基礎プログラミング+演習 ＃3 – 制御構造の組み合わせ+配列

久野 靖 (電気通信大学)

2017.12.11

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。その後の本日の内容としては、次のものを取り上げます。

- 制御構造の組み合わせについて (再)
- データ構造と配列

1 前回演習問題の解説

1.1 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
- 　$result \leftarrow a$。
- 　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。

1.2 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。

1.3 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
```

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。

- sign2: 数 x の正/負/零に応じて positive/negative/zero を返す
- 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。

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。

1.4 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).to_f / n
  s = 0.0
  n.times do |i|
    x = a + i * dx
    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 1)。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).to_f / n
  s = 0.0
  n.times do |i|
    x = a + i * dx
    y0 = x**2
    y1 = (x+0.5*dx)**2
    y2 = (x+dx)**2
```

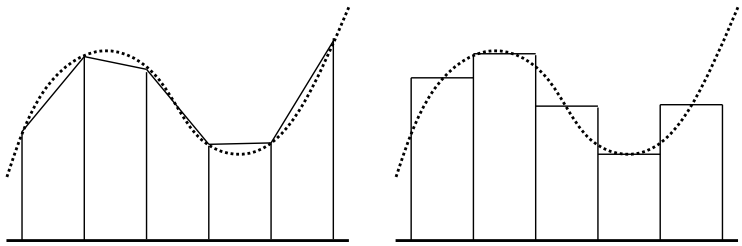

図 1: 台形公式と中点公式

```
    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, 10, 100
=> 333.0                ←ぴったり? 本当?
irb> printf "%.20g\n", integ4(1, 10, 100)
332.99999999999994316  ←確かにすごくよい
=> nil
irb> printf "%.20g\n", integ4(1, 10, 10)
333          ←分割数を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。[1]

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

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

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

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。

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

ところで

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

[1] 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

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

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

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。

1.5 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
```

階乗の方は「1 × 2 ×…× N」を計算したいわけですが、`times` が渡して来るカウント値は「0, 1, …, N-1」なので、全部 1 足してから掛けています。しかしそれはちょっと分かりにくいですね。

実は、N`.times` の代わりに N`.step(`M`,` d`)` という別のメソッドを使うと、初項 N、終項 M、増分 d を指定して計数ループを作ることができます (d は指定しないと「1」が使われます)。これを使えば、階乗は次のようにもっと分かりやすくなります。

```
def factx(n)
  result = 1
  1.step(n) do |i| result = result * i end
  return result
end
```

上の `step` は「`i` を 1 から `n` まで 1 ずつ増やしながら」という擬似コードに対応します。

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
  1.step(r) do |i|
    result = result * (i + (n-r)) / i
  end
  return result
end
```

1.6 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。

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

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

「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> Math::PI / 3
  => 1.0471975511966  ← π/3 (60 度) はこの値)
irb> sincos 1.0471975511966, 5   ←π/3 の sin,cos
  => [0.866025445099782, 0.500000433432913] ←微妙
irb> sincos 1.0471975511966, 10  ←項を増やすと
  => [0.86602540378444, 0.499999999999998] ←まあ OK
irb> sincos 3.141592653589, 10   ←π だと
=> [-5.2812499185062e-10, -1.00000000352908] ←微妙
irb> sincos 31.41592653589, 10   ← 10 π
=> [-167876715320.415, -104528895953.392] ←破綻
```

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。

2 制御構造の組み合わせ (再) exam

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。[2]

- 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。

```
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
```

[2]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, . . .」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。

```
(途中略)
97
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)。[3]
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、GCD) 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。)

[3] 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)。

3 配列とその利用

3.1 データ構造の概念と配列 exam

ここまででではプログラムが扱うデータは個々の「値」であり、1 つの変数に 1 つの値が入っていました。しかしこのやり方では、大量のデータを扱うのが困難なのは明らかです。ではどうするかというと、複数のデータを組にしたり、列として並べるなどの「構造」を持たせて扱う、というのが答えです。この、データに持たせる構造のことを**データ構造** (data structure) と言います。

プログラミング言語の用語では、データの種類のことを**データ型** (data type)、その中で「整数」「実数」など単一の値から成るものを**基本型** (primitive data type) と呼びます。それと対比して、組や列など複数の値が集まったデータのことは**複合型** (compound data type) と呼びます。実は文字列は、中に複数の文字が含まれているので複合型だといえます。

今回は複合型のうちでもよく使われる**配列** (array) を取り上げます。配列は既に「`[1, 2, 3]` のように値を並べたもの」として言及したことがありますが、要するに値が一列に並んだものです。図 2 のように、整数であれば 1 つの変数に 1 つの値しか入れられませんが、配列を使うことで 1 つの変数に一連の値を入れることができます。

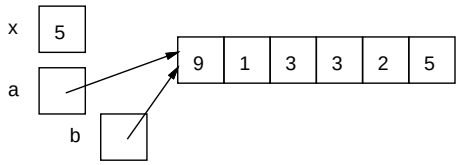

図 2: 配列の概念

図 2 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、複合型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。

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、変数に配列を入れると、配列本体はどこか別の場所に置かれ、変数にはその場所への参照が入ります。そして、「`b = a`」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、複合型では参照 (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) です。[4]

さらに、「2 つの変数が同じ場所を指している」状態でその複合データの中身を書き換えると、複合データは 1 つだけなので、どちらの変数から見た複合データ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間違えやすいので注意が必要です。

3.2 配列の生成 exam

配列を使うには、まず配列を作り出す必要があります。その方法が色々ありますので、ここではそれらについて説明しておきます。

```
a = [1, 2, 3]           # 直接指定
a = Array.new(100, 0)   # 要素数と初期値
a = Array.new(100) do 0 end   # 要素数とブロック
a = Array.new(100) do |i| 2*i end   # 〃
```

1 番目の方法はこれまでにも使ってきた、各要素を直接指定する方法です。[5] この方法は、比較的少数の値を用意する場合に使います。

[4] Ruby の場合。C 言語はまた違います。

[5] 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。

2 番目は、要素数と初期値を指定する方法で、要素数の多い配列を用意するときにはこの方法が一番単純です。[6]

3・4 番目も、要素数と初期値を指定する方法ですが、初期値として値を計算するブロック (do〜end) を指定するところが違います (この場合はブロックの中で式を直接指定します。メソッドではないので return は書けません)。0 などと定数を指定した場合は 2 番目と変わりませんが、ブロックは (times などと同様)「何番目」というパラメタを受け取ることができるので、それを用いて計算により初期値を決めてもよいのです。

配列は後からメソッド `push` で要素を追加できます。上の例の 4 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 |i| a.push(2*i) end  # 0〜198 を追加
```

現在の**配列の長さ** (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。

演習 5 ブロックを指定する形で 10 要素の配列を生成し、初期値を (a)〜(d) のようにしなさい。

(a)

10	9	8	7	6	5	4	3	2	1

(b)

0	1	0	1	0	1	0	1	0	1

(c)

4	3	2	1	0	1	2	3	4	5

(d)

1	1	1	1	1	0	0	0	0	0

なお、初期値を指定するブロックの中でも if を使えます。

```
if 条件 then 式1 else 式2 end
```

この if は「if 式」であり、条件の成否に応じて「式 1」「式 2」のいずれかが全体の値となります。

3.3 配列の利用 exam

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。

[6] 初期値を指定しないと各要素の初期値は `nil` になります。

- 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。

```
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。[7]

演習 6 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。[8]

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)。
e. 数の配列を受け取り、その内容を「小さい順」に並べて出力する (例: 4, 11, 5, 1 → 1, 4, 5, 11)。

演習 7 「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。

[7] 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。
[8] 「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。

b. 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。[9][10]

- 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 から始めて順次、その番号が「真」の値は素数として出力。
- 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、素数を出力するごとにその倍数番目を「偽」に変更。

本日の課題 3A

「演習 1」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. 制御構造の組み合わせができるようになりましたか。

Q2. 配列について学びましたが、使えそうですか。

Q3. 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 3B

「演習 1」〜「演習 7」(ただし演習 5 は除く) 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。配列の内容を含むことを強く勧めます。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. 配列が使いこなせるようになりましたか。

Q2. 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. 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。

[9] 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。

[10] 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。