プログラミング演習I
– 第8回 –
メソッド その 2

1　はじめに

第 8 回の演習の目標は,

- メソッドからメソッドを呼ぶ方法を理解する.
- メソッドの扱いに慣れる.

です.

2　メソッドが別のメソッドを呼ぶ

次のようなプログラムはもう何度か出てきた.

```
------------------------------------------------
def hello
  print("hello\n")
end

hello
------------------------------------------------
```

ここで，hello メソッドの中の print もメソッドであることに注意しよう．つまり，メソッドの中で別のメソッドを呼び出していることになる．メソッドの中でメソッドを呼ぶことは可能である．もちろん自分の定義したメソッドを呼ぶことも問題ない.

演習 8-1
hello メソッドを 3 回呼ぶメソッドを作って試してみよう.

```
------------------------------------------------
def hello
  print("hello\n")
end
```

```
def hello_3
  i = 0
  while i < 3
    hello
    i = i + 1
  end
end

hello_3
-----------------------------------------------
```

演習 8-2

三角柱の体積を求めるメソッドを定義しよう．以下の手順で行う．

1. メソッドを定義せず，三角柱の体積を求めるプログラムを書く．
2. 三角柱の体積を求める部分をメソッドとして定義する．
3. 三角形の面積を求めるメソッドを定義し，三角柱の体積を求めるメソッドから呼び出す．

その 1

```
-----------------------------------------------
base = 3
height1 = 4
height2 = 5
print(base * height1 / 2.0 * height2, "\n")
-----------------------------------------------
```

その 2

```
-----------------------------------------------
# 三角柱の体積を求めるメソッド
#   引数   base:     底辺
#          height1:  底面の三角形の高さ
#          height2:  三角柱の高さ
#   返り値 三角柱の体積
#
def triangular_prism(base, height1, height2)
  return base * height1 / 2.0 * height2
end

base = 3
height1 = 4
height2 = 5
print(triangular_prism(base, height1, height2), "\n")
-----------------------------------------------
```

その 3

```
------------------------------------------------
# 三角形の面積を求めるメソッド
#   引数   base:   底辺
#          height: 高さ
#   返り値 三角形の面積
#
def triangle(base, height)
  return base * height / 2.0
end

# 三角柱の体積を求めるメソッド
#   引数   base:     底辺
#          height1:  底面の三角形の高さ
#          height2:  三角柱の高さ
#   返り値 三角柱の体積
#
def triangular_prism(base, height1, height2)
  return triangle(base, height1) * height2
end

base = 3
height1 = 4
height2 = 5
print(triangular_prism(base, height1, height2), "\n")
------------------------------------------------
```

なお，メソッドに対するコメントの例を示した．適切に書く練習をしよう．

演習 8-3

三角錐の体積を求めるメソッドを定義しよう．三角形の面積を求めるメソッドを呼び出す場合と，三角柱の体積を呼び出す場合を考えてみよう．どちらが便利だろうか？

演習 8-4

三角形の面積を使うような操作を考えてメソッドとして定義し，そのメソッドから triangle を呼び出してみよう．

3　返り値を使う

演習 8-5
以下の成績評価プログラムを試してみよう．合否判定メソッドの返り値によって処理を分岐している例である．

```
-----------------------------------------------
# 合否判定メソッド
#   引数 tensu: 点数
#   返り値 合格か不合格（合格: true, 不合格: false）
#
def goukaku?(tensu)
  if tensu >= 60
    return true
  else
    return false
  end
end

print("あなたの点数を入れてください")
tensu = gets.chomp.to_i

if goukaku?(tensu)
  print("おめでとうございます．合格です．\n")
else
  print("不合格です．追試を受けてください．\n")
end
-----------------------------------------------
```

演習 8-6
合否判定メソッドを改良して，成績判定メソッドを作ってみよう．メソッドは点数を渡すと，成績（ABCD）を返すように作成する．成績ごとにメッセージを考えて表示しよう．

```
4  i = 0
   while i < 10
     練習
   end
```

演習 8-7
さあ，練習練習．この節の見出しが意味する練習回数は何回？(_____) 回

演習 8-8
正の整数 n を引数とし，1 から n までの和を返り値とするメソッドを作ろう．

演習 8-9
配列と文字列を引数とし，配列に文字列が含まれるかどうかを判定するメソッドを作ろう．

演習 8-10
HTML ファイルのヘッダ部分を出力するメソッドを作ってみよう．(教科書 p.40 参照)

演習 8-11
2 つの数値と四則演算子の記号を引数とし，計算結果を返すメソッドを作ってみよう．

演習 8-12
お店をキーとし値段を値とするハッシュを受け取って，値段の平均と最安値のお店と値段を表示するメソッドを作ってみよう．

演習 8-13
姓と名を引数とし，以下のことを実行するメソッドを作ろう．入力はキーボードから行い，姓名はアルファベットとする．

1. 入力形式に関わらず，姓は全て大文字とし，名前は先頭だけ大文字とし，姓と名をこの順で空白 1 文字で区切って表示する．
2. 姓と名それぞれの長さを表示する．
3. 姓と名の長さの和を表示する．

演習 8-14
顔文字を引数とし，別の顔文字に変換して返すメソッドを作成しよう．変換ルールは複数設定すること．例：変換ルール `^ --> @` と `. --> o`

入力：`(^.^) (^_^)`
出力：`(@o@) (@_@)`

参考：顔文字図書館 (`http://www.kaomoji.com/kao/text/`)

演習 8-15
演習 7-18 で，引数の配列の要素数によらず，最大値を返すにはどうしたよいか．(参考：演習 6-19)

演習 8-16
文字列の配列を引数とし，それらのうち最大の長さの文字列を返すメソッドを作ってみよう．

5　応用（余力のある人向け）

5.1　メソッドが自分自身を呼ぶ（再帰呼び出し）

メソッド内でそのメソッド自身を呼ぶことができる．これを再帰呼び出しという．プログラミングにおけるとても面白い考え方である．

演習 8-17
子うさぎが 1 羽いる．1 年後に子うさぎは親うさぎになる．親うさぎは 1 年後に子うさぎを産む．こうして年々うさぎが増えていくと 10 年後には何羽になっているだろうか？まず，親を R，子を r として紙に書いてみよう．

最初	r
1 年後	R
2 年後	Rr
3 年後	RrR
4 年後	
5 年後	
6 年後	
7 年後	
8 年後	
9 年後	
10 年後	

演習 8-18
以下のカッコを埋めよう．
親うさぎの数は，前年の親うさぎがそのまま残るのと，子うさぎが親になるので，

$$\begin{aligned} \text{親うさぎの数} &= \text{前年の親うさぎの数} + \text{前年の子うさぎの数} && (1)\\ &= (\underline{\qquad\qquad\qquad\qquad\qquad\qquad}) && (2) \end{aligned}$$

となる．
一方，子うさぎの方は，前年の親うさぎが 1 羽ずつ産むので，

$$\begin{aligned} \text{子うさぎの数} &= \text{前年の親うさぎの数} && (3)\\ &= (\underline{\qquad\qquad\qquad\qquad\qquad\qquad}) && (4) \end{aligned}$$

となる．なお，式 (4) は式 (2) から導かれる．
結局，『**うさぎの数は，(_____) と (_____) の和となる**』ことが分かる．ただし，最初と 1 年後は 1 羽ずつである．

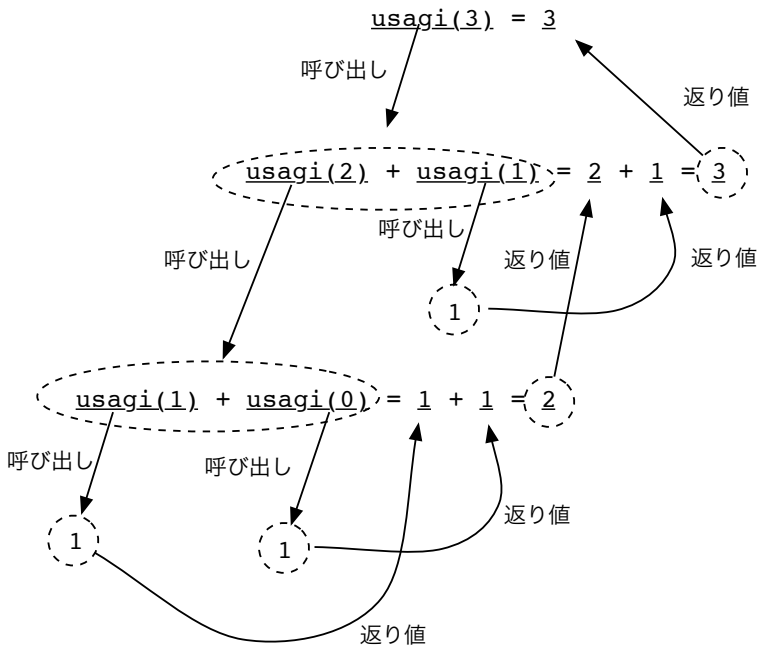
これをプログラムにすると以下のようになる．

```
-----------------------------------------------
# フィボナッチ数で増えていくうさぎの数を求めるメソッド
#   引数   n: 年数
#   返り値 n 年後のうさぎの数
#
def usagi(n)
  if n == 0 || n == 1
    return 1
  else
    return usagi(n - 1) + usagi(n - 2)
  end
end

print("10 年後にうさぎは", usagi(10), "羽だ!\n")
print("紙に書かせるなんてひどい！\n")
-----------------------------------------------
```

return usagi(n - 1) + usagi(n - 2) のところが再帰呼び出しである．メソッド usagi の中から usagi を呼び出している．ただし，引数に n - 2 があるため，n が 0 と 1 の時は，定義しておく必要がある．(数学が好きな人は，漸化式を思い出してもらえれば同じことである．)

これを n = 3 の場合の呼び出しと返り値の返り方の関係を図示すると下図のようになる．

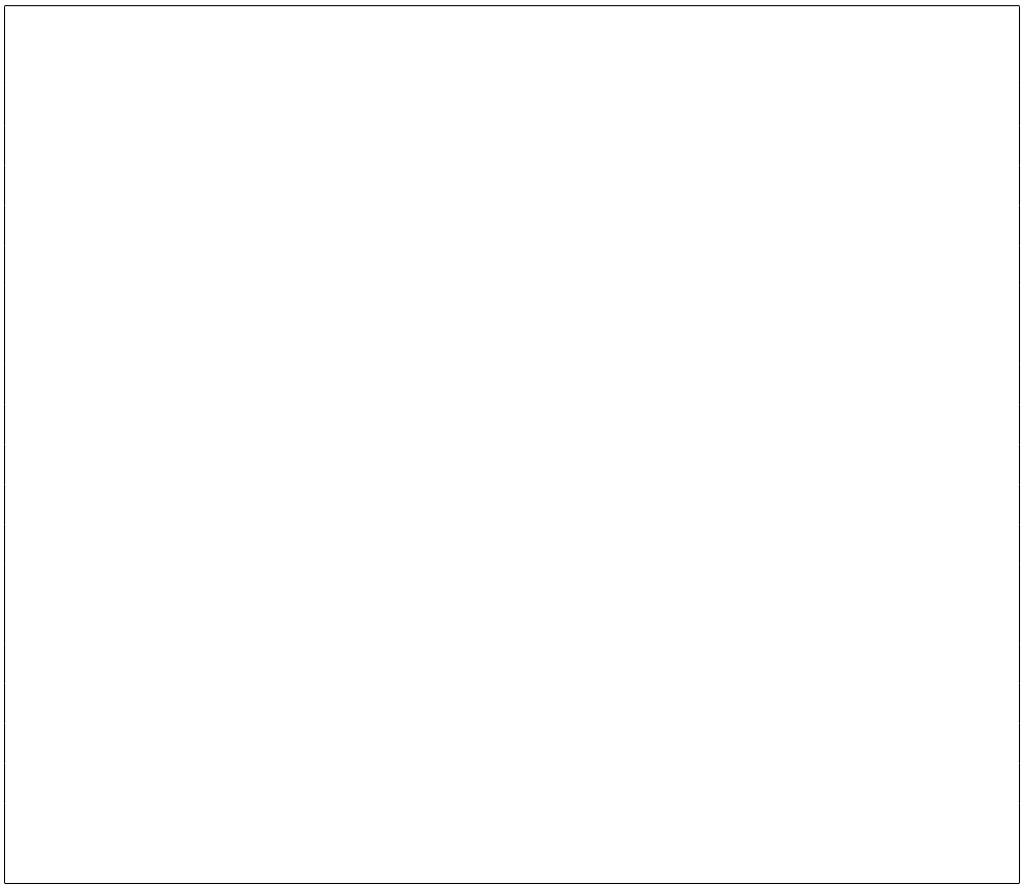


なお，このようにして求められる数列 1, 1, 2, 3, 5, 8, 13,... はフィボナッチ数列と呼ばれている．

ついでに補足しておくと，このうさぎ物語にオスうさぎは出てこない．

演習 8-19

`usagi(4)` の場合について図示してみよう．



演習 8-20

[**ハノイの塔**]「3 本の棒と，真ん中に穴のある大きさの異なる円盤 13 枚がある．初め，左の棒に大きい順に 13 枚穴を通して積み上げられている．この円盤を全て右の棒へ移動して，同じように大きい順に積み上げたい．ただし，円盤は他の棒にしか移動できず，移動できるのは 1 度に 1 枚である．しかも，自分より小さい円盤の上に積み上げることはできない．どのような動かし方をしたらよいか．手順を示せ.」

よく紹介される再帰呼び出しの問題である．ちょっと考えるとこんなことがプログラムでできるか思いつかないかもしれない．

まずは，次のように考えてみよう．

左の 13 枚を右に移動する手順は以下の通りである．

1. 左の 12 枚を中央に移動する．
2. 左の一番大きい円盤を右へ移動する．
3. 中央の 12 枚を右へ移動する．

絵に描いてみよう．

1
2
3

1 と 3 は枚数が 12 枚になっているだけで，もともとの問題である左の 13 枚を右に移動する手順と同じである．プログラムは以下のようになる．メソッド hanoi(n，from，to，tmp) は，n 枚の円盤を from 棒から to 棒へ tmp 棒を補助として使って移動する手順を求める．

```
-----------------------------------------------
# ハノイの塔を解く手順を表示するメソッド
#
#   引数   n:    円盤の数
#          from: 最初に円盤が積み上げられた棒の名前
#          to:   移動先の棒の名前
#          tmp:  作業用の棒の名前
#   返り値 なし
#
def hanoi(n, from, to, tmp)      # n 枚を from 棒から to 棒へ
  if n > 0
    hanoi(n - 1, from, tmp, to)        # n-1 枚を from 棒から tmp 棒へ
    print(n, "番目の円盤を" + from + "から" + to + "へ移動する\n")
    hanoi(n - 1, tmp, to, from)        # n-1 枚を tmp 棒から to 棒へ
  end
end
```

```
hanoi(3, "左", "右", "中")
hanoi(13, "左", "右", "中")
-----------------------------------------------
```

演習 8-21

階乗は以下のように定義される．再帰呼び出しを使って階乗を求めるメソッドを定義しよう．適切なコメントもつけるように．

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

```
-----------------------------------------------
# 階乗を計算するメソッド
#   引数   ?????
#   返り値 ?????
#
def factorial(n)
  if n > 1
    return ???
  else
    return 1
  end
end

print(factorial(3))
-----------------------------------------------
```

演習 8-22
組み合わせの定義は次の通りである．組み合わせを求めるメソッドを定義して使ってみよう．
($_5C_3 = 10, {}_{10}C_7 = 120$)

$$ {}_nC_r = \begin{cases} 1 & r = 0 \text{ または } n = r \\ {}_{n-1}C_r + {}_{n-1}C_{r-1} & \text{それ以外} \end{cases} $$

```
-----------------------------------------------
# 組み合わせを計算するメソッド
#   引数   ?????
#          ?????
#   返り値 ?????
#
def combination(n, r)
  if ???
    return 1
  else
    return ???
  end
end

print(combination(5, 3))
-----------------------------------------------
```

5.2 その他

演習 8-23
2 つの正の整数 x, y を引数とし，ベキ乗 x^y を返り値とするメソッドを定義しよう．結果は `x ** y` と比較せよ．

演習 8-24
x, y が正の整数でないときにはどうなるか？

演習 8-25
球の体積を求めるメソッドを作成しよう．ただし，必要な計算のうち，ベキ乗の計算は演習 8-23 で作成したメソッドを使うこと．

演習 8-26
変数名を引数とし，本演習での ruby コーディング規約の命名規約に違反しているかどうかをチェックして，良い変数名を返すメソッド
まずは，以下のことをやってくれるようにしてみよう．

1. 記号類が含まれていたら，‘_’ で置き換えたものを推薦してくれる．
2. 大文字が含まれていたら，小文字で置き換えたものを推薦してくれる．