基礎プログラミングII

第2回　メソッド定義と効果的活用（関数的処理）

メディア情報コース
平居　悠（ひらい　ゆたか）

到達目標

プログラミングを用いた実践的なデータ処理と情報システムの構築

- 定式化された処理を**関数の形**で記述し利用することができるようになる
- 再帰などの**アルゴリズム**を理解し問題に適用できるようになる
- 基礎的な**CGI**の仕組みの理解を通して**Webインターフェース**を設計できるようになる
- 多様な**社会事象への適用**を設計できるようになる
- 現実社会の課題に対応する**情報システム**を設計・作成できるようになる
- **生成AI**を学習の道具として利用できるようになる

前回

第１回	生成AIを効果的に利用した学習法
第２回	メソッド定義と効果的活用（関数的処理）
第３回	メソッド定義と効果的活用（データ集合処理）
第４回	メソッド定義と効果的活用（再起的アルゴリズム）
第５回	変数のスコープ・クラス設計（変数のスコープ規則）
第６回	変数のスコープ・クラス設計（クラス定義）
第７回	CGIと情報システム（CGIの基本要素）
第８回	CGIと情報システム（持続的な値のやり取り）
第９回	CGIと情報システム（実践的処理）
第１０回	チーム作品作成
第１１回	チーム作品の構築・trr試験
第１２回	チーム作品発表・評価
第１３回	合同成果発表会
第１４回	期末試験

前回の目標

生成AIを学習の道具として利用できるようになる。

前回学んだこと

1. 基礎プログラミングIの復習
2. 生成AIを活用した学修
3. 本日の課題

使用の流れ

1. 自分の**前提知識レベルを示す**。
2. 得たい答えが少なくなるように**条件を多めに**示した質問文を書く。
3. 出てきた答えからより厳密な**専門用語を引き出して**それを含む質問文を書く。
4. 得られた答えにあるキーワードをGoogle等の検索サイトで探し、**著者の明確**な（大学の授業等）資料を選んで**正確性を確認**する。

前回の問い

- 基礎プログラミングでは何を学んだか？
  - Rubyを通じて計算機上での情報の取り扱い方の基礎を学んだ。
- 生成AIを学習にどのように活用すれば良いか？
  - 自分前提知識を示す文を事前に入れて質問する。

タイピング練習スケジュール

第２回　　ホームポジション
第３回　　ローマ字
第４回　　英語初級
第５回　　日本国憲法
第６回　　ホームポジション
第７回　　ローマ字
第８回　　英語初級
第９回　　日本国憲法
第１０回　日本国憲法
第１１回　日本国憲法（trr試験、合格点：200点）

タイピングの練習 (ホームポジション)

1. ブラウザを起動し、https://www.koeki-prj.org/trr/ に繋ぐ。
2. 学籍番号（Cは大文字、省略なし８桁）を入力する。
3. Koeki MAILに届いたパスコードをPasscode: 欄に入力する。

ホームポジション


左手でタイプするキー
右手でタイプするキー
小指
薬指
中指
人差指
人差指
中指
薬指
小指
漢字
Back space
Tab
Enter
Caps Lock
Shift
Shift
Ctrl
Fn
Alt
Space
Alt
Fn
Ctrl
左手の人差指から小指までのホームポジション
両手の親指のホームポジション
右手の人差指から小指までのホームポジション

今回

第１回　生成AIを効果的に利用した学習法
第２回　メソッド定義と効果的活用（関数的処理）
第３回　メソッド定義と効果的活用（データ集合処理）
第４回　メソッド定義と効果的活用（再起的アルゴリズム）
第５回　変数のスコープ・クラス設計（変数のスコープ規則）
第６回　変数のスコープ・クラス設計（クラス定義）
第７回　CGIと情報システム（CGIの基本要素）
第８回　CGIと情報システム（持続的な値のやり取り）
第９回　CGIと情報システム（実践的処理）
第１０回　チーム作品作成
第１１回　チーム作品の構築・trr試験
第１２回　チーム作品発表・評価
第１３回　合同成果発表会
第１４回　期末試験

今日学ぶこと

1. メソッドの定義
2. メソッドの利用例
3. 繰り返し処理（復習）

今日の問い

- メソッドとは何か？
- メソッドはどのように利用できるか？

今日の目標

メソッドを自分で定義して使いこなせるようになる。

今日学ぶこと

1. メソッドの定義
2. メソッドの利用例
3. 繰り返し処理（復習）

メソッドとは

何かの「手順」を行い、その結果を返してくれるもの。

(例) printf, gets, to_i

メソッドとは

数学における関数のようなもの

例

$$f(x) = 2x + 1$$

f(4)を呼び出すと2×4+1を計算できる。fに与えた4のことを**引数（ひきすう）**という。

メソッドの定義方法

メソッドを定義するには**def**を用いる。

```
def メソッド名(引数リスト)
  定義本体
end
```

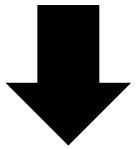
メソッドの定義例

```
def f(x)
   2 * x + 1
end
```

メソッドの定義例

```
def f(x)
```

()内の引数リストはxのみ



「メソッドfを呼ぶときは引数を1つだけ付ける」という意味。メソット定義の引数リストに書く変数は**仮引数**といい、メソッドを呼んだときに与えた値が代入される。

メソッドの定義例

```
2 * x + 1
```

メソッド定義本体。

printf(“2*%d + 1は %dです\n”, x, 2*x+1)
のようにRubyのメソッドを書いても良い。

メソッドの定義例

```
end
```

メソッド定義の終了を示す。

メソッドの定義例

method-1.rbというファイルを作成し、右のコードを書いて実行してみよう。

```
#! /usr/koeki/bin/ruby
# coding: utf-8

def f(x)
  2*x + 1
end
```

メソッドの定義例

メソッドは定義しただけでは動かない。

```
#! /usr/koeki/bin/ruby
# coding: utf-8


def f(x)
  2*x + 1
end
```

メソッドの参照と返却値

メソッドを参照するには、参照したい位置で

```
メソッド名(引数)
```

と書く。例えば、先ほど定義したfメソッドを呼ぶには、

```
f(3)
```

などとする。

method-2.rbというファイルを作成し、右のコードを書いて実行してみよう。

```
#! /usr/koeki/bin/ruby
# coding: utf-8

def f(x)
  2*x + 1
end

STDERR.puts “数値を入れて下さい。2倍して1足します。”
y = gets.to_i
printf(“%d\n”, f(y))
```

method-3.rbというファイルを作成し、右のコードを書いて実行して5を入力してみよう。

```
#! /usr/koeki/bin/ruby
# coding: utf-8

def g(x)
  a = x*2
  b = 1
  c = a + b
end

STDERR.puts “数値を入れて下さい。計算した結果を表示します。”
y = gets.to_i
printf(“%d\n”, g(y))
```

5を入力した場合

```
#! /usr/koeki/bin/ruby
# coding: utf-8

def g(x)
  a = x*2        → 10
  b = 1          → 1
  c = a + b      → 11
end

STDERR.puts “数値を入れて下さい。
計算した結果を表示します。”
y = gets.to_i
printf(“%d\n”, g(y))
```

メソッドの参照と返却値

メソッドを実行するとき最後に実行した文の値がメソッドの実行結果として返される。これを**返却値**という。

引数を２つ以上利用する場合

```
def foo(x, y)
  x + y * 2
end
```

のように引数リストで仮引数をカンマ(,)で区切って列挙する。呼び出す際はfoo(2, 5)のようにする。

引数が不要の場合

```
def bar()
  ...なんらかの内容...
end
```

のように引数リストの括弧内は空にする。

return文

メソッドの中で呼ぶと、メソッドの返却値を指定できる。

```
def daikei(jotei, katei, takasa)
  if jotei < 0 || katei < 0 || takasa < 0
    return nil
  end
  return (jotei+katei)*takasa/2
end
```

jotei, katei, takasaのいずれかが負の場合nilを返しメソッドを抜ける

仕事をするのみのメソッド

```
def hello(who)
  pirntf(“%s さんこんにちは！\n”, who)
end
```

method-4.rbというファイルを作成し、右のコードを書いて実行してみよう。

```
#! /usr/koeki/bin/ruby
# coding: utf-8

def hello(who)
  printf(“%s さんこんにちは！\n”, who)
end

STDERR.puts “あなたの名前を入れて下さい。”
you = gets.chomp!
hello(you)

STDERR.puts “あなたの近くに座っている人の名前を入れて下さい。”
friend = gets.chomp!
hello(friend)
```

今日の問い

- メソッドとは何か？
  - 何かの「手順」を行い、その結果を返してくれるもの。
- メソッドはどのように利用できるか？

今日学ぶこと

1. メソッドの定義
2. メソッドの利用例
3. 繰り返し処理（復習）

消費税計算

金額を渡すと消費税を計算するメソッド

例：金額250円、税率10%なら

tax (250, 1.10)

とする

```
#! /usr/koeki/bin/ruby
# coding: utf-8

                金額    税率
def tax(price, ratio)
  charge = price*ratio
  charge.to_i
end
```

引数のデフォルト値

特に何も指定しなければ採用される値のことを
デフォルト値という。

```
def tax(price, ratio=1.10)
  charge = price*ratio
  charge.to_i
end
```

ratioの値を省略したときは1.10として計算する。

tax-2.rbというファイルを作成し、右のコードを書いて実行してみよう。

```
#! /usr/koeki/bin/ruby
# coding: utf-8

def tax(price, ratio=1.10)
  charge = price*ratio
  charge.to_i
end

STDERR.puts “税抜き価格はいくらですか？:”
price = gets.chomp!.to_i

printf(“消費税10%%だと %d 円になります.\n”, tax(price))
printf(“でも消費税8%%だと %d 円になるんですよ.\n”, tax(price, 1.08))
```

引数省略時の注意点

引数省略時のデフォルト値は、引数リストの後ろの方からしか使えない。

OK

```
def tax(price=1000, tax=1.1)
def tax(price, tax =1.10)
```

NG

```
def tax(price=1000, tax)
```

定義したメソッドを呼ぶときは引数を順番通り渡す必要があるため、前の方の引数を省略して後ろの引数を指定できない

メソッドに配列を渡す例

配列をメソッドに渡すこともできる。

右の例では、数値が複数入っている配列を受け取り、それらの値の平均値を求める。

```
def average(score)
  sum = 0.0
  score.each do |x|
    sum += x
  end
  sum/score.length
end
```

入力した数値全ての平均値を出力するプログラム

average.rbというファイルを作成し、右のコードを書いて実行してみよう。

```
#! /usr/koeki/bin/ruby
# coding: utf-8

def average(score)  # scoreは数値のたくさん入っている配列
  sum = 0.0         #割り算する予定なので浮動小数点数にしておく。
  score.each do |x|
    sum += x
  end
  sum/score.length
end

points = []               # からっぽの配列を作る
i = 0                     # iがpointsの添え字となる
while true
  STDERR.print
  line = gets
  if line == nil
    break
  end
  points[i] = line.chomp!.to_f
  i += 1
end

printf("\n平均値は %6.2f です。\n", average(points))
```

今日の問い

- メソッドとは何か？
  - 何かの「手順」を行い、その結果を返してくれるもの。
- **メソッドはどのように利用できるか？**
  - **メソッドを定義して計算や仕事をさせることができる。**

今日学ぶこと

1. メソッドの定義
2. メソッドの利用例
3. 繰り返し処理（復習）

いろいろな繰り返し (while以外)

- **times**: 一定回数の繰り返し
- **upto, downto**: 数え上げ、カウントダウン
- **each, for**: 要素に対する繰り返し
- **step**: 要素に対する繰り返し

times: 決められた回数だけ繰り返す

繰り返したい回数を***N***とすると、以下のように記述する。

```
N.times do
  ...繰り返し本体...
end
```

shitsukoi.rbというファイルを作成し、右のコードを書いて実行してみよう。

```
#! /usr/koeki/bin/ruby
# coding: utf-8

5.times do
  puts “ねえねえ”
  puts “なんだよ”
end

puts “んー、ねえねえ”
puts “なんだっての、しつこいよ！”
```

upto, downto: 整数を数えながらの繰り返し

whileを使った場合 (復習)

```
sum = 0
i = 1
while i <= 10
  sum += i
  i += 1
end

printf(“合計は %d です\n”, sum)
```

uptoを使った場合

```
sum = 0
1.upto(10) do |x|
  sum += x
end

printf(“合計は %d です\n”, sum)
```

each, for: 配列要素すべてに対する繰り返し

配列の各要素1つ1つ取り出しながらの繰り返し

配列.each do |**変数**|
　...上記の**変数**を使った処理...
end

for **変数** in **配列**
　...上記の**変数**を使った処理...
end

合計点を計算するプログラム

```
point = [10, 20, 15, 40, 33]

sum = 0
point.each do |z|
  sum += z
end

printf(“合計は %d です\n”, sum)
```

```
point = [10, 20, 15, 40, 33]

sum = 0
for z in point do
  sum += z
end

printf(“合計は %d です\n”, sum)
```

配列を昇順にした結果を与える場合

```
配列.sort.each do |変数|
  ...上記の変数を使った処理...
end
```

step: 飛び飛びの繰り返し

1, 3, 5, 7, 9...や10, 20, 30のように数値を一定間隔で変えながら利用したい場合はstepメソッドを用いる。

```
開始値.step(終了値[, 増加値]) do |変数|
  ...変数を利用した処理
end
```

増加値は省略できて、省略した場合は1とみなされる。

数値で繰り返すときの注意事項

contdown-int.rbというファイルを作成し、右のコードを書いて実行してみよう。

```
#! /usr/koeki/bin/ruby
# coding: utf-8
c = 10
while c != 0
  printf(“%d\n”, c)
  c -= 1
  sleep 0.7
end
printf(“おしまい\n”)
```

数値で繰り返すときの注意事項

contdown-float.rb というファイルを作成し、右のコードを書いて実行してみよう。

```
#! /usr/koeki/bin/ruby
# coding: utf-8
c = 1.0
while c != 0.0
  printf(“%f\n”, c)
  c -= 0.1
  sleep 0.7
end
printf(“おしまい\n”)
```

数値で繰り返すときの注意事項

c = 0.000000のときにループが終了していない

```
1.000000
0.900000
  :
…中略…
  :
0.100000
0.000000
-0.100000
-0.200000
-0.300000
-0.400000
^Ccountdown-float.rb:7:in `sleep': Interrupt
        from countdown-float.rb:7
```

ループが終了しない理由

計算機による小数計算の誤差

計算機では内部で２進数を用いているので、小数を表すときに限られた桁数では表しきれないことがある。

例：0.1 (10進数) = 0.0001100011... (2進数)

1.0から0.1を10回引いたものは厳密な0.0にならない。

数値で繰り返すときの注意事項

小数刻みのものが必要な場合、整数を割り算して利用する。

```
#! /usr/koeki/bin/ruby
# coding: utf-8
c = 1.0
while c != 0.0
  printf(“%f\n”, c/10.0)
  c -= 1
  sleep 0.7
end
printf(“おしまい\n”)
```

今日学んだこと

1. メソッドの定義
2. メソッドの利用例
3. 繰り返し処理（復習）

今日の目標

メソッドを自分で定義して使いこなせるようになる。

課題

計算のためのパラメータ（変数）を２つ以上必要とする公式を計算するメソッドを定義し、それを利用するプログラムを書け。日常生活で使いそうな計算式などで面白いものを題材にするとよい。

1. 起動すると必要な値を読み込みメソッドを用いて計算した結果を出力する。
2. プログラム起動時にコマンドラインで与えた引数を、定義したメソッドに渡すようにする（ARGV変数から情報を受け取る）。
3. **今回習ったループ（upto, downto, for）を利用し、メソッドに引数を変動させながら順次メソッドを呼ぶようにする。**

レポート提出方法

s4基礎プロII(F)の「#02課題 (メソッド)」に指示どおり書き込む。提出後修正して良い。その場合は「編集」リンクから書き換えること。

締切：10月11日 (土)

次回

第１回　生成AIを効果的に利用した学習法
第２回　メソッド定義と効果的活用（関数的処理）
第３回　メソッド定義と効果的活用（データ集合処理）
第４回　メソッド定義と効果的活用（再起的アルゴリズム）
第５回　変数のスコープ・クラス設計（変数のスコープ規則）
第６回　変数のスコープ・クラス設計（クラス定義）
第７回　CGIと情報システム（CGIの基本要素）
第８回　CGIと情報システム（持続的な値のやり取り）
第９回　CGIと情報システム（実践的処理）
第１０回　チーム作品作成
第１１回　チーム作品の構築・trr試験
第１２回　チーム作品発表・評価
第１３回　合同成果発表会
第１４回　期末試験

事前課題

quiz ruby-enumを３回以上行い、最も早くクリアした記録の結果をs4基礎プロII(F)の「#03 事前課題 (quiz enum)」に貼り付ける。