情報科学 2007　久野クラス　＃3

久野　靖*

2007.10.26

はじめに

今回はまず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。また、本日の新しい内容として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　演習問題解説 — 枝分かれ

1.1　演習 2a

演習 2a は 2 つの枝分かれですから例題とほとんど同じ。まず疑似コードを見よう。

- max2: 数 a、b の大きい方を返す
- 　もし a > b であれば、
- 　　$result \leftarrow a$。
- 　そうでなければ、
- 　　$result \leftarrow b$。
- 　枝分かれおわり。
- 　$result$ を返す。

Ruby では次の通り。

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

しかし、次のような「別解」はどうだろう。

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

*筑波大学大学院経営システム科学専攻

- 　*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
```

どっちが好みですか? これもどちらが正解ということではない。

1.2　演習 2b — 枝分かれの入れ子

しかし演習 2b はもうちょっとややこしい。まず考えるのは、a と b の大きい方はどっちか決めて、それぞれの場合についてそれを c と比べるというもの。

- 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、
- 　　$result \leftarrow c$。
- 　　枝分かれおわり。
- 　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、
- 　　$result \leftarrow c$。
- 　　枝分かれおわり。
- 　枝分かれおわり。
- 　*result* を出力する。

うーむ大変だ。これを Ruby にしておく。

```
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。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、できるだけ避けた方がよい。ときに、先の別解から発展させるとどうだろう?

- 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。

この方がすっきりしているでしょう? Ruby でも次のとおり。

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

今度はどちらが好みですか? 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方が分かりやすいといえる。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ね。なお、「もし」の疑似コードが 1 行に書かれているが、この場合はこちらのの方が見やすいと思ったので。

1.3　多方向の枝分かれ

演習 2c は 3 通りに分かれるのだから、if の中にまた if が入るのはやむを得ない。Ruby コードを見てみよう。

```
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る。具体的には、`if` 文には「`elsif` 条件 `then` 動作」というのをいくつでも途中に入れることができる。具体的には次の通り:

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

順序が前後したが、疑似コードだと次のようになる。

- 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。
- 枝分かれおわり。

「そうでなくて〜ならば、」は何回現われてもよい。またそのどれもが成り立たない場合は「そうでなければ」に来るが、この部分は不要ならなくてもよい。

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 が最大だと分かる。これを利用した 3 方向枝分かれで書くこともできる (変数を使わず値を返すスタイルにしてみた):

```
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 に比例してしまう。もっともだからいけないというわけではなく、N の個数がいくつと決まっていればこのやり方を使ってもいいかも知れない。

2 演習問題解説 — 繰り返し

2.1 演習 4a〜4c

このあたりは簡単なのでさっさと Ruby プログラムを示そう。2 の N 乗。

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

計算だけなら `2**n` でよいのだが、繰り返しを使うという指示があったので。つぎは N の階乗。

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

組み合わせ。

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

なお、組み合わせは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行かない。$\frac{4\times5\times6\times7}{1\times2\times3\times4}$ みたいに。この順序でやれば常に割算が割り切れるので誤差なしで計算できる。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。

2.2 演習 4d

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

```
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、i 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く。一応、計算式を再録しておく。

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

$$cosx = \frac{1}{0!}x^0 - \frac{1}{2!}x^2 + \frac{1}{4!}x^4 - \frac{1}{6!}x^6 + \cdots$$

ところで、「何項まで計算するか」によって精度が代わって来るが、言い替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る。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る。これを**打ち切り誤差**といい、既に学んだ**丸め誤差**、**情報落ち**、**桁落ち**とならんで数値計算における誤差の要因の 1 つとなる。では実際に計算してみよう。

```
irb(main):040:0> sincos(1.04719755112, 5)       ← π/3 (60 度)
=> [0.866025445061482, 0.50000043349925]        ← 確かに cos が 0.5
irb(main):041:0> sincos(3.14159265359, 5)       ← π
=> [0.00692527070730282, -0.976022212623592]    ←いまいち…
irb(main):042:0> sincos(3.14159265359, 10)      ← n を増やすと
=> [-5.2912555176166e-10, -1.00000000352908]    ←まあ OK
irb(main):043:0> sincos(314.159265369, 10)          ← 10 π
=> [-2.28691271590877e+30, -1.38360262196954e+29] ←ぐちゃぐちゃ…
```

何がいけないのだろう? 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。ではどうすれば? sin とか cos は周期関数なので、上の方法で計算するのは絶対値の小さいところ、つまり $0 \leq x \leq \frac{\pi}{4}$ だけにしておくのがよい (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るから)。あとついでに、上のプログラムではやっていなかったが、加えて行く順序をテイラー級数の後ろの項から順にした方がよい。というのは、後ろの方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するため。それは 5 項とか決めておけば、その順で計算式を書けばよいだけのこと。

2.3 演習 5

課題には「考え方」だけ書かれていたが、疑似コードを示す。

- 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。

Ruby のコードは次の通り。

```
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だろう? 次のように考えてみるとよい。なお、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。[1]
- したがって、$x - y$ を改めて x とおいて、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。

どうですか、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、という意味が分かります?

3 数値積分 (つづき)

演習解説の途中だけれど、ここからが数値積分の話の本番なのであらためて。前回、数値積分の考え方を説明して、しかし区間の左端や右端の $f(x)$ の値では誤差があることまで述べた。

で、演習問題として「左端と右端の平均を取ったら」というのが演習 3a だった。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に等しい。このため、これを数値積分の「台形公式」と呼ぶ (区間の幅を d で表す)。

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

[1]証明: 最大公約数を G とおくと、x も y も G の整数倍なのだから、$x - y$ もまた G の整数倍である。ということは、G は $x - y$ と y の公約数である (最大かどうかはまだ分からない)。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。ということは、H は $x - y + y = x$ の約数でもある。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する。従って G は $x - y$ と y の最大公約数でもある。

Ruby プログラムを示しておく。

```
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、関数の 2 階微分が 0 でない場合、つまり上や下に凸な場合は誤差が出る。具体的には、上に凸だと少なく、下に凸だと多くなる。ところで演習の問題文にも書いたように、左端/右端の代わりに区間の中央の x を使って長方形で計算すると (これを**中点公式**という)、逆に上に凸だと多く、下に凸だと少なくなる (図 1)。だからこれをちょうどよく混ぜたらいいのでは、というのが演習 3c になっていた。

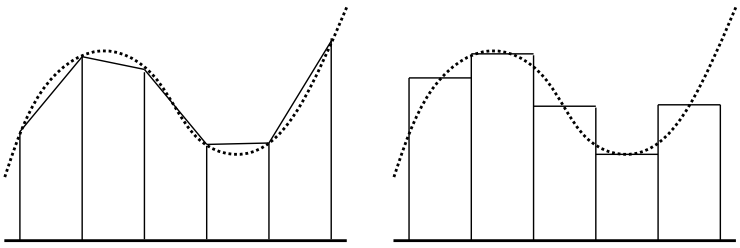

図 1: 台形公式と中点公式

実は、左端:中央:右端を 1:4:1 で混ぜると (つまり台形:中央を 1:2 で混ぜると) よい結果が得られる。そのプログラムも示しておく。

```
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
```

実際にやってみよう:

```
irb(main):006:0> integ4(1.0, 10.0, 100)
=> 333.0                    ←ぴったり? 本当か?
irb(main):007:0> printf "%.20g\n", integ4(1.0,10.0,100)  ← 20 桁表示
332.99999999999994316       ←確かにすごくいい値
=> nil
irb(main):008:0> printf "%.20g\n", integ4(1.0,10.0,10)   ←分割数 10
```

```
  333                              ←分割数減らしたら逆にぴったり?
=> nil
```

実は、この計算式は**シンプソンの公式**と言われ、数値積分では標準的な方法である (下の式では見やすくするため区間の半分を d としている、そのためプログラムの 6 で割る代わりに 3 で割っている):

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

なぜこれがいいのかというと、シンプソンの公式では当該区間を 2 次曲線で補間していることになるから。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) よかったわけである。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…

ではなぜ 2 次式の補間になるのか、その理由を説明する (やりたくないけど)。当該区間の曲線を 2 次式

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

で表せるとする。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 とおく。上の 2 次式の不定積分は $\frac{1}{3}ax^3 + \frac{1}{2}bx^2 + cx$ だから、面積 (定積分) は

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

となる。これを整理すると

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

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 (そんなの見比べて分かるか?!)、

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

になる。というわけで、上の式が出て来るわけである。

さて、ではシンプソンの公式が一番いいのかというと、必ずしもそうとは言えない。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はとっておいて、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よい。

このような計算方法を**漸近的**という。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 つの方法である。

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、つまり求解を取り上げる。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る:

- 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

ひらたく言えば、a ではマイナス、そこからずっと増えていって、b ではプラスになっていて、途中で「飛んでいる」ところがないならその間のどこかに解があるからそれを求める、ということ。

たとえば、「$N(> 1)$ の平方根を求める」としよう。$f(x) = x^2 - N$ とすれば、$f(0) < 0$、$f(N) > 0$ だから上の条件を満たしているので、解を求めることができる (そしてそれが N の平方根なわけだ)。

では次の疑似コードを見てみよう:

- 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、$f(0) < 0$ なのだから、十分小さい d を用意し、$d, 2d, 3d, 4d, \cdots$ 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 で解が求まるわけだ。これを**数え上げ**による方法という。Ruby のコードは次の通り (「ループから抜け出す」のには **break 文**というものを使えばよい)。

```
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
```

演習 1 このプログラムを打ち込んでそのまま動かせ。また、精度を上げたとき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ときは「Control+C」で中止。)

演習 2 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 とか固定してもいいし、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。

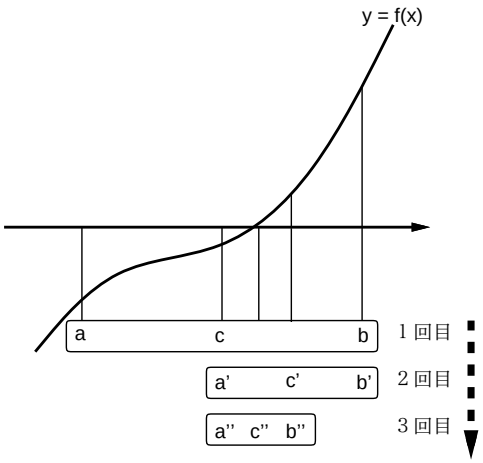

図 2: 区間 2 分法による求根

4.2 区間 2 分法

先の前提では、区間 $[a, b]$ 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だった。ところで、$c = \frac{a+b}{2}$ を求め、$f(c)$ を計算してみたらどうだろうか。もしも $f(c) < 0$ であれば、根がある範囲は区間 $[c, b]$ に狭められる。そうでなければ、根がある範囲は区間 $[a, c]$ に狭められる。つまり a か b のどちらかを c で置き換えられるわけだ。

その後また同様に繰り返すことで、区間の幅を半分ずつにして行ける。2^{10}〜1000 だから、40 回も繰り返せば区間の幅を 10^{12} 分の 1 にできる、言い替えればその精度で解が求まったとも言えるわけだ。擬似コードを掲げておく:

- 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

ニュートン法はかの万有引力の発見者ニュートンに由来する方法で (彼は微積分学の発明者でもある)、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する。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、これを反復していく。これを r_i のような数列と考える。

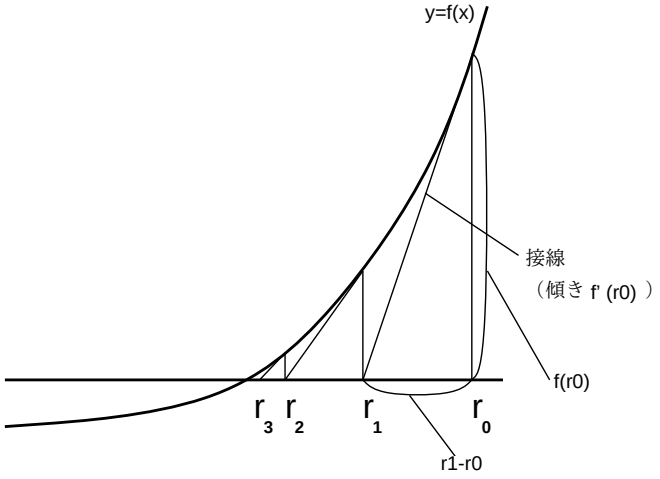

図 3: ニュートン法による求解

具体的に求めてみよう。$x = r_i$ のときの接点の座標は $(r_i, f(r_i))$、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、

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

となる。いちおう親切までに $f(x) = x^2 - n$ の場合、$f'(x) = 2x$ より、

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

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**と呼ぶ。反復の結果、値がほとんど変化しなくなったら**収束**したこととし、そこでの近似値を解とする。つまり収束条件は

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

しかし今回は平方根と分かっているので $|r^2 - n|$ で見てもよい。なお、絶対値は前回の絶対値のプログラムを書いてもいいけど面倒なので、よかったら x`.abs` を使ってください。

ニュートン法は収束すれば高速なことで知られるが、収束しないケースもある (その条件とかは難しいので説明しない、というか久野には説明できません `^_^;`)。ともかく、平方根の場合は近似値として n から始めれば問題ない。こちらも擬似コードを掲げておく:

- 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

＃ 1 でもやったように、コンピュータではさまざまなデータを 2 進表現して扱っている。もとのデータが何であるかによって、どのような 2 進表現を使うかを適宜選択する。実際には、プログラムを書くとき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する。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) と呼ぶ。

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**を別扱いする。Ruby はそのような別扱いをしないが、一応他の言語で基本データ型に相当するような種類のデータというものはあるので、ここで見ておくことにする:

- 整数のデータ型 — 2 進表現
  - `Fixnum` — 整数型。31 ビットの 2 進表現。「`123`」
  - `Bignum` — 多倍長整数型。ある意味「内部構造を持つ」。「`100000000000000000000000`」
- 実数のデータ型 — 2 進浮動小数点表現
  - `Float` — 64 ビット IEEE754 形式浮動小数点。「`3.14`」
- `String` — 文字列型。文字の並び。「`'abcd'`」。なお、文字列は中に文字が複数入っていることが明らかなので、基本データ型とは考えない流儀もある。
- `Symbol` — 記号型。Ruby 以外では Lisp や Smalltalk など一部の言語にだけ見られる方。Ruby では「`:abc`」のように「`:`」の後に名前を書いたもの。または、名前以外の (空白や区切り文字等を含む) 記号は「`'` 文字列`'.intern`」で作り出せる。逆に「記号`.to_s`」で記号に対応する文字列が得られる。では何のために使うかというと、「互いに同じか違うか区別できるもの」のために使う。
- `true`/`false` — 真偽値 (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` というクラスに属する値。
- `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ないし**複合データ型**とは、その中に基本データ型を複数個含み得るような種類のデータ型をいう (図 4)。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 の場合を説明する。

- **配列型** — (多くの言語では) 同種類の値が並んだもの。数学の x_i(添字つき変数) みたいなもの。添字は「`[…]`」の中に書いて表す言語が多い。たとえば `a` が配列なら、`a[0]`、`a[i+1]` などのようにして個々の要素を指定する。さらに詳しくは後述。

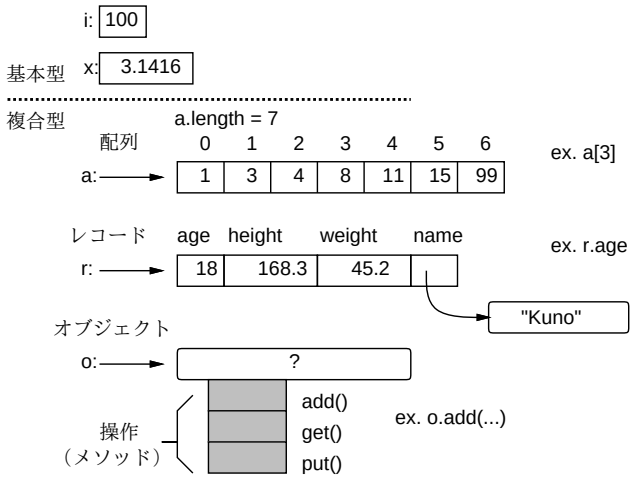

図 4: さまざまなデータ型

- **レコード型** — 複数の型の値を組みにしたもの。それぞれの (中に含まれている) 値を**フィールド**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。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、という風に使う。
- **オブジェクト型** — 内部的に (レコード型のように) データを保持しているが、外部からはさまざまな操作を呼び出して利用するようなもの。オブジェクトをサポートする言語 (オブジェクト指向言語) ではレコードとオブジェクトの機能が融合されていることが多い (Ruby、Java、C++)。これらの言語では操作 (**メソッド**と呼ぶ) を呼び出す場合も「オブジェクト . メソッド名 (パラメタ…)」のように「.」で区切る。さらに Ruby ではパラメタが無い場合は「()」も書かなくていい (レコードのフィールド参照も Ruby では実はメソッドになっている)。

Ruby は「すべてがオブジェクト」という設計の言語だが、とりあえず上の基本データ型に近いものから (おおむね) 順に、必要に応じて取り上げて行く。

図 4 を見て不思議に思ったことはないだろうか。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るが、オブジェクト型 (配列等) では少しはな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る。実はこの矢印はデータのありかを示す**参照** (実体はメモリ上の番地だと思ってよい) である。で、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、フィールドにはそこの参照が入っている。

この「値と参照の区分」はまた後でもやるが、とりあえず「`a = b;`」のようにして変数間で代入をしたとき、基本型では値 (箱の中身) がコピーされるが、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)、と考えるのがよい (全部矢印であると考えておいてもよい。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だが、単純な値は箱の中に書いた方が判りやすいのでそうしてある)。

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る。このあたりの挙動は勘違いしやすいので注意が必要である。

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」という意味である。Ruby では値の種別を制約しないので、同種でなくてもいいが、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いので結局同種のものを入れるのが普通である。

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

```
a = [1, 2, 3]            // 3 要素の配列を作り初期値 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初期値を指定しないと上記の nil が入る。全体的に、初期値を指定する方法は少数の値を用意する場合に使い、個数を指定する方法は大きな配列で使う。配列は後から「.push(値)」で要素を追加できる。たとえば上の例の 2 番目と次は同じ結果になる:

```
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 (要素数) は「.length」で取得できる。上の例では a.length は 100 である。

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る。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する (これを添字と呼ぶ)。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る (例によって 0 番目から数えるのに注意)。

あと、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) を使うと nil が返る。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 で埋められる。

では、配列を与えてその合計を求めるというのをやろう。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 コードは次の通り。

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

一応、動かすところのようす:

```
irb(main):002:0> 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方が少し簡単になる。コードだけ示しておこう。

```
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
```

合計ならこの方が少し簡単だが、「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。

演習 3 上記のプログラム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。

a. 数の配列を受け取り、その最大値を返す。

b. 配列に実数いくつかを読み込み、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のとする。

c. 配列に実数いくつかを読み込み、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. 配列に整数をいくつか読み込み、その平均より小さい要素を出力する (例: 1、4、5、11 → 1、4、5)。

「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` 値」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。

演習 4 数の配列を受け取り、それを「小さい順に並べて」出力する (または配列として返す) プログラムを作ってみよ。受け取った配列の値は書き換えてよいものとする。アルゴリズムの例としては、次のような方法が考えられる。

a. 単純選択法 — たとえば添字番号で言って 0〜9 番があったとする。まず、0〜9 番のうちで最大のものを選び、それを 0 番に入れる (これまで 0 番に入っていたものはその最大のものが入っていた場所に代わりに入れる)。つぎに 1〜9 番で最大のものを選び、それを 1 番に入れる (これまで 1 番に入っていたものはその最大のものが入っていた場所に代わりに入れる)。以下同様にしていく。

b. 比較交換法 — 隣接する 2 つの要素の大小を比べ、「大-小」の順で並んでいたらそれを交換して「小-大」の順にする。これを「それぞれの位置について十分な回数だけ」繰り返す。

これ以外の他の方法を自分で考えてもいいです。

演習 5 演習 4 のような方法で「並べ換え」をテストしていると、データの量が少ないので性能の差が分からない。そこで、入力する代りに「`rand(`N`)`」(区間 $[0, N)$ の一様乱数生成) を使って十分大きな配列 (1 万とか 10 万とか) にデータを入れ、作成した整列アルゴリズムで所要時間を計測してみよ。また、もっと速くする工夫があれば試してみよ。

時間計測の方法については、資料＃ 1 にあるように作業開始時と終了後に「`Process.time.utime`」を参照して時刻を取得し、その差を取ればよいでしょう。

6 アルゴリズムの改良

6.1 演習 6 — 論理型

すごーくお待たせしたが、演習問題解説の続き。素数、やってみましたか? 疑似コードは次の通り。

- isprime: 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」
- 　繰り返しおわり。
- 　sosu を返す。

この sosu は先に説明した真偽値 (真/偽のいずれかだけを表す) なので、「真」は `true`、「偽」は `false` で表す。もっとも、習ってないから整数の 1/0 で表したりした、とかでも結構です。なお、この変数 sosu は最初に「真」を入れておいて、どこかで素数でないと思ったら「偽」にして、最後に結果がどちらか見る。このような使い方の変数のことを「旗」(flag) と呼ぶ。では Ruby コードを見てみよう。

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

前にも説明したが、「%」は剰余演算子 (割った余りを返す) なので「n % i == 0」で「n が i で割り切れる」という意味になる。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ある (増分は省略すると 1)。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よね。

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

でもまあ、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。

6.2 演習 7 — プログラムの改良

さて、演習 7 は演習 6 のメソッドを「下請けとして呼び出す」ようにするとラクに作ることができる。疑似コードは略して Ruby コードを見てみよう。

```
def primes(n)
  2.step(n) do |i|
    if isprime(i) then puts(i) end
  end
end
```

ところで、これでやると私の手元では 6000 までやるのに 10 秒くらい掛かっている。これを工夫して速くするにはどうしたらいいだろう? たとえば次のようなことが考えられる。

- 2 は別に打ち出すことにして、候補として 3 以上の奇数だけ調べる (候補の数が半分になる!)。
- 割ってみる数も 3 以上の奇数だけ試す (割ってみる数も半分に!)。
- $N-1$ まで試さなくても、$\sqrt{N}$ まで試せば十分 ($\sqrt{N}$ より大きい因数があるなら、必ず $\sqrt{N}$ より小さい因数もあるわけだから)。
- 割り切れると分かったところで return を返してしまい、その先は調べないようにする。

これらの改良を施した版を次に示す。

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

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

return は直ちに値を返してメソッドを終了させることに注意。で、これだと、10 秒で 170000 までできた。30 倍弱のスピードアップ! ところで、さらに次のような改良もあり得ますね?

- 3 からの奇数全部で割ってみる代りに、これまでに見つかった素数でだけ割ってみれば十分。

素数は数が大きくなるとかなりまばらにしかないので、これで割ってみる数が減らせる。ただし残念なことに、これまでに学んだやり方では「見つかった素数を覚えておいて利用する」ことができない! 実は、後で学ぶ「配列」を使うとこれができるので、ぜひこの改良にチャレンジしてみて欲しい。

演習 6 配列を使って「N 未満の素数を全部打ち出す」プログラムを「これまでに分かっている素数でだけ割って見る」ように改良し、どれくらい速くなったか調べよ。

演習 7 次のような構想 (これはまだ疑似コードではない!) に従って「N 未満の素数を全部打ち出す」プログラムを作り、速さを評価せよ。

- 論理値 (boolean) 型が並んだ要素数 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 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 3A」とする。
2. 学籍番号、氏名、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。

   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 3」〜「演習 7」の (小) 課題からプログラムを 2 つ以上作り、報告すること。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、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。