも数値の代わりに) 打ち出す (fizzbuzz 問題)。[5]
c. 0 から 99 までの数を順に打ち出すが、ただし 3 の倍数と 3 がつく数字の時は数値の代わりに aho と打ち出す。

以下の 3 問は前回の演習 5〜7 と (ほぼ) 同じなので、やってしまった人はご勘弁ください。というか、今回解説する時間がないので次回解説するため、ここに再録しています。

演習 2 2 数 a、b の**最大公約数** (greatest common divisor) を求めるアルゴリズムを次に示す:

- gcd1: 整数 x、y の最大公約数を返す
- 　x ≠ y である間繰り返し、
- 　　x ＞ y なら、
- 　　　x ← x - y。
- 　　そうでなければ、
- 　　　y ← y - x。
- 　　枝分かれ終わり。
- 　繰り返し終わり。
- 　x を返す。

これを Ruby プログラムにして動かせ。これで最大公約数が求まる理由も併せて説明すること。(ヒント: $x > y$ ならば $gcd(x, y) = gcd(x - y,\ y)$ 等のこと (つまり x と y の大きいほうから小さいほうを引いても 2 数の最大公約数は変化しないこと) を示せばよいわけですね。)

演習 3 「正の整数 N を受け取り、N が素数か否かを (`true`/`false` で) 返す Ruby プログラム」を書け。まず擬似コードを書き、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割り切れない、つまり剰余が 0 でないということ。剰余は演算子`%`で計算できるのでしたね。)

演習 4 「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。N が大きくなるように工夫してくれるとなおよい。(ヒント: 処理を速くするためには、(1) 割ってみる数をできるだけ少なくとどめる、(2) 素数の候補とする数をできるだけ少なくとどめる、という 2 点を工夫するとよいでしょう。たとえば 2 は別扱いして奇数だけ扱うなど。)

[5] fizzbuzz 問題については、「(米国で) プログラマを募集して応募者にこの問題のプログラムを書かせてみたら書けない奴が多い。だから応募者のふるい分けに使っている」という話があります。本当だとしたら、これを書けた人はプロ級かも (そんなわけはない)。

4　$f(x) = 0$ の求解

4.1　数え上げによる求解

こんどは関数 $f(x)$ について、$f(x) = 0$ を満たす x を求めるという問題、つまり **1 変数方程式の求解** (solving single variable equation) を取り上げましょう。これも解析的に解けなくても、次のような条件があればプログラムで解を求めることができます:

> ある区間 $[a, b]$ において、$f(x)$ が 単調増大、連続、かつ $f(a) < 0$、$f(b) > 0$ となるような a、b が分かっている

a でマイナス、そこからなめらかに増えていって、b でプラスになっているのなら、その間のどこかに解があるわけですから、それを求めればよいわけです。

「$N(> 1)$ の平方根を求める」ことを考えましょう。$f(x) = x^2 - N$ とすれば、$f(0) < 0$、$f(N) > 0$ なのでここで説明する方法で解を求められます (そしてそれが N の平方根なわけです)。では擬似コードを見てみましょう:

- solve1: n の平方根を求める
- 　d ← n / 1000000。
- 　i を 0 から 1000000 の手前まで変えながら繰り返し、
- 　　r ← i × d。
- 　　もし $r^2 - n \geq 0$ なら、繰り返しを抜け出す。
- 　繰り返し終わり。
- 　r を出力する。

つまり、$f(0) < 0$ なのですから、十分小さい d を用意し、$d, 2d, 3d, 4d, \ldots$ について順に $f(x)$ を計算し、最初に 0 以上になったところでやめれば精度 d で解が求まるわけです。これを**数え上げ** (enumeration) による方法と呼びます。Ruby のコードは次のとおり:

```
def solve1(n)
  r = 0
  d = n / 1000000.0
  1000000.times do |i|
    r = i * d
    if r**2 - n >= 0 then break end
  end
  return r
end
```

ループから抜け出すのに **break 文** (break statement) を使っています。break 文を実行すると、現在実行中の一番内側のループを終わって、ループの次の文の実行に進みます。

演習 5　このプログラムを打ち込んでそのまま動かせ。また、精度を上げた時に何桁くらいなら実用になるか試せ。(終わらなくて止めたい時は、多くの OS では「Control+C」で止めることができます。)

演習 6　もっとましな方法を実現してみる (説明は下記。どちらも、分割数 n は不要で、代わりに許容誤差 e を指定する。e は 0.000001 とか固定してもよいし、入力させてもよい)。

a. 区間 2 分法の考え方によって平方根を求めるようにしてみよ。

b. ニュートン法の考え方によって平方根を求めるようにしてみよ。

4.2 区間 2 分法

先の前提では、区間 $[a,b]$ において、$f(a)<0$、$f(b)>0$ であり、この区間中に根があるという前提でした。ところで、$c=\frac{a+b}{2}$ を求め、$f(c)$ を計算してみたらどうでしょうか。もしも $f(c)<0$ であれば、解がある範囲は区間 $(c,b]$ に狭められます。そうでなければ、解がある範囲は区間 $[a,c]$ に狭められます。つまり a か b のどちらかを c で置き換えられるわけです。その後また同様に繰り返すことで、区間の幅を毎回半分ずつにしていけます。この方法を**区間 2 分法** (binary search method) と呼びます。

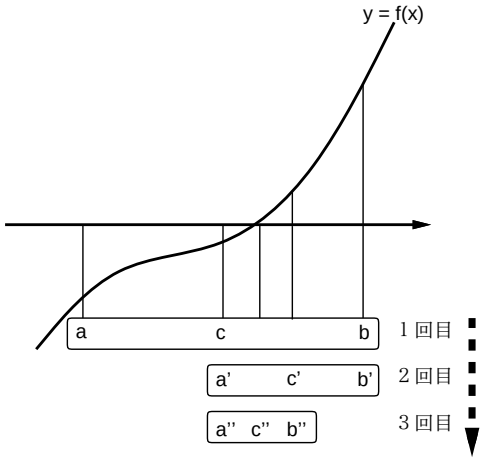

図 3: 区間 2 分法による求根

$2^{10}=1024$ ですから、区間を半分にすることを 10 回繰り返すと区間の幅はおよそ 1000 分の 1、40 回繰り返すとおよそ 1 兆分の 1 になります。言い換えれば、その精度で解が求まるわけです。擬似コードを示します: [6]

- sqrt2bun : n の平方根を誤差 e で求める
- 　a ← 0。b ← n。
- 　$|a-b|>$ e である間繰り返し、
- 　　c ← $\frac{a+b}{2}$。
- 　　もし $c^2>n$ なら、
- 　　　b ← c。
- 　　そうでなければ、
- 　　　a ← c。
- 　　枝分かれ終わり。
- 　繰り返し終わり。
- 　a を返す。

4.3 ニュートン法

ニュートン法 (Newton's method) は、かの万有引力の発見者ニュートンに由来する方法で、[7]適当な近似値 r から始めて、その近似値を改良していくことで解に到達します。具体的には、$f(x)$ の $x=r$ における接線を求め、接線と X 軸が交わる点の X 座標を新たな r とし、これを反復していきます。その i 回目の値を r_i と書き、各回の計算内容を**漸化式** (recurrence formula) として表現しましょう。

[6] x の**絶対値**を求めるのには、前にやったようにいちいち if 文を書かなくても、x`.abs` で求めることができます。
[7] 彼は微積分学の発明者の一人でもあります

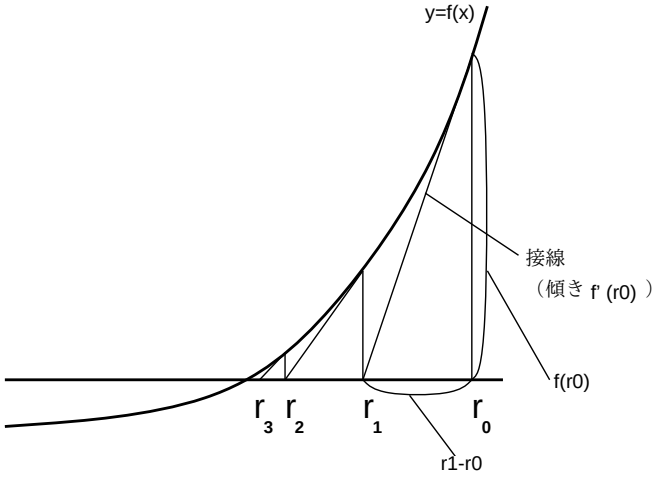

図 4: ニュートン法による求解

具体的にやってみましょう。$x = r_i$ の時の接点の座標は $(r_i, f(r_i))$、そこでの接線の傾きは $f'(r_i)$ (もちろん関数は微分可能でないといけません)。

$$\frac{f(r_i)}{r_i - r_{i+1}} = f'(r_i)$$

より、

$$r_{i+1} = r_i - \frac{f(r_i)}{f'(r_i)}$$

となります。$f(x) = x^2 - n$ の場合、$f'(x) = 2x$ より、

$$r_{i+1} = r_i - \frac{{r_i}^2 - n}{2r_i} = \frac{r_i}{2} + \frac{n}{2r_i}$$

となります。そこで $r_0 = N$ とおき、$r_1, r_2, \cdots$ を計算していくと、その値は $\sqrt{N}$ に**収束** (converge) していくわけです。一般にこのような、近似値を反復によって改良していく方法を**反復解法** (iterative method) と呼びます。反復の結果、値がほとんど変化しなくなったら、つまり $|r_{i+1} - r_i| < \epsilon$ となったら収束したこととし、そこでの近似値を解とするわけです。

なお、収束する値が大きい値であるような計算をする場合は、**絶対誤差** (absolute error) ではなく**相対誤差** (relative error) に基づいて収束を判定するのがよいかもしれません。その場合は反復をやめる条件は次のようになります:

$$\left|\frac{r_{i+1} - r_i}{r_i}\right| < \epsilon$$

ニュートン法は収束すれば高速なことで知られていますが、収束しない場合もあります。平方根の計算の場合は、最初の近似値として N から始めれば問題ありません。こちらも擬似コードを示します:

- sqrt2newton：n の平方根を誤差 e で求める
- 　r ← 0。r1 ← n。
- 　$|r1 - r| > e$ である間繰り返し、
- 　　r ← r1。
- 　　r1 ← $\frac{r}{2} + \frac{n}{2r}$。
- 　繰り返し終わり。
- 　r を返す。

5　さまざまなデータ型

5.1　基本データ型

コンピュータではさまざまなデータを 0/1 の列として表現して扱っています。もとのデータが何であるかによって、どのような表現を使うかを適宜選択する必要があります。実際には、プログラムを書く時はプログラミング言語で記述するので、プログラミング言語が提供している表現方法をそのまま利用したり、組み合わせて利用したりしてデータを表現します。この、「表現の種類」ないし「データの種類」のことを**データ型** (data types) と呼びます。

多くの言語では、内部に構造を持たないデータ型である**基本データ型** (primitive data types) を別扱いしています。Ruby はそのような別扱いをしない言語ですが、他の言語で言う基本データ型に相当するよう種類のデータ型はあるので、関連する型も含めてここで見ておくことにします:

- **整数**型 — 2 進表現で整数を表す。「`123`」など。既に学んだように、Ruby では小さい値には固定ビット数の表現が使われ、それで表現できなくなると複数語を使う表現に切り換えられる。後者は内部構造を持つ型なので基本データ型ではない。
- **実数**型 — 2 進浮動小数点表現。「`3.14`」など。Ruby では 64 ビット IEEE754 形式浮動小数点が使われる。
- **文字列**型。文字の並び。「`'abcd'`」など。文字列も中に文字が複数入るという構造を持つので、基本データ型ではない。
- **記号** (symbol) 型。Ruby 以外では Lisp や Smalltalk など一部の言語にだけ見られる型。Ruby では「`:abc`」のように「`:`」の後に名前を書いたものがその名前に対応する記号**リテラル** (literal) となる。[8]記号型は「複数の値について、互いに同じか違うか区別できることがだけが必要」な場合に使われる。たとえば、猫と犬と馬は別のものなので、変数に`:cat`、`:dog`、`:horse` のどれかを入れておいて区別を覚えておく、などの使い方が典型的。[9]
- **true**/**false** — **真偽値** (truth value — はい/いいえの値)。これらは多くの言語では**論理型** (Boolean) と呼ばれる型を構成するが、Ruby ではそれぞれ `TrueClass`/`FalseClass` というクラス (後述) に属する値である。
- **nil** — 「値がない」ことを表すのに使う目印の値。

5.2　構造を持ったデータ型

構造を持ったデータ型 (structured data type) ないし**複合データ型** (compound data type) とは、その中に基本データ型を複数個含みえるような種類のデータ型を言います (図 5)。以下ではまずプログラム言語としての一般論を説明して、それから Ruby の場合を説明します。

- **配列**型 — (多くの言語では) 同種類の値が並んだもの。[10]
- **レコード** (record) 型 — 複数の型の値を組にしたもの。それぞれの (中に含まれている) 値を**フィールド** (field) と呼び、名前で参照できる。フィールド参照の前に「`.`」を置く言語が多い。[11]
- **オブジェクト** (object) 型 — 内部的に (レコード型のように) データを保持しているが、外部から内部のデータに直接アクセスするのではなく、**操作** (operation) ないし**メソッド**を呼び

[8]リテラルとは「そのまま」という意味で、プログラミング言語では「値を直接書いたもの」を意味します。たとえば `3.14` や`'abc'` はそれぞれ数値リテラル、文字列リテラルです。

[9]文字列 s に対して `s.intern` というメソッドを呼び出すことで s を名前として持つ記号が得られ、この方法なら空白などを含む記号も作れます。逆に y が記号のとき、メソッド `y.to_s` で y の名前の文字列が得られます。

[10]数学の x_i(添字つき変数) みたいなものと考えてください。詳しくは後述します。

[11]たとえば `h` という変数が人のデータを表すレコード型なら、`h.name` には名前 (文字列)、`h.age` には年齢 (整数) が入っている、というふうに使います。

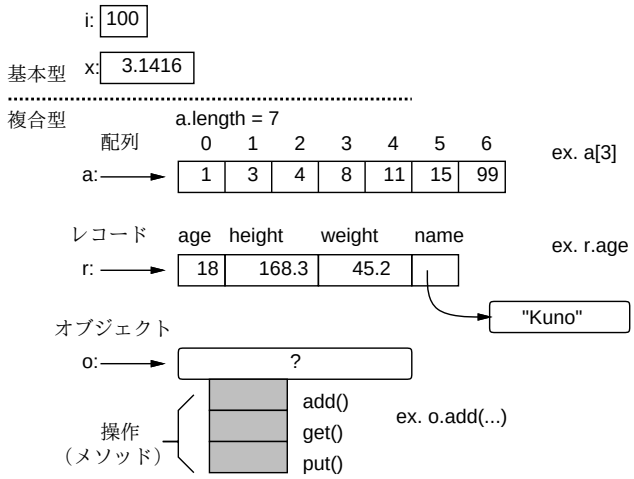

図 5: さまざまなデータ型

出してその機能を利用するようなもの。オブジェクトの機能をサポートするプログラミング言語を**オブジェクト指向言語** (object-oriented language) と呼ぶ。(多くのオブジェクト指向言語では、オブジェクト o に対して $o.method(\cdots)$ でメソッド呼び出しを指定します。Ruby のようにパラメタがない場合は丸かっこも省略できる言語も複数あります。オブジェクト型とレコード型は「内部に値を保持する」という点で似ているので、多くのオブジェクト指向言語ではオブジェクト型とレコード型を統合しています。Ruby もそのような言語の 1 つです。具体例については次の回に扱います。)

図 5 を見て不思議に思ったことはないでしょうか。具体的には、基本型 (整数等) では変数の位置に「箱」が書かれていてそこに値が入っていますが、オブジェクト型 (配列等) では少し離れたところにデータを入れる場所があって、変数からはそこに矢印が出ていますね。実はこの矢印はデータのありかを示す**参照** (reference — ありかを指す値で、実体はメモリ上の番地だと思ってよい) です。そして、レコードのフィールド `r.name` に文字列を入れるとすると、実際には文字列は別の場所に入っていて、フィールドにはその場所への参照が入っているわけです。

この「値と参照の区分」はまた後でも出てきますが、とりあえず「`a = b`」のように変数間で代入をした時、基本型では値 (箱の中身) がコピーされますが、オブジェクト型では参照 (矢印) がコピーされるだけで、本体は 1 つのまま (単に 2 つの変数が同じ場所を指すだけ)、と考えてください。

または、全部が矢印であると考えておいても構いません。実は、整数や実数の「中身を変更する」方法はないので、どちらで考えても同じことなのです。しかし、単純な値は箱の中に書いた方が判りやすいので、ここではその方針で説明しています。

さらに、「2 つの変数が同じ場所を指している」状態でそのオブジェクトの中身を書き換えると、もちろんオブジェクトは 1 つだけなので、どちらの変数から見たオブジェクトも同じように変化していることになります。このあたりの挙動は勘違いしやすいので注意が必要です。

5.3 配列

上述のように「配列」とは、「同種のデータを沢山ならべたもの」という意味です。Ruby では値の種別を制約しないので、同種でなくてもよいのですが、先に書いたように配列は x_i のような添字つき変数として使うので、添字によって値がまったく別種のもの、というのは扱いづらく、結局同種のものを入れるのが普通です。

配列を使うには、まず配列を作り出す必要があります。その方法としては色々ありますが、ここでは代表的なもの 2 つを説明しておきます:

```
a = [1, 2, 3]
  # 3 要素の配列を作り値 1〜3 を入れる
a = Array.new(100, 0)
  # 100 要素の配列を作り全要素に初期値 0 を入れる
```

要素数を指定する方法では、初期値を指定しないと各要素には nil が入ります。全体的に、値を並べて指定する方法は少数の値を用意する場合に使い、[12]個数を指定する方法は大きな配列で使います。配列は後からメソッド push で要素を追加できます。たとえば上の例の 2 番目と次は同じ結果になります:

```
a = []                          # 0 要素の配列を作り
100.times do a.push(0) end  # 100 回「0」を追加
```

なお、現在の**配列の長さ** (array length) ないし要素数は、メソッド length で取得できます。上の例では a.length は 100 です。

いちど用意してしまえば、配列の個々の要素は 1 つの変数と同様に扱えます。ここで「どの要素か」を指定するのに [...] の中に式を書いて指定します。これを**添字** (index) と呼びます。たとえば上の例だと a[0]〜a[99] という要素があることになります (0 番目から数えることは慣れないと忘れやすいので注意してください)。

また、Ruby ではまだ用意していない添字番号 (たとえば上で「100 番」とか) の要素を参照すると nil が返ります。飛び離れた添字番号 (たとえば上で「200 番」とか) に値を格納すると、そこまでの途中の要素は全部 nil で埋められます。

では、配列を与えてその合計を求めるというのをやってみましょう (合計は積分とかで散々やったので簡単ですね):

- arraysum：配列 a の数値の合計を求める
- 　sum ← 0。
- 　i を 0 から配列要素数の手前まで変えながら繰り返し、
- 　　sum ← sum + a[i]。
- 　繰り返し終わり。
- 　sum を返す。

Ruby コードは次のとおり:

```
def arraysum(a)
  sum = 0
  a.length.times do |i|
    sum = sum + a[i]
  end
  return sum
end
```

一応、動かすところの様子を示します:

```
irb> arraysum([1,2,3,4,5])
=> 15
```

実は Ruby では「配列の各要素を取りながら周回するループ」というのもあって、そのほうが少し簡単になります。コードだけ示しておきます:

[12]値を並べて書く方法は「そのまま値を書く」ことから「配列リテラル」と呼ぶこともあります。しかし、配列では初期値を指定するのに変数や任意の式を指定できるので、厳密に言えば「そのまま」ではありません。Ruby の用語でもこの書き方は**配列式** (array expression) というのが正式な呼び方です。

```
def arraysum1(a)
  sum = 0
  a.each do |x|      # x に配列の各要素が順次入る
    sum = sum + x
  end
  return sum
end
```

合計ならこのほうが少し簡単ですが、「何番目」を必要とする場合もあるので、その場合には計数ループを使うことになるでしょう。[13]

演習 7 上記の配列合計プログラムの好きな方をそのまま打ち込んで動かせ。動いたらこれを参考に下記のような Ruby プログラムを作れ。[14]

a. 数の配列を受け取り、その最大値を返す。
b. 数の配列を受け取り、最大値が何番目かを返す。なお先頭を 0 番目とし、最大値が複数あればその最初の番号が答えであるとする。
c. 数の配列を受け取り、最大値が何番目かを出力する。なお先頭を 0 番目とし、最大値が複数あればそれらをすべて出力する。
d. 数の配列を受け取り、その平均より小さい要素を出力する (例: 1、4、5、11 → 1、4、5)。

演習 8 「素数列挙」の問題は、配列を使うとより高速にできる可能性がある。次の 2 つの方針を用いたプログラムを作成し、これまでに作ったものと速度を比較せよ。

a. 素数は値の大きいところではまばらにしかないので、これまでに見つかった素数を配列に覚えておき、新たな素数の候補をチェックする時に「これまで見つかった素数で割ってみて割り切れなければ素数」という方針にすれば、チェックする回数がかなり少なくできる。
b. まったく別の考え方として、N 未満の素数を打ち出すのに次の方針を用いるのはどうだろう: [15][16]

- 論理値が並んだ要素数 N の配列を作り、全部「真」に初期化。
- 2、4、6、…と、2 の倍数番目の部分を「偽」に変更。
- 3、6、9、…と、3 の倍数番目の部分を「偽」に変更。
- 同様に、素数の倍数番目を「偽」に変更していく。
- 最後に、「真」で残っているところを順に調べ何番目かを出力。

A　本日の課題 3A

「演習 1」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 3A」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 1」で動かしたプログラムどれか 1 つのソース。

[13] メソッド `each_index` で配列の添字を順次取り出してループすることもできます。
[14] 「返す」の場合は上の例と同様に `return` を使い、「出力する」の場合は `puts` を使って画面に直接 (その場で) 出力させてください。`return` は使った瞬間にそのメソッド呼び出しは終ってしまうので、複数回 `return` を使うことはできません。
[15] これは「方針」であって、まだ擬似コードでもないことに注意してください。
[16] この方法を考案したのはギリシャの哲学者**エラトステネス** (Eratosthenes) であり、この方法を彼の名前を冠して**エラトステネスのふるい** (sieve of Eratosthenes) と呼びます。なぜ「ふるい」かというと、素数でないもの (各数の倍数) をふるい落としてしまうと、残ったものは素数だ、という方針でできているからです。

4. 以下のアンケートの回答。

   Q1. 繰り返しを使ったプログラムに慣れましたか。
   Q2. 配列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 3B

次回までの課題は「演習 1」〜「演習 8」の (小) 課題からプログラムを 2 つ以上作り、報告すること。ただし、配列を使うものが最低 1 つは含まれるように選ぶこと。レポートは授業開始時刻の **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 3B」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. その簡単な説明。
5. もう 1 つのプログラムのソース。
6. その簡単な説明。あれば考察等も。
7. 下記のアンケートの回答。

   Q1. 配列が使いこなせるようになりましたか。
   Q2. 今回もまた、疑似コードを書くのと、Ruby に直すのと、打ち込んで動かすのとで掛かった手間の比率を教えてください。
   Q3. 課題に対する感想と今後の要望をお書きください。