情報科学 (第 7 週)

関数から計算へ

2007 年 11 月 29 日 増原 英彦

※この資料は暫定版から一部修正されています。

1 結果の提出方法

結果の作成 今回の演習の結果は Ruby のプログラムの形式で提出する。まずエディタを用いて `ex07.rb` というファイルを作成する。(エディタの使い方については HWB15 章を参照せよ。テキストエディット・Emacs のどちらでも構わない。)

次にファイルの先頭に表題、氏名、学生証番号を書く。これらは Ruby のコメントとなるように各行の先頭には `#` を付ける。`#` は半角文字でなければいけないことに注意せよ。

続けて各問題について、次のようにしてメソッドと実行結果を作成してゆく。

1. 「`####` 節番号 – 問題番号」という見出しを書く。
2. メソッド定義を書く。
3. 「ターミナル」で irb を実行し、その中で `load("./ex07.rb")` と入力してメソッド定義を読み込む。
4. 正しく読み込めたら、メソッド定義が正しいことを確認するような式を入力し、実行結果を確認する。
5. 正しい実行結果が得られることを確認したら、4 で入力した式と実行結果を `ex07.rb` にコピーする。さらに各行の先頭に `#` を付けて Ruby のコメントとする。

例えば最初の問題について行った場合は以下のようになる。

```
#情報科学（木曜２限）第７週練習問題
#氏名:  東大 太郎
#学生証番号:  987654

#### 2-1
def power2(n)
  （省略）
end
# irb(main):213:0> power2(10)
# => 1024
```

提出 提出の機会は二度ある

1. 授業時間終了まで — これは演習の進捗状況を把握するためのものなので全員が提出すること。個人差があるので一概には言えないが、平均的には授業時間中に用意した練習問題の 1/3 から半分が解けることを想定している。
2. 12 月 5 日 (水)23:59 まで — 演習時間内に行えなかった問題を可能な限り解いて提出すること。提出は何度でもできるが、複数の提出があった場合は最後に提出されたもののみを見る。ここで提出されたものは成績評価に反映されるので、自力で解くこと。

提出は CFIVE を通して行う。授業 web ページから「練習問題・課題の提出先」というリンクをたどる。「第 7 週練習問題」という課題をクリックし、上で作成した `ex07.rb` というファイルを提出する。提出後、自分が提出したファイルをもう一度ダウンロードして、正しく提出ができたことを確認せよ。

提出期限は機械的に制限されており、ネットワークの不調などがあっても延長することはできない。不測の事態に備え、1 日以上前に提出を完了しておくことを強く勧める。

2 再帰による計算

再帰によってプログラムを作る手順は次のようなものである。例えば、「1 から n までの積 $f(n)$」を求めたい場合には

a) $f(n)$ を f を使って表わしてみる。この場合 $f(n) = 1\times2\times\cdots\times(n-1)\times n$, $f(n-1) = 1\times2\times\cdots\times(n-1)$ であることから $f(n) = f(n-1)\times n$ である。

b) 基底の場合を考える。$f(n)$ が $f(n-1)$ によって定義されているならば $f(n-1)$ は $f(n-2)$ によって定義されている。これを続けてゆくと n はどんどん小さくなる一方である。どこかで $f(n)$ を f を使わずに定義しなければいけない。この場合は $f(1) = 1$ あるいは $f(0) = 1$ とすればよい。より一般的な後者を採用しよう。

c) 関数定義をまとめると次のようになる。

$$f(n) = \begin{cases} n \times f(n-1) & (n > 0) \\ 1 & (n = 0) \end{cases}$$

d) 関数定義をメソッドにする。場合分けを `if..else..end` を使って表わすので、次のようになる。

```
def f(n)
  if n>0
    n*f(n-1)
  else
    1
  end
end
```

次の計算を再帰的な関数として定義してみよ。それに従って、Ruby のメソッドを定義せよ。

1. 2 の `n` 乗 `power2(n)`
2. `x` の `n` 乗 `power(x,n)`
3. 1 から `n` までの和 `sum(n)`
4. 1 から `k` までの整数のうち、`n` の約数である数の和 `divisors_sum(n,k)` (ヒント: `k` が 1 の場合、`n` の約数の場合、そうでない場合の 3 通りに場合分けする)
5. `n` が完全数であるときのみ `true` となる `perfect(n)`。ただし完全数とは `n` の約数のうち `n` 以外のものの和が `n` となるような数で、例えば 496 がそうである。(注: これは `divisors_sum` を使うだけで、再帰的な定義は不要である。)
6. 1 から `k` までの整数のうち、`n` の約数である数の中で最大のもの `greatest_divisor(n,k)` (ヒント: `k` が `n` の約数だったら再帰しない。)

7. n が素数であるときのみ true となる prime(n)。(ヒント: これも greatest_divisor を使うだけで、再帰的な定義は不要である。)

8. n 個から m を選ぶ組み合わせ数 combi(n,m)。(ヒント:combi(n,m) は combi(n-1,m) と combi(n-1,m-1) を使って定義できる。n が 0 のときと m が 0 または n のときはどうなるか?)

9. 1 から n までの整数 k のうち、最大の c(k) を与える longest_c(n)。ただし c(k) は授業で紹介した k のコラッツ数。(ヒント: 補助関数として 2 数のうち大きい方の数を返す max(x,y) を定義しておいて使うと簡単。)

10. 2 から n までの間の素数の個数 prime_count(n)

3 繰り返しによる計算

繰り返しによる計算にはいくつかの典型的な計算方法がある。これらを身につけておくと、色々な応用ができる。

3.1 和や積などを求める

例えば $c(1)+c(2)+\cdots+c(n)$ の値を求める sum_c(n) は次のようになる。

```
def sum_c(n)
  s = 0                  #合計をしまう変数
  for i in 1..n          #i を変化させて
    s = s+c(i)           #合計値を増やしてゆき
  end
  s                      #繰り返し終了後の合計が答
end
```

3.2 条件を満たす値を探す

ある条件を満たす値を求める場合には、繰り返しの形が変わる。例えば $c(k)=x$ となるような最小の k を求める find_c(x) は何回繰り返すのかが分からないので while を使って計算する。

```
def find_c(x)
  k=1                    #k を 1 から変化させて
  while c(k) != x        #c(k)==x となったら終了
    k=k+1                #そうでない間は k を 1 ずつ増やす
  end
  k                      #繰り返し終了後の k は c(k)==k となっている
end
```

3.3 最大値や最小値を求める

最大値の計算は、和や積を求める場合とあまり変わらない。異なるのは、1 回の計算が「より大きな値が見つかったら置き換える」ことになる点である。例えば n 以下の k について $c(k)$ の最大値を求める slongest_c(n) は次のようになる。

```
def slongest_c(n)
  longest=-1              #いままで見つかった最大値を覚える変数。
  for k in 1..n           #1 から n までの k について
    if c(k) > longest     #c(k) がそれまでの最大値より大きい場合は
      longest = c(k)      #最大値を更新する
    end
  end
  longest                 #繰り返し終了時の longest が最大値
end
```

3.4 練習

次の計算を繰り返しとして定義してみよ。それに従って、Ruby のメソッドを定義せよ。

1. 2 の n 乗 spower2(n)
2. x の n 乗 spower(x,n)
3. 1 から n までの和 ssum(n)
4. 階乗を逆順で計算するメソッド sf_down(n)。(ヒント: k を $n, n-1, n-2, \ldots$ と変化させ、g を k 倍してゆく。)
5. 1 から k までの整数のうち、n の約数である数の和 sdivisors_sum(n,k)
6. n が完全数であるときのみ true となる sperfect(n)。ただし完全数とは n の約数のうち n 以外のものの和が n となるような数で、例えば 496 がそうである。(注: これは sdivisors_sum を使うだけ。)
7. 1 から k までの整数のうち、n の約数である数の中で最大のもの sgreatest_divisor(n,k)
8. n が素数であるときのみ true となる sprime(n)。(ヒント: これも greatest_divisor を使うだけ。)
9. 上の 3.3 で示した slongest_c(n) を、2 つの数の大きい方を返す max(x,y) を使って書き直した slongest_c2(n)。
10. 1 から n までの整数 k のうち、最大の c(k) を与える k を返す slongest_ck(n)。(ヒント: longest_c(n) を少し変更するだけ)
11. 2 から n までの間の素数の個数 sprime_count(n)
12. 分母が m 以下の分数で実数 x を最も良く近似する分数 $\frac{n}{d}$ を見つけ、その分母 d を答えるメソッド sapprox_d(x,m)。(ヒント: 分母が d のときの x を良く近似する分数は $\frac{\lfloor xd \rfloor}{d}$ あるいは $\frac{\lfloor xd \rfloor+1}{d}$ のどちらかである。このときの誤差 $|\frac{\lfloor xd \rfloor}{d} - x|, |\frac{\lfloor xd \rfloor+1}{d} - x|$ の小さい方を答えるメソッド error(x,d) をまず定義せよ。また実数 x の整数部分は x.to_i という式で求められる。) また、これを使って円周率 π を最も良く近似する分母 1000 以下の分数を調べてみよ。

4 配列と繰り返し

配列のように大きさが一定のデータ構造に対する計算は、繰り返し構文で記述するのが自然である。
配列を扱う場合には次のような記法をよく使うことになる。

- 配列 a の要素数を調べるには a.size という式を書く。例えば a=[1,5,13] のとき a.size は 3 になる。
- 決められた大きさの配列を作るには Array.new(x) という式を書く。x は配列の大きさである。作られた配列の各要素には nil が入っている。

- x から y までの整数について順に計算を行う場合は次のように書く。

```
for i in x..y
  計算
end
```

これは次のような while を使った構文とほぼ[1]同じである。

```
i=x
while i <= y
  計算
  i=i+1
end
```

次の計算を行うメソッドを定義せよ。

1. 大きさ n で、全ての要素が x であるような配列を作る make_array(n,x)
2. 大きさ n で、k 番目の要素が $c(k+1)$ であるような配列を作る collatz_array(n) ただし $c(k)$ はコラッツ数だとする。0 番目の要素は何でもよい。
3. 大きさ n × m で、全ての要素が x であるような配列を作る make_matrix(n,m,x) 2 次元以上の配列は、配列の配列として表わされる。n × m の配列を作りたかったら、まず大きさ n の配列を作り、次に n[0] から n[m-1] の各要素に、大きさ m の配列を作って代入すればよい。
4. n × n の単位行列を作る identity_matrx(n) (単位行列とは対角要素だけが 1 で、他は 0 であるような正方行列)
5. 行列 a の転置行列を求める transpose(m) (転置行列とは、行列の行と列を交換したような行列)
6. 2 つのベクトル v1 と v2 の内積を計算する dot_product(v1,v2) (ただし v1 と v2 は同じ長さの配列だとする)
7. エラトステネスの篩 (ふるい) とは、素数を高速に求めるアルゴリズムの 1 つである。考え方はとしては、次のようになる。
   - 2 から始まる整数の列を用意する

     $$2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, \ldots$$

   - 列の先頭の数字が消されていなければ、それは素数である。

     $$\boxed{2}, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, \ldots$$

   - 素数を見つけたら、その倍数を列から消す

     $$\boxed{2}, 3, \cancel{4}, 5, 6, 7, 8, 9, \cancel{10}, 11, \cancel{12}, 13, \cancel{14}, 15, \cancel{16}, 17, \cancel{18}, 19, \cancel{20}, 21, \cancel{22}, 23, \cancel{24}, 25, \cancel{26}, 27, \cancel{28}, 29, \cancel{30}, 31, \ldots$$

   - 消されていない数を左から探してゆき、次に見つかったものも素数である。

     $$\boxed{2}, \boxed{3}, \cancel{4}, 5, 6, 7, \cancel{8}, 9, \cancel{10}, 11, \cancel{12}, 13, \cancel{14}, 15, \cancel{16}, 17, \cancel{18}, 19, \cancel{20}, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, \ldots$$

   - その倍数を消す

     $$\boxed{2}, \boxed{3}, \cancel{4}, 5, \cancel{6}, 7, \cancel{8}, \cancel{9}, \cancel{10}, 11, \cancel{12}, 13, \cancel{14}, \cancel{15}, \cancel{16}, 17, \cancel{18}, 19, \cancel{20}, \cancel{21}, \cancel{22}, 23, \cancel{24}, 25, \cancel{26}, \cancel{27}, \cancel{28}, 29, \cancel{30}, 31, \ldots$$

[1] 厳密には同じにならない場合もある。例えば y の値が繰り返しの途中で変化する場合など。

- 繰り返す

  $\boxed{2}, \boxed{3}, \cancel{4}, \boxed{5}, \cancel{6}, \boxed{7}, \cancel{8}, \cancel{9}, \cancel{10}, \boxed{11}, \cancel{12}, \boxed{13}, \cancel{14}, \cancel{15}, \cancel{16}, \boxed{17}, \cancel{18}, \boxed{19}, \cancel{20}, \cancel{21}, \cancel{22}, \boxed{23}, \cancel{24}, \cancel{25}, \cancel{26}, \cancel{27}, \cancel{28}, \boxed{29}, \cancel{30}, \boxed{31}, \ldots$

これを Ruby のメソッドとして定義してみよう。ここで「消されたかどうか」を要素が真偽値 (true か false) であるような配列で表わすことにする。消されていないつまり素数の可能性があるときに true であるとしよう。

(a) 大きさ `n` の全てが true であるような配列を作るメソッド `all_true(n)`

(b) 配列 `p` の $2i, 3i, 4i, \ldots$ 番目を false に変える `delete_multiples(p,i)`
例えば `p=[true,true,true,true,true,true,true]` のとき `delete_multiple(p,2)` を実行すると `p=[true,true,true,true,false,true,false]` となる。(`p[2]` は true のままであることに注意せよ。)

(c) `n` 未満の整数が素数であるかどうかを表わした配列を作る `eratosthenes(n)` ただし 0 番目,1 番目は true でよい。
例えば `eratosthenes(3)` は `[true, true, true, true, false, true]` を返す。

8. 2 から `n` までの素数の個数を返す `aprime_count(n)`

5 計算量の見積り (余力がある人向け)

上の演習では 2 から `n` までの素数の個数を数えるメソッドを 3 種類定義した。実行時間計測プログラムを使ってそれぞれのメソッドの実行時間の `n` に対する変化を調べてみよ。