講義使用向けの文書形態であることに注意して利用してください。 CC-BY での取り扱いとします。

2026-09-27 広瀬雄二

第10講 メソッド

◆メソッドを定義する

◇メソッドとは

これまで作成したプログラムでは *変数* をたくさん使ってきた。変数は自分で好きな値を代入すると、それを保持してくれて、あとで参照したときにその値を返してくれる。

変数が数値、文字列、などのある時点の「値」のみを返してくれるものである一方、*メソッド* は何かの「手順」を行ない、その結果を返してくれる。これまでよく利用してきた printf, gets, to_i などはいずれもメソッドである。printf はフォーマットを決めて文字列を出力してくれる、getsは決められた場所からデータを読んでくれる、 to_i は元の値を整数に変換してくれる。このような *働き*をするのがメソッドである。

メソッドを理解する上で、数学で習った「関数」を思い浮かべると分かりやすい。

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

という定義をしておけば、*f(4)* のように呼び出すことで、2×4 + 1 を計算したことになる。ここで、f に与えた *4* のことを*引数(ひきすう)* という。関数は、引数に与えられた数値を *x* に代入して計算を行なう。メソッドでもこの点は同様である。

◇メソッドの定義方法

メソッドを定義するには *def*を利用する。

```
def メソッド名(引数リスト)
  定義本体
end
```

簡単なメソッドを定義してみよう

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

これは、先ほどの *f(x) = 2x + 1* と同じ内容なので、だいたい意味は分かるだろう。

- `def f(x)`
  メソッドとして f という名前のメソッドを定義する。括弧（ ）の中に書くのは引数リストという。そして、ここには x とだけ書いてある。これは、

  > メソッドfを呼ぶときには引数を1つだけ付けてね

  ということを意味している。つまり、別の場所でfを呼びたいときには f(5) のように必ず一つの

引数が必要となる。メソッド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。つまり、 f(5) と呼んだ場合 x には5が代入される。

- 2 * x + 1

def……endの間に書くのがメソッドの定義本体である。ここには、Rubyの文を何でも書くことができる。ここではたまたま、数学の関数のように数式を書いたが、

```
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。

- end

endでメソッド定義の終了を示す。

この例をファイルに保存しよう。

method-1.rb

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

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

実際に実行してみよう。

```
% chmod +x method-1.rb
% ./method-1.rb
```

なにが起きるだろうか。

◇メソッドの参照と返却値

メソッドは定義しただけでは動かない。プログラムの別の箇所で参照しなければなにもせずに終わってしまう。これは数学の答案で最初の方に

f(x) = 3x + 1 とする

と書いたのに、別のところで *f(x)* を使わなければ意味がないのと同じである。

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)
```

と書くだけで良い。たとえば、先ほど定義した fメソッドを呼ぶには、

```
f(3)
```

のようにすればよい。プログラムを改良してみよう。

method-2.rb

```
#!/usr/local/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))
```

f(y) の部分が、定義したメソッドの呼び出しである。ここでは、直前の行で読み込んだ数値をyに入れていて、それをfメソッドに渡している。f メソッドでは、仮引数 x に入力した値を代入して計算した結果を返す。

では、次のようなプログラムはどんな結果を返すだろうか。

method-3.rb

```
#!/usr/local/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))
```

今度は g メソッドに3行の式がある。

- a = x*2
- b = 1
- c = a + b

のどの行が g(x) の値になるのだろうか? たとえば5を与えたら

- a = 5*2 → 10
- b = 1 → 1
- c = a + b →11

10か1か11が返ると予想できる。実際に実行してみよう。

```
% chmod +x method-3.rb
% ./method-3.rb
数値を入れて下さい。計算した結果を表示します。
5
11
```

このように、メソッドでは、そのメソッドを実行するときに *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。

また、引数を2つ以上利用するときは

```
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。この例の場合呼出し側では2個の値を渡す必要がある。つまりfoo(2, 5)のように呼ぶ。引数が不要の場合は

```
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

メソッドの中途で直ちに呼び主に制御を返すこともできる。

```
def daikei(jotei, katei, takasa)
  if jotei < 0 || katei < 0 || takasa < 0
    # jotei, katei, takasa いずれかが負なら nil を返す
    return nil
  end
  return (jotei+katei)*takasa/2
end
```

上記の例では、jotei, katei, takasa いずれかが負の場合に直ちに nil を返しメソッドを抜ける。

◇仕事をするのみのメソッド

必ずしも数学の関数のように「結果の値」を返してもらう必要はない。たとえば、次のようなメソッドも考えられる。

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

このメソッドでは、仮引数として who をもらい、それが文字列だとして 「さんこんにちは!」を後置したメッセージを出力するというだけのものである。実際に利用してみよう。

method-4.rb

```
#!/usr/local/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)
```

実際に実行してみよう。

```
% chmod +x method-4.rb
% ./method-4.rb
あなたの名前を入れて下さい。
taro
taro さんこんにちは!
あなたの近くに座っている人の名前を入れて下さい。
hanako
hanako さんこんにちは!
```

メソッドは、決まった式の計算をしてもらうこともできるし、決まった仕事をしてもらうこともできる。

> ちなみに、printf メソッドも仕事をした結果として値(nil)を返している。したがって、hello メソッドの返却値も nil となる。ただし、呼出し側の方で、 helloメソッドの返す値を全く利用していないのでこの値は捨てられることになる。

◆メソッドの利用例

もう少し複雑なメソッドを定義してみよう。

◇消費税計算

金額を渡すと消費税を計算するメソッドを作成してみよう。税率として「10%固定」というのでは安物電卓と変わらなくて悔しいので税率も選べるようにしよう。

tax-1.rb

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

def tax(price, ratio)   # priceに値段、ratioに税率 をもらう
  charge = price*ratio
  charge.to_i
end
```

ここで定義したtaxメソッドを参照するときは、第1引数に金額、第2引数に税率を指定する。

```
tax(250, 1.10)
```

これでtaxメソッドで price=250, ratio=1.10 となり、税率10%で計算した結果を返してくれる。

◇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。たとえば、消費税は大抵の場合10%だが、食料品など特定商品にに軽減税率 8%が適用されたり、金券のように非課税(0%)となるものもある。 *税率を指定しなかったら10%(1.10)ということにしておく* のようにメソッド定義デフォルト値を定めることができる。次のようにする。

```
def tax(price, ratio=1.10)   # priceに値段、ratioに税率 をもらう
  charge = price*ratio       # ratioを省略したときは 1.10
```

```
  charge.to_i
end
```

このように、仮引数の変数の後に 「*=値*」を書くと、それが引数を省略した場合のデフォルト値になる。実際に次のプログラムを動かしてみよう。

tax-2.rb

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

def tax(price, ratio=1.10)   # priceに値段、ratioに税率 をもらう
  charge = price*ratio       # ratioを省略したときは 1.10
  charge.to_i
end

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

printf("消費税5%%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。

○ `def tax(price=1000, tax=1.1)`
○ `def tax(price, tax=1.10)`
× `def tax(price=100, tax)`
× `def foo(x=3, y=2, z)`
○ `def foo(x, y=2, z=3)`
× `def foo(x, y=2, z)`

なぜなら、定義したメソッドを呼ぶときには引数を順番通り渡さねばならないため、前の方の引数を省略して後ろの引数を指定することはできないからである。

◇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。次のメソッドは、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ものである。

```
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
```

これを利用して、入力した数値全ての平均値を出力するプログラムを作ると以下のようになる。

average.rb

```
#!/usr/local/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の添字となる
while true
  STDERR.print "数値を入力して下さい: "
  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))
```

実際に実行してみよう。データの入力の最後には[C-d]をタイプする。

```
% ./average.rb
数値を入力して下さい: 1
数値を入力して下さい: 2
数値を入力して下さい: 3
数値を入力して下さい: 4
数値を入力して下さい: [C-d]
平均値は    2.50 です
```

◆いろいろな繰り返し

プログラム中、何回も同じ処理を繰り返すときはこれまで主に while を利用してきた。そのほかにも用途に応じた繰り返しの構文がある。次講から頻繁に登場する多数のデータの本格的処理をスムーズに進めるためには、様々な繰り返し構文をしっかりと理解して自在に操れるようにしておく必要がある。ここで「繰り返し」のための構文を整理しておこう。

- 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

◇決められた回数だけ繰り返す

繰り返したい処理の回数だけが重要な場合は、整数に備わっている *times* メソッドで繰り返す。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
```

以下のプログラムは、timesを使って5回繰り返す例である。

shitsukoi.rb

```
#!/usr/local/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!"
```

このプログラムをshitsukoi.rbというファイル名で保存して実行してみよう。

```
% chmod +x shitsukoi.rb
% ./shitsukoi.rb
ねえねえ
なんだよ
ねえねえ
なんだよ
ねえねえ
なんだよ
ねえねえ
なんだよ
ねえねえ
なんだよ
んー、ねえねえ
なんだっての、しつこいよ!
```

◇整数を数えながらの繰り返し

たとえば1…10までの和を計算したりするような場合には、整数を1つずつ増やしていく必要がある。そのような場合には、これまでのプログラムでは以下のようにしていた。

```
sum = 0
i = 1
while i <= 10
  sum += i
  i += 1
end
printf("合計は %d です\n", sum)
```

このループは、要するにiを 1 から上限の数まで変えながらsumに足していきたいものである。このような場合は、整数に備わっているメソッド upto を利用すると以下のようにすっきり書ける。

upto-sum.rb

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

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

printf("合計は %d です\n", sum)
```

upto とは逆の、1ずつ値を減らしていく downto メソッドもある。

もうひとつ、連続した整数を集合とみなして繰り返すには for と*整数範囲*を用いて以下のように繰り返しを表現する。

```
for 変数 in 整数範囲
  …繰り返し本体…
end
```

たとえば、1から10までの数を足すには以下のようにする。

```
sum = 0
for i in 1..10
  sum += i
end
printf("1から10までの和は%dです\n", sum)
```

◇配列要素すべてに対する繰り返し

配列の各要素を1つ1つ取り出しながらの繰り返しには、 each または for を使う。

```
配列.each do |変数|
  ……上記の変数を使った処理……
end
# または
for 変数 in 配列
  ……上記の変数を使った処理……
end
```

実際には以下のようになる。

each-sum.rb

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

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

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

printf("合計は %d です\n", sum)
```

for-sum.rb

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

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)
```

この場合の変数 z には、point 配列の各要素の値が順次代入される。一回目は z=10、二回目は z=20、三回目は z=15、四回目は z=40、五回目は z=33、と代入された上でループ内部が実行される。

上記の例では10, 20, 15, 40, 33が全てsumに足されるので

```
合計は 118 です
```

と出力される。

◇飛び飛びの繰り返し

1, 3, 5, 7, 9, ... とか、10, 20, 30, ... のように、数値を間隔一定で変えながら利用したい場合は、整数に備わっている step メソッドを利用する。書式は以下のとおり。

```
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。実際の利用例は以下のようになる。

- 【例1】 1…99の奇数を足したい場合（step99.rb）

```
#!/usr/local/bin/ruby
# coding: utf-8
sum = 0
print "0"
1.step(99, 2) do |i|
  printf("+%d", i)
  sum += i
end
printf("は %d です\n", sum)
```

- 【例2】 カウントダウンをしたい場合（countdown.rb）

```
#!/usr/local/bin/ruby
# coding: utf-8
print "新年まで"

10.step(1, -1) do |sec|
  if sec <= 3
    printf("%c", 0x7)   # CTRL-G (ベル)のコードは7
  end
  printf("%d秒前\n", sec)
  sleep(1)
end
puts "-" * 79           # 文字列に掛算 * を使うとその回数繰り返す
system "banner 'A Happy' 'New Year!'"
```

この例で利用している system メソッドは、 Rubyプログラムの中から、外部コマンドを起動するものである。 KTermで

```
% banner 'A Happy' 'New Year!'
```

く同じ結果が得られる。

◇数値で繰り返すときの注意事項

次の二つのプログラムを実際に実行してみよう。

countdown-int.rb

```
#!/usr/local/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")
```

こちらは予想どおりの結果となる。

```
10
9
8
7
6
5
4
3
2
1
おしまい
```

10から0までの整数の繰り返しを、1.0から0.0までの小数繰り返しに変えたプログラムを動かしてみる。

countdown-float.rb

```
#!/usr/local/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")
```

期待に反して、実行すると以下のようになる。

```
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
```

whileループでは、「cが0.0と等しくないなら繰り返せ」という条件を指定しているのに、実際にはc=0.000000のときにループが終了していない。

これは、計算機による小数計算の誤差に起因する。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切れないことがある。 3分の1を10進数で表そうとすると 0.333333... となって、途中で切り捨てなければならないのと同様、10進数の0.1を2進数で表すと循環小数になり途中で切り捨てなければならなくなる。そのため0.1を10回積み重ねても、切り捨て誤差も積み重なって正確な1.0にはならない。それゆえ、1.0から0.1を10回引いたものは厳密な0.0にはならないのである。

数値を使ってループを作るときは、ループ変数に浮動小数点数を使うのは避けなければならない。どうしても小数刻みのものが欲しければ、整数を割り算して利用する。たとえば、1.0から0.0でカウントダウンしたければ

countdown-f2.rb

```
#!/usr/local/bin/ruby
# coding: utf-8
c=10
while c != 0
  printf("%f\n", c/10.0)
  c -= 1
```

```
  sleep 0.7
end
printf("おしまい\n")
```

のように、整数のループ変数を割って利用する。

練習問題 10

1. 数学的関数 $f(x) = x^2+3x-1$ に相当するメソッドfを定義せよ。定義の書き始めは、

   ```
   def f(x)
   ```

   とする。
2. 底辺(cm)と高さ(cm)を意味する引数(2つ)を受け取り、長方形の面積を計算した結果を返すメソッドrectangleを定義せよ。定義の書き始めは

   ```
   def rectangle(width, height)
   ```

   とする。
3. 1つの引数を*y*として受け取り、
   - もし*y*が4で割り切れないかまたは、 *y*が100で割り切れてなおかつ*y* が400で割り切れないなら
     → 365 を返す
   - 上記に当てはまらない場合は
     → 366 を返す

   ようなメソッドleapを定義せよ。
4. 2つの引数*element, content*(ともに文字列)を受け取り、 *element*で指定したHTML要素として*content* を括った文字列を返すメソッドtagify を作成せよ。つまり、

   "<" *elementの値* ">" *contentの値* "</" *elementの値* ">"

   のように文字列を結合したものを返す。たとえば、 tagifyメソッドに "pre", "hogehoge" を渡すと以下の値が返る。

   ```
   tagify("pre", "hogehoge")
   => "<pre>hogehoge</pre>"
   ```
5. 前問で作成したtagifyメソッドを利用し、入力した文字列を pre 要素として出力するプログラム pre.rbを作成せよ。
6. 値に配列を持つ引数をarrとして受け取り、その先頭要素と最終要素の値を交換た配列を値として返すメソッド sflを定義せよ。たとえば、このメソッドに [5, 9, 3, 7] を渡すと、[7, 9, 3, 5] が返る(以下参照)。

   ```
   sfl([5, 9, 3, 7])
   => [7, 9, 3, 5]
   ```

練習問題 11

1. コマンドラインでプログラム起動時に引数として整数を与えると、その数だけ "Hello, world!" と出力するプログラム nhello.rb を作成せよ。実行例は以下のとおり。

   ```
   % ./nhello.rb
   出力回数を指定してください。
   例: ./nhello.rb 5
   % ./nhello.rb 3
   Hello, world!
   Hello, world!
   ```

```
Hello, world!
```

2. 起動時に第1引数に指定したファイルを、第2引数で指定したファイルにコピーするプログラムcopy.rb を作成せよ。たとえば、

   ```
   % ./copy.rb foo.rb foo.rb.bak
   ```

   と起動すると、foo.rbファイルの内容が foo.rb.bakにコピーされる。

3. 起動時に第1引数に指定したファイルのバックアップファイルを作るプログラムbackup.rbを作成せよ。指定したファイル名が foo だとすると以下のルールでバックアップファイルの名前を決定するものとする。

   - 元のファイル名の後に .1 を付けたものにする
   - ただし foo.1 が既に存在したら数字を1増やす
   - ただし foo.2 が既に存在したら数字を1増やす
   - 以下同様…既存ファイルがない番号まで foo.*N* を増やす

   なお、数を含んだ文字列を生成するにはsprintfを利用するとよい。たとえば、

   ```
   name = sprintf("%s-%d", x, i)
   ```

   とすると、xに代入されている文字列とiに代入されている整数をハイフンで繋いだ文字列がnameに代入される。

4. 前問のbackup.rbを改良し、コマンドラインで与えた複数のファイルのバックアップを取れるようにした nback.rb を作成せよ。たとえば、

   ```
   % ./nback.rb foo.rb bar.c hoge
   ```

   と起動すると、foo.rb bar.c hoge、3つのファイルのバックアップファイルが一気に作れる。

第11講 配列とハッシュの応用

◆配列処理メソッド

計算機のプログラムの役割の大きな柱のひとつがデータ処理である。多くの数のデータを処理できるプログラムは有用である。 Rubyでは配列を利用することで、多くのデータを処理するプログラムを手軽に作ることができる。また、Rubyでは配列/ハッシュ/文字列には共通したメソッドが用意されている。それらを複合的に覚えていこう。

◇集合的な処理を行なうメソッド

配列と文字列はどちらも要素の並びと捉えられる。

配列 の場合

x = [5, 6, 3, 2, 9] の格納イメージ

x				
x[0]	x[1]	x[2]	x[3]	x[4]
5	6	3	2	9

文字列の場合

s = "Hello" の格納イメージ

s				
s[0]	s[1]	s[2]	s[3]	s[4]
H	e	l	l	o

いずれも添字0から始まる入れ物に順序よく収まっているものと考えると、共通の概念で処理できるものがいくつか存在する。まずは共通したメソッドを挙げる。

- *+ other* - 連結・結合

  メソッド +は、2つのオブジェクトを繋げた*新しい*配列/文字列を返す。

  配列の場合

  元の配列と別の配列 *other* を繋げた*新しい* 配列を返す。元の配列は破壊されない。

```
x = [3, 4, 1, 2]
y = [9, 8, 7]
```

  としたとき

```
x + y
 => [3, 4, 1, 2, 9, 8, 7]
```

  となる。xとyの値は変わらない。

```
x
 => [3, 4, 1, 2]
y
 => [9, 8, 7]
```

  文字列の場合

  元の文字列と別の文字列 *other* を繋げた*新しい* 文字列を返す。元の文字列は破壊されない。

```
s = "abc"
t = "hoge"
```

  としたとき

```
s + t
 => "abchoge"
```

  となる。sとtの値は変わらない。

```
s
 => "abc"
t
 => "hoge"
```

- *<< obj* - 末尾への破壊的要素追加

  メソッド << は、末尾に新しい要素を追加する。元の配列や文字列自体が変更される。

  配列の場合

```
x << 5
```

  文字列の場合

```
s << "hoge"
```

```
 => [3, 4, 1, 2, 5]
```

xの値も変わる。

```
x
 => [3, 4, 1, 2, 5]
```

```
 => "abchoge"
```

sの値も変わる。

```
s
 => "abchoge"
```

このように、元の(配列の)値を直接書き換える操作を *破壊的操作*という。

- *concat*（*other*） - 配列の破壊的結合

  引数に指定した配列 *other*を、元の配列の末尾につなげる。

```
y = [9, 8, 7]
x.concat(y)
 => [3, 4, 1, 2, 9, 8, 7]
```

xの値も変わる。

```
x
 => [3, 4, 1, 2, 9, 8, 7]
```

yの値は変わらない

```
y
 => [9, 8, 7]
```

文字列にも concat メソッドは使用でき、 << と同じ働きをする。

- *each ブロック* - 配列/ハッシュの各要素に対するブロックの評価

  配列/ハッシュの各要素に対して後続する*ブロック*を繰り返し評価する。

配列の場合

要素を受ける1つのブロック変数で繰り返す。

```
x = [3, 4, 1, 2, 5]
x.each do |i|
 # この例の場合
 # i=3, i=4, i=1, i=2, i=5 で5回繰り返す
end
```

これは、for によるループに書き換えることができる。

```
for i in x
 ……各要素を変数 i に代入して繰り返す……
end
```

ハッシュの場合

key と value を受ける2つのブロック変数で繰り返す。

```
h = {"りんご" => 150, "みかん" => 30}
h.each do |item, price|
  # この例の場合
  # 「item="りんご", price=150」と
  # 「item="みかん", price=30」の組で
  # 2回繰り返す
end
```

これは for によるループに書き変えることができる。

```
for item, price in h
 ……各key, valueを変数 item, price
        に代入して繰り返す……
end
```

- *[nth]* - nth番目の取り出し

  配列/文字列の添字 *nth* の位置の値を取り出す。添字の先頭は0(ゼロ)、末尾から数えることもできてその場合は -1 から負の数で指定する。範囲外指定には nil が返る。

配列の場合

```
x = [3, 1, 2, 4]
x[0]
 => 3
x[-2]
 => 2
x[99]
 => nil
```

文字列の場合

```
s = "hoge"
s[0]
 => "h"
s[-2]
 => "g"
s[99]
 => nil
```

- *[start..last]*

  [start, len] - 部分の取り出し

  配列/文字列の部分集合を得る。開始添字位置(*start*)と、終了位置(*last*)または長さ(*len*)で指定する。

  配列の場合

```
z = [1, 3, 5, 6, 2, 10, 3, 5]
z[2..5]
 => [5, 6, 2, 10]
z[2..-1]
 => [5, 6, 2, 10, 3, 5]
z[-2..-1]
 => [3, 5]
z[0, 7]
 => [1, 3, 5, 6, 2, 10, 3]
```

  文字列の場合

```
u = "C1154320"
u[2..5]
 => "1543"
u[2..-1]
 => "154320"
u[-2..-1]
 => "20"
u[0, 7]
 => "C115432"
```

- *index*(*val*) - 指定要素の発見

  配列/文字列から特定の値を探し位置を返す。

  配列の場合

  配列の中に、引数に指定した値 *val* に等しいものがあるか調べ、最初に見つかった位置(添字)を返す。見つからなかった場合は、 nil を返す。

```
x = [3, 1, 2, 4]
x.index(3)
 => 0
x.index(4)
 => 3
x.index(5)
 => nil
```

  文字列の場合

  文字列の中に、引数に指定した部分文字列 *val* に等しいものがあるか調べ、最初に見つかった位置(添字)を返す。見つからなかった場合は、nil を返す。

```
m = "koekitaro@e.koeki-u.ac.jp"
m.index("@")
 => 9
m.index("koeki-u.ac.jp")
 => 12
m.index("f.koeki-u.ac.jp")
 => nil
```

- *length* - 配列/文字列の長さを返す

  size - lengthと同じ

  配列/文字列の長さ(要素数)を返す。

  配列の場合

  配列の要素数を返す。

```
x = [3, 1, 2, 4]
x.size
 => 4
x.length
 => 4            (同じ)
```

  文字列の場合

  文字列の長さを返す(バイト数ではない)。

```
"abcde".length
 => 5
"あいうえお".length
 => 5
"あいうえお".bytesize
 => 15           (UTF-8の場合)
```

  バイト数は bytesize メソッドで得られる。

- *reverse*

  reverse! - 逆順にする。

  配列/文字列の中味を全て逆順に並べたものを返す。

  配列の場合

```
x = [3, 1, 2, 4]
```

  文字列の場合

```
s = "Hello"
```

```
x.reverse
 => [4, 2, 1, 3]
```

xの値は変わらない。

```
x
 => [3, 1, 2, 4]
```

reverse! は元の配列を破壊的に変更する。

```
x.reverse!
 => [4, 2, 1, 3]
x
 => [4, 2, 1, 3]
```

```
s.reverse
 => "olleH"
```

sの値は変わらない。

```
s
 => "Hello"
```

reverse! は元の配列を破壊的に変更する。

```
s.reverse!
 => "olleH"
s
 => "olleH"
```

◇配列用のメソッド

配列に固有のメソッドを示す。

- *join*

  join(*sep*) - 配列を結合して文字列化

  配列の各要素の間に文字列 *sep* を挟んで連結した文字列を返す。*sep* を省略した場合は変数 $, の値が利用される。$,のデフォルト値はnil で、その場合挟む文字列は空文字列("")である。連結時に、配列の各要素は文字列化される。

```
x.join("/")
 => "3/1/2/4"
x.join(", ")
 => "3, 1, 2, 4"
$,
 => nil
x.join()
 => "3124"
```

- *shift* - 先頭要素の取り出し

  配列の先頭要素を取り出し、その値を返す。配列はひとつずつ前に詰められる。

```
x = [3, 1, 2, 4]
x.shift
 => 3
x
 => [1, 2, 4]
```

これを繰り返すと、配列の要素を先頭から取り出すループが、

```
while i=x.shift
  printf("%d\n", i)
end
```

のように作成でき、以下の出力が得られる。

```
3
1
2
4
```

ループから抜けたときのxの値は []となる。

- *unshift*(*val*) - 先頭への要素の追加

  配列の先頭に *val* を追加し、その配列を返す。各要素はひとつずつ後にずれる。

```
x = [3, 1, 2, 4]
```

```
x.unshift(20)
 => [20, 3, 1, 2, 4]
x
 => [20, 3, 1, 2, 4]
```

- *uniq*

  uniq! - 重複要素の削除

  配列から重複した要素を取り除き、取り除いた部分を前に詰めた配列を返す。

```
z = [1, 0, 2, 0, 3, 9, 7, 7, 1, 3, 2, 6, 5, 2]
z.uniq
 => [1, 0, 2, 3, 9, 7, 6, 5]
```

  zの値は変わらない。

```
z
 => [1, 0, 2, 0, 3, 9, 7, 7, 1, 3, 2, 6, 5, 2]
```

  いっぽう、uniq! は、元の配列を破壊的に書き換えて、重複したものを取り除いた配列を返す。

```
z = [1, 0, 2, 0, 3, 9, 7, 7, 1, 3, 2, 6, 5, 2]
z.uniq!
 => [1, 0, 2, 3, 9, 7, 6, 5]
z
 => [1, 0, 2, 3, 9, 7, 6, 5]
```

- *select ブロック* - 条件を満たす要素の選択

  配列の要素をひとつずつ取り出し*ブロック* に渡し，ブロックが真(false以外)を返す要素のみからなる新規配列を返す。複数の要素から特定の条件を満たすもののみ選び出したいときに利用する。

  たとえば，z が以下の内容のときを考える。

```
z = [1, 0, 2, 3, 9, 7, 6, 5]
```

  この場合に偶数のみを含む配列を得たいときは，「2で割った余りが0に等しいもの」を選ぶことで偶数を選別できる。

```
even = z.select {|i| i%2 == 0}
 => [0, 2, 6]
```

  ブロック先頭にある |i| は，順次取り出す要素を変数 i に代入しつつ繰り返すことを意味する。i に代入された値で i%2 を計算し，それを0と比較することを繰り返している。

  数値に限らずどのような要素でも適用できる。以下の配列 f から，ひらがなで始まるもののみを選んでみる。

```
f = ["柿", "りんご", "サクランボ", "桃", "いちご"]
hira = f.select {|el| /[ぁ-ん]/ =~ el}
 => ["りんご", "いちご"]
```

- *reject ブロック*

  reject! ブロック - 条件を満たす要素の削除

  select メソッドの逆で，条件を満たす要素を削除する。reject には破壊的メソッド reject! がある。

```
f
 => ["柿", "りんご", "サクランボ", "桃", "いちご"]
f.reject{|i| /[ぁ-ん]/ =~ i}
 => ["柿", "サクランボ", "桃"]
f
 => ["柿", "りんご", "サクランボ", "桃", "いちご"]
f.reject!{|i| /[ぁ-ん]/ =~ i}
 => ["柿", "サクランボ", "桃"]
```

```
 f
 => ["柿", "サクランボ", "桃"]
```

- *sort*

  sort ブロック - 配列の内容のソート

  配列の内容を並べ換える。*ブロック* を指定しない場合は各要素を、演算子 <=> で比較して小さい順(昇順)に並べ換える。並べ換える規準を変えたいときには *引数を二つ取るブロック* を指定する。具体的には

  配列.sort {|x, y| *xとyの比較式*}

  のように書く。たとえば、大きい順(降順)に並べ換えたい場合は、

```
 z = x.sort{|x, y| y <=> x}
```

  などとする(xとyの順番に注目)。

- *sort_by ブロック* - 配列の内容の効率的ソート

  ブロックを渡す sort 同様だが、ソート基準とする値を取り出す式のみ指定する。つまり、

  配列.sort {|x| *比較すべき値を取り出す式*}

  のようにする。ハッシュの value の大小関係でソートしたいときに有用で、たとえば

```
 item = {
   "アルカリ異音水" => 150,
   "おでんつゆ" => 50,
   "マグロの煮こごり" => 500
 }
```

  という商品名vs.単価リストを持つハッシュの商品名を単価の小さい(安い)順に並べ換えるには以下のようにする。

```
 item.keys.sort_by{|x| item[x]}
```

◇文字列固有のメソッド

- *chop*

  chop!

  - 文字列の末尾削除

  元の文字列の最後の一文字を取り除いたものを返す(最後が DOS改行 \r\n なら2文字削除する)。

  chop! は元の文字列を破壊的に書き変える。

```
 "abc\n".chop
  => "abc"
 "abc\r\n".chop
  => "abc"
 "abc\n".chop.chop
  => "ab"
```

- *chomp* (*rs*)

  chomp

  chomp! (*rs*)

  chomp! - 文字列の末尾削除(条件付)

文字列末尾から調べて *rs* という文字列があればそれを取り除く。なければなにもせず、nil を返す。 *rs* を指定しない場合は改行文字が取り除かれる(厳密には変数 $/ の値)。chomp! は、元の文字列を破壊的に変更する。

```
x="abc\n"
x.chomp
 => "abc"
x
 => "abc\n" (変わらない)
x="abc"
x.chomp
 => "abc"
x="abc\n"
x.chomp!
 => "abc"
x
 => "abc" (変わる)
```

通常、ファイルや端末から1行ずつ読み込んだ文字列には末尾に改行文字(\n)がつくので、これを取り除きたいときに利用する。

- *downcase*

  downcase! - 文字列の小文字化

  upcase

  upcase! - 文字列の大文字化

  capitalize

  capitalize! - 文字列のキャピタライズ

  元の文字列のうちアルファベットをすべて 小文字化（downcase)、大文字化(upcase)、または先頭大文字化(capitalize)する。 downcase! upcase! capitalize! は、破壊的に書き変える。

```
x="Hello, World!!"
x.upcase
 => "HELLO, WORLD!!"
x.downcase
 => "hello, world!!"
x.capitalize
 => "Hello, world!!"
```

- *gsub* (*pattern*){*replace*}

  gsub! (*pattern*){*replace*} - 文字列の置き換え

  元の文字列のうち *pattern* にマッチするものを *すべて replace* に置き換え、その結果の文字列を返す。

  単純に文字列を置換することもできるし、

```
x = "This is a pen"
y = x.gsub(/is/){"IS"}
x
 => "This is a pen"
y
 => "ThIS IS a pen"
```

  パターンを指定して、マッチするものすべてを置換することもできる。

```
x = "This is a pen.  That is a pencil."
x.gsub(/is/){"IS"}  (単純な is 指定)
 => "ThIS IS a pen.  That IS a pencil."
x.gsub(/\bis\b/){"IS"}  (\bは「単語の境界」を意味する正規表現)
```

```
  => "This IS a pen.  That IS a pencil."
```

単語の末尾1字のみ大文字に変えるには以下のようにする。

```
 x = "This is a pen.  That is a pencil."
 x.gsub(/(\w*)(\w)\b/){$1+$2.upcase}
  => "ThiS iS A peN.  ThaT iS A penciL."
```

パターンの部分 *pattern* にグルーピングの（ ）を使った場合、括弧内の正規表現にマッチした部分文字列が自動的に $1, $2, $3, ...に保存される。これらは置き換え文字列 *replace* で利用でき、 $1は1個目の( )にマッチした文字列に、 $2は2個目の( )にマッチした文字列に、 $*N*は*N*個目の( )にマッチした文字列に、それぞれ置き換えられる。

上記の正規表現では最初の括弧（\w*）が最後の1字を除いた英単語で、2個目の括弧（\w）が英単語の最後の1字にマッチする。\b は単語の境界を意味するのでマッチングは 1単語ごとに行なわれる。

以下の例は、文字列中に現れるすべての `dog' または `cat' をそれぞれ、`pretty dog', `pretty cat' に置き換える。cat, dog が複数形だった場合にもマッチし（s?)、その場合は置換え後の文字列にもsを付ける。

```
 x="I saw cats scattered by a dog."
 x.gsub(/\b(dog|cat)(s?)\b/i){'pretty '+$1+$2}
  => "I saw pretty cats scattered by a pretty dog."
```

置換え後の文字列に $1, $2 を使う場合には引用符の外に出すように気を付ける。

- *sub* (*pattern*){*replace*}

  sub! (*pattern*){*replace*} - 文字列の置き換え

  上記 gsub(gsub!) はパターンにマッチするものすべてを置き換えるが、sub(sub!) は最初の一つだけを置き換える。その他の点はgsub(gsub!)と同じ。

```
 x = "This is a pen.  That is a pencil."
 x.sub(/is/){"IS"}
  => "ThIS is a pen.  That is a pencil."
```

- *split*

  split (*sep*) - 文字列の分割

  元の文字列を、指定したパターン *sep* を区切りと見なして分割し、分割された文字列を配列に格納して返す。*sep* を省略したときは空白を区切りと見なす(厳密には変数 *$;* の値で、*$;* が*nil* の場合は先頭を除いた空白となる)。

  特定の記号(カンマなど)で区切られた文字列を一気に分割して配列に格納するときに利用する。join の逆と思えば良い。たとえば、カンマで区切られた次のようなデータがあるとする。

```
    中町太郎, 90,  70,20
```

  各項目が「カンマ(,) スペース(が0個以上)」 で区切られている。これを正規表現にすると /,\s*/ となるので、以下のようにするとこの行から項目だけを抽出した配列が生成される。

```
 "中町太郎, 90,  70,20".split(/,\s*/)
  => ["中町太郎", "90", "70", "20"]
```

これまで何度か登場した成績処理プログラムをsplit を利用して書き換える場合、データの読み込み部分を以下のように変更すればよい。

正規表現で分解するバージョン:

```
point = Hash.new

while line=gets
  if /(\S+)\s+(\d+)\s+(\d+)/ =~ line
    point[$1] = [$2.to_i, $3.to_i] # 配列を代入
  end
end
```

↓

splitで分解するバージョン:

```
point = Hash.new

while line=gets
  data = line.split(/,\s*/)
  point[data[0]] = data[1..-1]
end
```

◆様々な条件での並べ換え

ソート(並べ換えること)は、大量のデータ処理には欠かせない。数値だけが並ぶ配列であればsortメソッドでソートできるが、複雑な配列やハッシュを様々な条件で並べ換えるときに有用な sort_by の利用方法に慣れておこう。

以下の説明では分かりやすくするため、ソート対象となる配列(またはハッシュ)に初期値を代入しているが、実践プログラムではプログラム起動時に入力されたデータなどを元にその都度変わる。

◇一般的なソート

たくさんの値が要素として配列に格納されている場合、それらを昇順にソートしたものを得たいならたんにsortメソッドを呼ぶだけでよい。

```
x = [9, 20, 10, 15, 8]
y = x.sort
y
 => [8, 9, 10, 15, 20]
x = ["orange", "apple", "strawberry", "pear"]
y = x.sort
y
 => ["apple", "orange", "pear", "strawberry"]
```

数値でも文字列でも昇順に並べ換えられる。降順にしたい場合は結果を reverse で反転すればよい。

```
x = [9, 20, 10, 15, 8]
y = x.sort.reverse
y
 => [20, 15, 10, 9, 8]
foo = ["orange", "apple", "strawberry", "pear"]
bar = foo.sort.reverse
bar
 => ["strawberry", "pear", "orange", "apple"]
```

◇独自規準のソート

降順順、昇順以外の複雑な並べ換えもできる。たとえば、次のようなCSVデータの並べ換えを考える。

ja-math.csv

```
氏名,国語,数学
山田太郎,50,40
中町太郎,90,70
飯森花子,91,90
鶴岡一人,60,50
酒田三吉,52,80
三川一二三,12,96
```

このCSVファイルを

```
score = CSV.read("ja-math.csv", headers:true)
```

として読み込むと概念的に以下のような構造で読み込まれる(実際には CSV::Tableの値だがハッシュの配列と同じ性質を持つ)。

```
score = [                       # 内部にハッシュを複数持つ配列
  {"氏名" => "山田太郎",        "国語"=> 50, "数学"=>40},
  {"氏名" => "中町太郎",        "国語"=> 90, "数学"=>70},
  {"氏名" => "飯森花子",        "国語"=> 91, "数学"=>90},
  {"氏名" => "鶴岡一人",        "国語"=> 60, "数学"=>50},
  {"氏名" => "酒田三吉",        "国語"=> 52, "数学"=>80},
  {"氏名" => "三川一二三",      "国語"=> 12, "数学"=>96},
]
score.length                    # 要素数は6
 => 6
# たとえば添字番号2の要素は "飯森花子" 行の値となる
```

このようなときに「6つの要素を「国語」キーの値(つまり国語の点)の高い順に並び換える」ことを考える。

普通の sort メソッドは各要素どうしをそのまま直接比較してソートするが、要素そのままではなくハッシュのキーを取り出した「値」で比べることを指定する。そのために使うのが sort_by メソッドで、以下のように用いる。

```
配列.sort_by{|変数| 変数から比較すべきものを取り出す式}
```

score 変数の例の場合では、たとえば

```
x = score[2]
```

とした場合に、x から国語の点数を取り出すには、 x["国語"].to_i とすればよいことから、

```
score.sort_by{|x| x["国語"].to_i}
```

とすることで国語の点数で昇順にした配列が得られ、さらにこれを reverse メソッドに渡して降順の結果が得られる。

```
score.sort_by{|x| x["国語"].to_i}.reverse
 =>
 [#<CSV::Row "氏名":"飯森花子" "国語":"91" "数学 ":"90">,
```

```
  #<CSV::Row "氏名":"中町太郎" "国語":"90" "数学":"70">,
  #<CSV::Row "氏名":"鶴岡一人" "国語":"60" "数学":"50">,
  #<CSV::Row "氏名":"酒田三吉" "国語":"52" "数学":"80">,
  #<CSV::Row "氏名":"山田太郎" "国語":"50" "数学":"40">,
  #<CSV::Row "氏名":"三川一二三" "国語":"12" "数学":"96">]
```

このソート結果を順序よく出力するには、each で1つずつ取り出して処理する。

```
score.sort_by{|x| x["国語"].to_i}.reverse.each do |i|
  ……出力処理……
end
```

以上をまとめたプログラムを示す。

kokugo-rank.rb

```
#!/usr/bin/env ruby
# coding: utf-8
require "./kprintf.rb"
require 'csv'

# ja-math.csv: 氏名,国語,数学
score = CSV.read("ja-math.csv", headers:true)

puts "元の並び:"
print "--氏名--------------+-国語-+-数学-+-合計--\n"
score.each do |row|
  name, ja, math = row["氏名"], row["国語"], row["数学"]
  total = ja.to_i + math.to_i
  printf("%-20s %5d  %5d  %5d\n", name, ja.to_i, math.to_i, total)
end
puts "-"*42

puts "国語上位から:"
print "--氏名--------------+-国語-+-数学-+-合計--\n"
score.sort_by{|x| x["国語"].to_i}.reverse.each do |row|
  name, ja, math = row["氏名"], row["国語"], row["数学"]
  total = math.to_i + ja.to_i
  printf("%-20s %5d  %5d  %5d\n", name, math.to_i, ja.to_i, total)
end
puts "-"*42
```

以下の出力が得られる。

```
% ./kokugo-rank.rb
元の並び:
--氏名--------------+-国語-+-数学-+-合計--
山田太郎                  50     40     90
中町太郎                  90     70    160
飯森花子                  91     90    181
鶴岡一人                  60     50    110
酒田三吉                  52     80    132
三川一二三                 12     96    108
------------------------------------------
国語上位から:
--氏名--------------+-国語-+-数学-+-合計--
飯森花子                  90     91    181
中町太郎                  70     90    160
鶴岡一人                  50     60    110
酒田三吉                  80     52    132
山田太郎                  40     50     90
三川一二三                 96     12    108
```

```
┆----------------------------------------
```

◆データの格納と検索・絞り込み

テキストや数値を複合的に含むデータを保持し、それらから利用者の希望するもののみ提示するような機能を考える。ここでは単純化したシラバスを例に考え方を示す。

◇シラバスの表現例

たとえば以下のようなシラバスに基づく科目情報を考える。

表1: 履修科目データの例

科目名	基礎プログラミング
科目コード	154
担当教員	鳥海三輔
開講時期	春学期
概要	プログラミングを通じて問題解決能力を向上を図る。本講ではRubyを用い，データ処理の構造と考え方の基礎を学ぶ。
科目名	応用プログラミング
科目コード	254
担当教員	飯森大和
開講時期	秋学期
概要	問題解決を通じてプログラミング能力向上を図る。本講ではプログラミング言語Cを通じて，問題解決に最適なアルゴリズムを適用する方策を学ぶ。
科目名	応用もっけ論
科目コード	39
担当教員	酒田育造
開講時期	春学期
概要	山形県庄内地方でよく使われる「もっけ」には様々な意味があり，他の地域にもこのような多様性を持つ言葉がある。その成立ちをふまえて正しくその地域の「もっけ」を理解し，深い交流のきっかけとする。

以下同様の項目が多数続く…

このように，形式が同じものが多数あるようなデータを設計し，検索できるデータベースを作成する過程を示す。

これ全体が「値」

キー1	値1c
キー2	値2c
キー3	値3c

これ全体が「値」

キー1	値1b
キー2	値2b
キー3	値3b

1科目分の情報は1つのハッシュで表せる。そのハッシュ全体がさらに「値」となり，それを複数集めることで科目データベースができ上がる。ハッシュで表現した1科目分の「値」をどのように集めるかについて考察する。

-->

◇CSVファイルとの連携

上記のシラバス例をCSV化する。Calc のような表計算ソフトを利用して入力する例を示す。

syllabus.csv

	A	B	C	D	E
1	科目名	科目コード	担当教員	開講時期	概要
2	基礎プログラミング	154	鳥海三輔	春学期	プログラミングを通して問題…
3	応用プログラミング	254	飯森大和	秋学期	問題解決を通してプログラミ…
4	応用もっけ論	39	酒田育造	春学期	山形県庄内地方でよく使われ…

「概要」の列は長くて編集しにくいかもしれない。LibreOffice Calc であれば「概要」の列見出し右クリックし、セルの書式設定を選ぶ。「配置」タブで「テキストを自動的に折り返す」をチェックしてから列幅を画面に収まる程度にすると多少緩和される。

RubyのCSVライブラリを用いて CSV ファイルをハッシュとしてアクセスできる形式に変換する。syllabus.csv のように1行目が見出しとして構成された CSV ファイルであれば以下の流れで読み取りを行なう。短い手順なので irb で実験してみよう。

```
irb> require 'csv'
=> true
irb> csv = CSV.read("syllabus.csv", headers: true)
=> #<CSV::Table mode:col_or_row row_count:4>
```

通常 CSV.read で得られた値は、あたかも配列のように行ごとに取り出しできる。見出し行を除く先頭行が添字0となる。

```
irb> csv[0]
=> #<CSV::Row "科目名":"基礎プログラミング" "科目コード":"154" "担当教員":"鳥海三輔" "開講時期":"春学期" "概要":"プログラミングを通じて問題解決能力を向上を図る。\n本講ではRubyを用い，データ処理の構造と考え方の基礎を学ぶ">
```

このように取り出した行は、あたかもハッシュのように特定の列の値を取り出せる。

```
irb> csv[0]["科目コード"]
=> "154"
```

また、headers オプションを true にしたときはこれに加え、 1行目にある見出し項目をハッシュのキーのように指定することで列全体を配列として取り出せる。

```
irb> csv["科目名"]
=> ["基礎プログラミング", "応用プログラミング", "応用もっけ論"]
```

この様子を示すと以下の図のようになる。

CSV全体をCSV.readで読み、変数 c に代入したときのイメージ

c["列名$_1$"] ⇓	c["列名$_2$"] ⇓	c["列名$_3$"] ⇓		
列名$_1$	列名$_2$	列名$_3$	……	
値$_{1\text{-}1}$	値$_{1\text{-}2}$	値$_{1\text{-}3}$	……	⇐ c[0]
値$_{2\text{-}1}$	値$_{2\text{-}2}$	値$_{2\text{-}3}$	……	⇐ c[1]
⋮	⋮	⋮	……	
値$_{n\text{-}1}$	値$_{n\text{-}2}$	値$_{n\text{-}3}$	……	⇐ c[n]

たとえば、列で縦方向に取り出す場合、c["列名2"] とすると2列目の値が配列となり、

[値$_{1\text{-}2}$, 値$_{2\text{-}2}$, 値$_{3\text{-}2}$, ..., 値$_{1\text{-}2}$]

が得られる。

◇CSVファイルの読み取り・検索プログラム

以上のしくみをふまえて、ヘッダつきで書かれた CSV ファイルを読み取るプログラムを示す。

csv-read.rb

```
#!/usr/bin/env ruby
# coding: utf-8
require 'csv'                    # CSVライブラリ
csvfile = "syllabus.csv"         # シラバス定義CSVファイル

csv = CSV.read(csvfile, headers: true)

# 動作確認用に1行ずつ表示してみる
csv.each {|row|
  # ppメソッドはpメソッドよりさらに見やすく値表示する
  pp row.to_h                    # to_h メソッドはハッシュに変換してくれる
}
```

syllabus.csvを保存してから 上記プログラムを実行してみよ。

▶ csv-read.rbの実行例: Open/クリックして開く

この3つのレコードのうち，「開講時期」が "春学期" であるものを選択するには以下のようにする。

```
# csv-read.rbの続きから
db = csv                          # いったん別の変数にコピー
db = db.select {|row| row["開講時期"] == "春学期"}
```

ブロック変数 row は、3度代入され繰り返される。 row["開講時期"] の値は回ごとに、 "春学期", "秋学期", "春学期" が得られるため、== "春学期" で比較して true が返るのは1行目と3行目である。

なお、selectメソッドによる絞り込みでは，文字列比較だけでなく正規表現マッチや数値比較も使える。これらの絞り込みを繰り返すことでどのような検索も可能で、複合条件の指定もできる。以下の例は，開講時期と担当教員での絞り込みを正規表現マッチで行ない，該当するもののみを出力するプログラムである。

csv-search.rb

◇数値項目の処理

CSV.read によるCSVファイル一括読込の標準動作では、各項目を文字列として代入する。したがって、CSV データの項目中に数値と見なすべきデータがあった場合は、数値に変換する処理が必要となる。これには以下の2つの方法が考えられる。

- 要素を取り出してから数値に変換する(to_i, to_f)
- CSV読み取り時に数値変換

■要素を数値化して比較

表1の履修科目表を CSV.read で読み取ったものから、「科目コード」が100以上200未満のものを選別してみる。このとき「科目コード」はいずれも必ず整数値が入るものと仮定できるとする。

上記 csv-search.rb の科目検索部分の select による選択を以下のようにすればよい。

```
db = db.select {|row|   # dbの集合から1行ずつ row 変数に取り出して選別
  row["科目コード"].to_i >= 100 && row["科目コード"].to_i < 200
  # 科目コード(数値) >= 100     かつ   科目コード(数値) < 200
}
```

■CSV読み取り時に数値変換

以下のように CSV.read に与えるオプションで converters: :numeric を与えると数値と解釈できる値は数値化される。

```
x = CSV.read(csvfile, headers: true, converters: :numeric)
```

これにより、上記科目コード選択は以下のように書けることになる。

```
db = db.select {|row|
```

```
  row["科目コード"] >= 100 && row["科目コード"] < 200
}
```

もちろん、元のCSVデータに数値化できない値があった場合は数値化されず、数値に対してのみ行なえる演算を施すとエラーになるので注意が必要である。数値でない項目がある可能性を考える場合は、converters: オプションを用いず、その都度 to_i や to_f を使うのが安全といえる。

◇練習問題

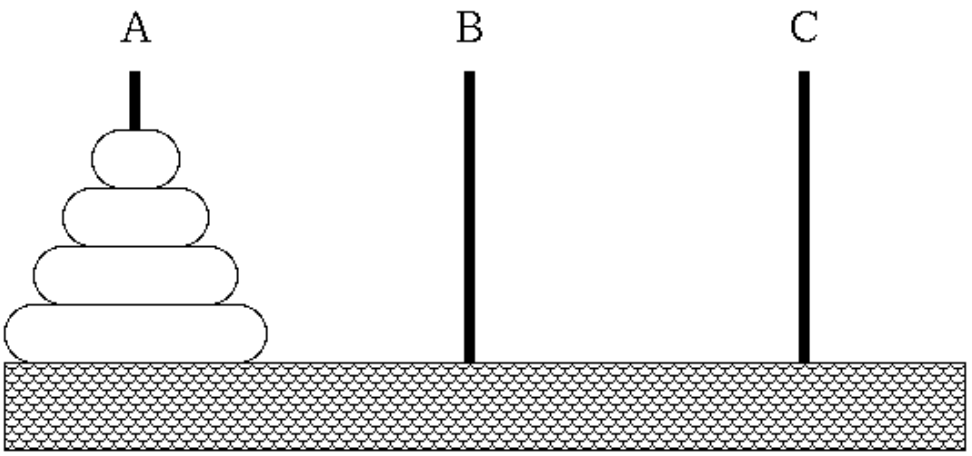
1. 複数の値が入った配列 a を受け取り、各要素を整数に直したものの平均値を求めるメソッド mean() を定義せよ。
2. ja-math.csvを CSV.read で読み取り、国語の得点を配列で得て puts で出力するプログラム ja-list.rb を作成し、実行例を示せ。
3. 1 , 2を組み合わせ、ja-math.csv から国語の平均点を求めるプログラム ja-mean.rb を作成し実行例を示せ。
4. 上で求めた平均点を用い、平均点以上を取った生徒の一覧を出力するプログラム ja-good.rb を作成し実行例を示せ。

第12講 再帰

◆ハノイの塔

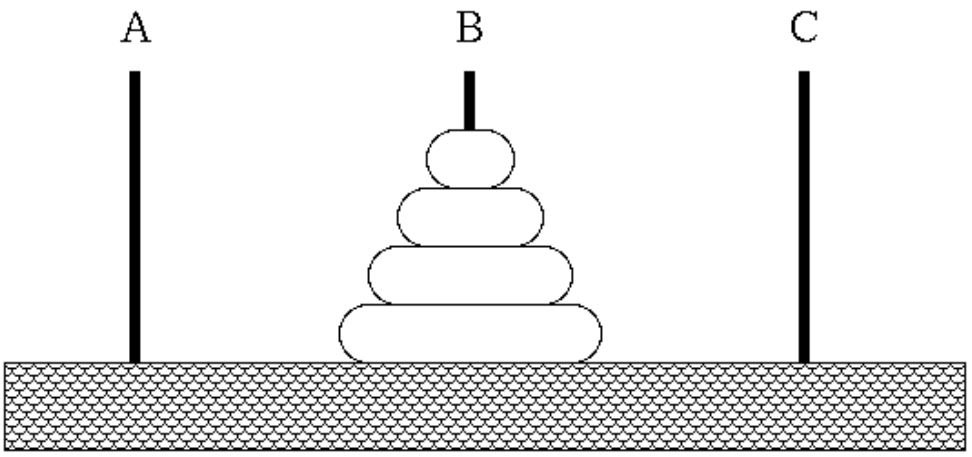
*ハノイの塔*というパズルゲームがある。

ハノイの塔(移動前)



A, B, C 3本の棒があり、そのうちのAに円盤が何枚か挿してある。この円盤すべてを別の棒に移動する。

ハノイの塔(移動完了後)



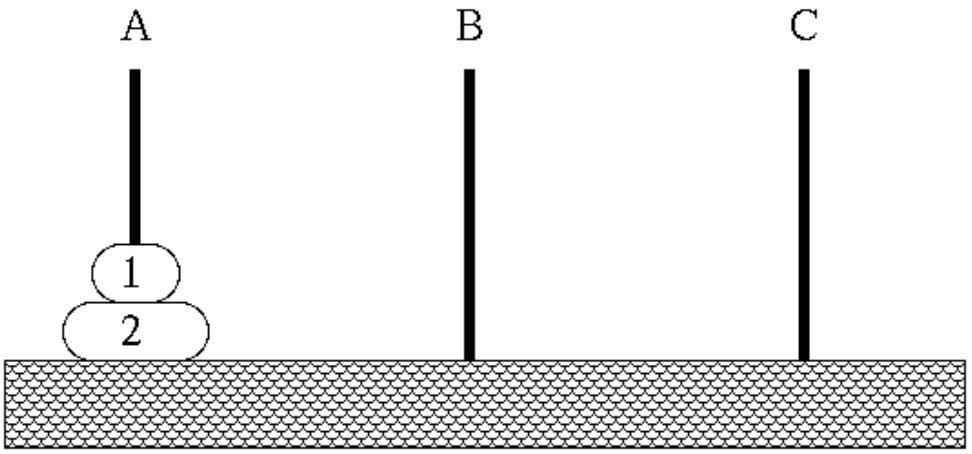
ただし、円盤すべてをまるごと動かしてよいわけではなく、以下の規則に従わなければならない。

1. 一度に動かしてよいのは1枚だけ
2. ひとつの円盤の上にはそれより大きな円盤を置いてはいけない

このパズルを解くRubyプログラムを作ってみよう。

プログラムを作りやすいように円盤に番号を付けて管理する。

ハノイの塔(番号付き)



Rubyプログラムを起動すると、次のように教えてくれるものとする。

```
1番の円盤 を A から C に移動しなさい
2番の円盤 を A から B に移動しなさい
1番の円盤 を C から B に移動しなさい
```

この例は分かりやすくするために、2枚だけで示しているが、実際に作るプログラムは板が何枚でも手順を教えてくれるものである。このようなプログラム hanoi.rb を作ってみよう。

> 問題: ハノイの塔を解き、その手順を教えてくれる Ruby プログラムはだいたい何行くらいになると思うか、直感で答えよ。

◆再帰

◇再帰とは

何かの中味に、それ自身が現れることを*再帰*という。何かの定義に、それ自身が登場するようなものを*再帰的定義* という。たとえば年齢に関する以下の二つの定義を比べてみよう。

1. 年齢とは

   生まれた年月日から経過した年数のことである。

   ただし、1年に満たない月と日は切り捨てる。

2. 年齢とは

   1年前の年齢に1を足したものである。

   ただし、1年前に生まれていない時の年齢は0歳である。

一般的な感覚でいえば、1.の定義が妥当だろう。しかし、現実にわれわれが生活のなかで自分の年齢を数えているのは2.であることが多い。たとえば、誕生日が来たときいちいち自分の生まれた年を引き算して毎年自分の年齢を計算する人はまれだろう。昨日まで意識していた年齢に1を足してこれから1年間の年齢とするだろう。その方が簡単だからである。

ここで出した例は、年齢の計算という非常に単純なものなので、1と2どちらでやろうとも大差ないの

だが、もっと複雑な問題のなかには定式どおり解くよりも再帰的な方式に則って解くと簡単になることが多い。

ここで気を付けるべきことがある。2.の定義は「年齢」の定義のなかにさらに「年齢」という語があるので、それではいつまで経っても厳密に語の意味が定まらないように感ずるが、そうではない。ただし書きがあるので、最終的に「1年前に生まれていない時の年齢は0歳」という定義文により曖昧さなく年齢が確定する。

◇再帰を利用したプログラミング

簡単な例をもうひとつ。プログラミングにしやすい例題を考えてみよう。自然数に対する*階乗*の(再帰的でない)定義は以下のようになる。

1. *N* の階乗とは

   1から *N* までのすべての自然数を掛けたもの

これを再帰的定義に直すと以下のようになる。

2. *N* の階乗とは

   N-1 の階乗と *N* を掛けたものである。

   ただし、1の階乗は1である。

それぞれの定義をRubyのメソッドにすると以下のようになるだろう。

fact1.rb

```
# Nの階乗を求めるメソッド(定義1)

def factorial(n)
  ans = 1
  i = 1
  while i <= n
    ans *= i
    i += 1
  end
  ans
end
```

fact2.rb

```
# Nの階乗を求めるメソッド(定義2)

def factorial(n)          # nのfactorial(階乗)とは
  if n <= 1               # nが1なら
    1                     # 答は1
  else                    # そうでなければ
    n * factorial(n-1)    # n と factorial(n-1)を掛けたもの
  end
end
```

定義1によるメソッドの動きに関しては、これまで学習したので説明を省略し、定義2によるメソッドについて考える。定義2のプログラム中下線を施した部分が *再帰呼び出し*となっている。

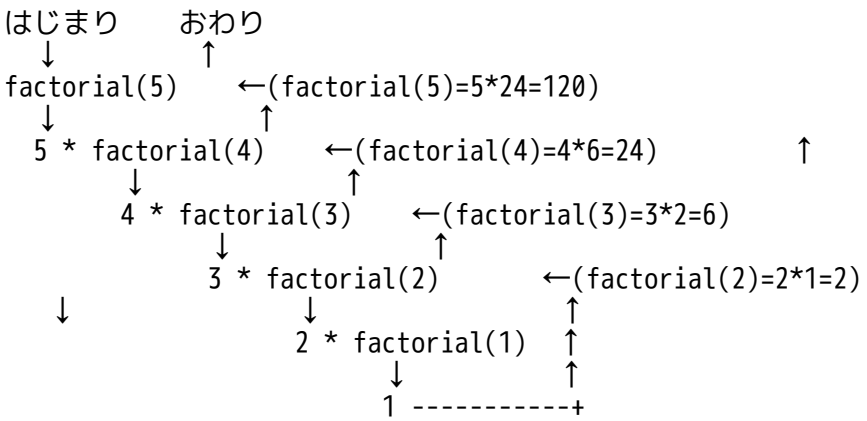
たとえば、定義2のときに、factorial(5) と呼び出した場合を考えよう。

```
def factorial(n)        # ←n=5
  if n <= 1             # 5 > 1 なので
    1                   # ここには来ない
  else                  # こっち↓
    n * factorial(n-1)  # 5 と factorial(5-1)を掛けたもの
  end
end
```

n=5なので、1よりも大きくelse節内が評価される。すると、5 * factorial(5-1) を計算することになる。この時点ではfactorial(5-1)の値はまだ分からない。 factorial(4) が呼ばれる。

```
def factorial(n)        # ←n=4
  if n <= 1             # 4 > 1 なので
    1                   # ここには来ない
  else                  # こっち↓
    n * factorial(n-1)  # 4 と factorial(4-1)を掛けたもの
  end
end
```

n=4なので、1よりも大きくelse節内が評価される。すると、4 * factorial(4-1) を計算することになる。ここでまた、factorial(3) が呼ばれる……。ということを繰り返すうちに、nがひとつずつ減っていき、最終的にfactorial(1)が呼ばれ、factorial メソッドは1を返す。これを図式化すると以下のようになる。



実際にコマンドラインから利用できるプログラムに直して実行してみよう。

factorial.rb

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

def factorial(n)
  if n <= 1
    1
  else
    n * factorial(n-1)
  end
end

if ARGV[0] == nil
  STDERR.puts "階乗を計算したい数を指定して下さい"
  STDERR.puts "例: ./factorial.rb 5"
  exit 0
```

```
end

n=ARGV[0].to_i
printf("%dの階乗は%dです\n", n, factorial(n))
```

◇再帰的メソッドのポイント

メソッドを再帰的に定義する場合、以下の点に注意する。

- 問題を分割して考えたときに、分割した部分を解く考え方と全体を解く考え方が同じである場合、再帰を使うことができる
- 問題分割を繰り返し、これ以上問題が分割できない場合に結果そのものを返す条件ブロックを入れる(*脱出条件*)
- 分割した問題を再帰的に同じメソッドを呼んで得られた結果を合成したものを最後の結果として返す

階乗計算は再帰を用いず、ループを利用しても十分に簡単であるので、本来はループ処理で定義するのが自然である。再帰処理は実際には複雑な問題を簡単にして解く場合に有効となる。次節ではそのような例を見ていこう。

◆再帰処理

再帰的な方法を用いてプログラムを組み立ててみよう。

◇再帰処理による並べ換え

Rubyでは、配列にsortメソッドが備わっているので、配列に入っているデータを並べ換えてもらうことができる。最近の多くの言語では、言語処理系の方に標準で並べ換え用のメソッドが用意されているのでたくさんのデータを並べ換える具体的な手順を知らなくてもよくなっている。

しかしここでは再帰処理を理解する上での題材として、並べ換え手法のひとつであるクイックソートを自分の手で作ってみよう。

◇クイックソートとは

クイックソートはたくさんあるデータ要素を次のような手順で並べ換える方法である。

【クイックソートの方法】

大きな箱 *X*に任意個のデータが入っているものとする。クイックソートは以下の手順で行なう。

1. *X* の中のデータが1個以下なら 並べ換え完了、そうでなければ次に進む
2. *X* の中から、任意のデータを1個抜き取り、その値を *k* とする。
3. *k*を抜き取ったあとの中のデータを順次見ていき、それが

    - *k* より小さかったら左側に置く
    - そうでなければ右側に置く

  ことをデータがなくなるまで繰り返す
4. 左側に寄せたものをクイックソートする
5. 右側に寄せたものをクイックソートする
6. 左右のかたまりの中間に *k* を置いて 並べ換え完了

◇クイックソートしてみよう

クイックソートをカードを並べ換える方法に適用して体験して理解しよう。トランプを8枚用意しよう。といっても、すぐには用意できないので使用済みの紙切れを8枚に切って、それぞれに好きな数字を書こう。

数字を書いた8枚のカード(もっと多くてもよい)が準備できたら、よくシャッフルする。そうしたら以下の手順で並べ換えよう。

「クイックソートによるカードの並べ換え」

1. 並べ換えるべき枚数が1枚以下なら終了、それ以上なら次に進む
2. まず1枚のカードを机の真中に置く
3. 手に残ったカードから1枚ずつ引いていき
    - 真中に置いたカードの数字より小さかったら左側に置く
    - そうでなければ右側に置く

  ことをカードがなくなるまで繰り返す(このとき左右に置くカードはでたらめに並べるか重ねておいてよい)
4. 左側に置いた山を左側の机スペースでクイックソートする
5. 右側に置いた山を右側の机スペースでクイックソートする
6. おしまい

> 最初に選ぶカードに何を選んでもクイックソートが成功することを確認しよう

◇クイックソートを行なうメソッド

クイックソートを行なうメソッド `qsort` を定義しよう。

次のような設計とする。

- `qsort` メソッドは引数として配列を受け取る
- 左側の箱、右側の箱としてそれぞれ、配列`left`、配列`right`を利用する
- `left, right` に対して再帰的に `qsort` する

- 最後に left, 真中の値, right を並べて完了

これを実際にメソッドにしてみよう。いきなりメソッドにするのは難しいので、ことばで骨子を考える。

```
def qsort(data)          # dataは数値の入った配列
  if データが1個以下なら
    dataそのものを返して並べ換え完了
  else
    left(左側配列)を新規作成
    right(右側配列)を新規作成
    変数dataからデータを1個取り出す
    これを k とする

    残ったdata全てに対して以下を繰り返す
      if k より小さかったら
         leftに追加
      else
         rightに追加
      end
    繰り返し終わり

     「leftをqsortしたもの」と k と「rightをqsortしたもの」
    をその順に結合したものを結果として並べ換え完了
  end
end
```

上記の「ことば」で定義した qsort メソッドを Rubyプログラムに直そう。

```
def qsort(data)         # 配列を受け取る

  if data.length <= 1   # データが1個以下なら
    data                # そのものを返して並べ換え完了
  else
    left = Array.new    # left(左側配列)を新規作成
    right = Array.new   # right(右側配列)を新規作成
    k = data.shift      # 変数dataの先頭を取り出してkにする

    for x in data       # 残ったdata全てに対して繰り返す
      if x < k          # kより小さかったら
         left << x      # 左側に追加
      else
         right << x     # 右側に追加
      end
    end                 # 繰り返し終わり

    # qsort(left)の末尾にkを足し、さらにqsort(right)を結合
    qsort(left).push(k) + qsort(right)
  end
end
```

この qsort メソッドを使って、並べ換えできるか確認しよう。そのために、配列を読み込んだり、画面に表示したりするメソッドを作っておこう。

- 【配列を読み込むメソッド】

  標準入力から順次数値を読み取り配列にするメソッド

```
def readNumArray()
  ary = []
  while line=gets               # 入力が続く間
```

```
    datum = line.chomp!.to_i    # 整数に直して
    ary.push(datum)             # aryに追加していく
  end
  ary                           # 最後にaryを呼び主に返す
end
```

- 【配列を画面表示するメソッド】

  配列を受け取り、各要素をカンマで区切って表示する。これには join が利用できる。

```
def printArray(a)
  printf("[%s]\n", a.join(", "))
end
```

これらのメソッドを組み合わせて、プログラムは完成する。

qsort.rb

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

def qsort(data)         # 配列を受け取る

  if data.length <= 1   # データが1個以下なら
    data                # そのものを返して並べ換え完了
  else
    left = Array.new    # left(左側配列)を新規作成
    right = Array.new   # right(右側配列)を新規作成
    k = data.shift      # 変数dataの先頭を取り出してkにする

    for x in data       # 残ったdata全てに対して繰り返す
      if x < k          # kより小さかったら
         left << x      # 左側に追加
      else
         right << x     # 右側に追加
      end
    end                 # 繰り返し終わり

    # qsort(left)の末尾にkを足し、さらにqsort(right)を結合
    qsort(left).push(k) + qsort(right)
  end
end

def readNumArray()
  ary = []
  while line=gets               # 入力が続く間
    datum = line.chomp!.to_i    # 整数に直して
    ary.push(datum)             # aryに追加していく
  end
  ary                           # 最後にaryを呼び主に返す
end

def printArray(a)
  printf("[%s]\n", a.join(", "))
end

ary = readNumArray()

puts "整列前:"
printArray(ary)
puts "整列後:"
printArray(qsort(ary))
```

上記のプログラム、qsort.rb を保存して実行し、1行に1つずつデータを入れて小さい順に並べ換えでき

ることを確認せよ。

実行例:

```
% ./qsort.rb
 45
58
5
9
45
20
 [C-d][Return]
整列前:
[45, 58, 5, 9, 45, 20]
整列後:
[5, 9, 20, 45, 45, 58]
```

◆ハノイの塔を解くメソッド

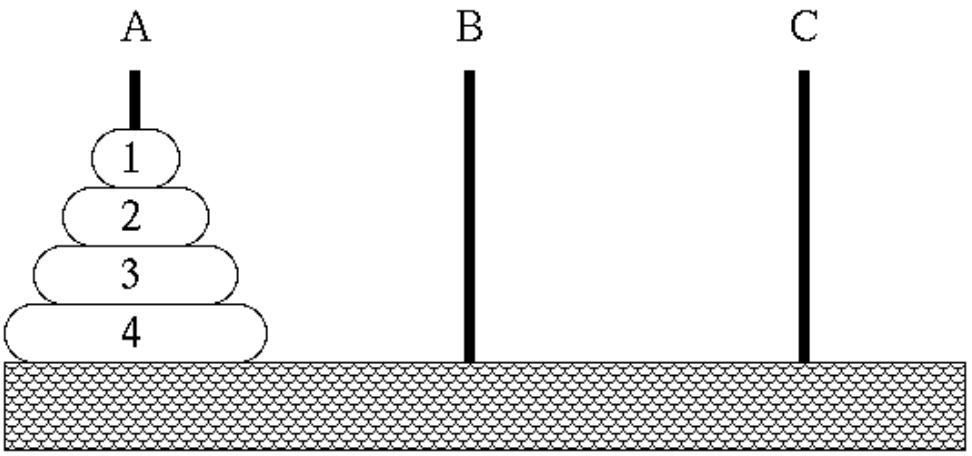
ハノイの塔の問題に戻ろう。

一見難しそうなハノイの塔も、再帰的手法を用いて問題を分割すると簡単になる。

◇ハノイの塔の問題分割

まず初期状態を見る。

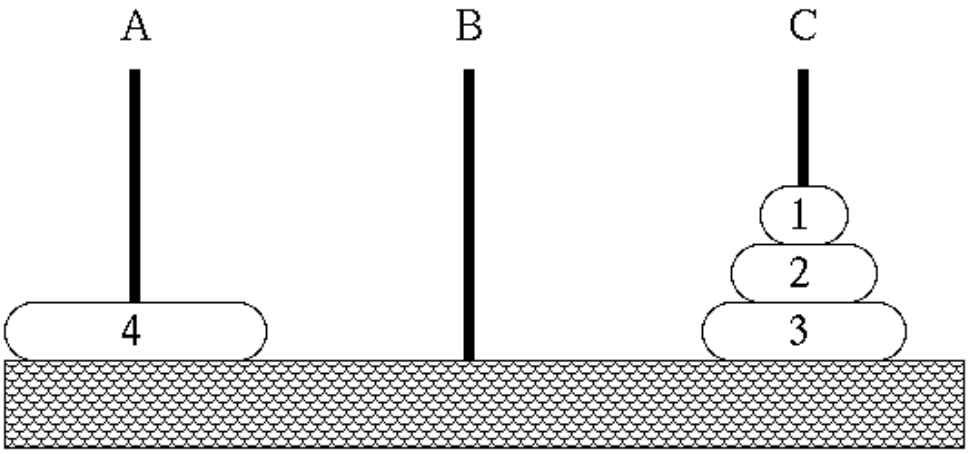
ハノイの塔(移動前)



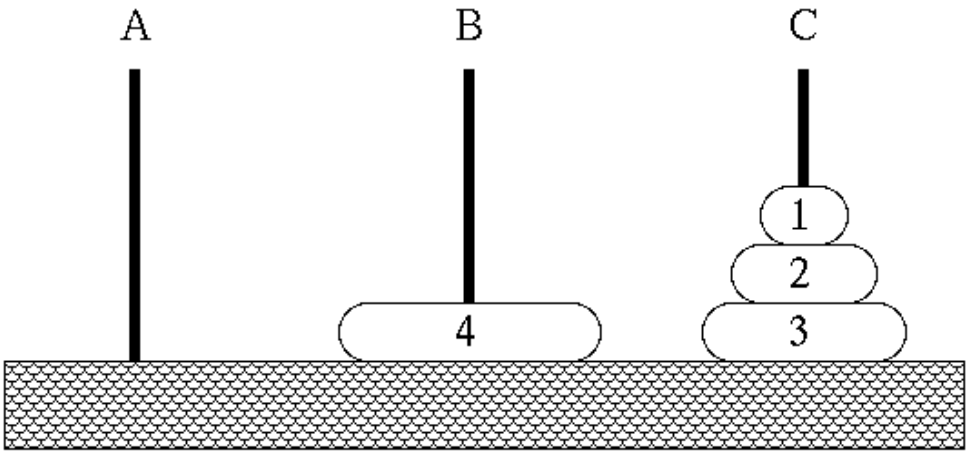
Aに積み重なっている4つの円盤群を、規則に従い1枚ずつBに移したい。このときに、

> 「もし私の知らないところで誰かが1から3をAからCに移動してくれれば楽だなあ。そしたら私は4番をBに移して、最後にその人に1から3をBに戻してもらえばおしまいだ」

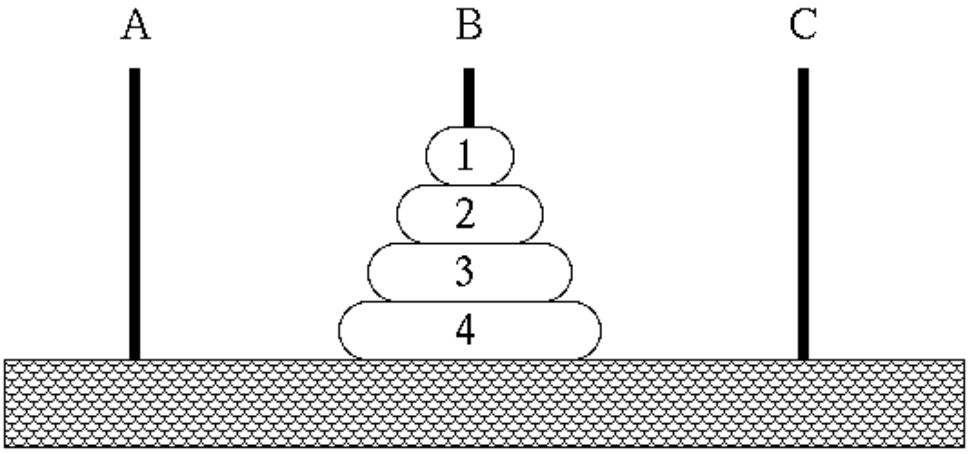
という風に考える。つまり、ハノイの塔に慣れた人を連れて来て、 1から3をCに移動してもらう(下図)。



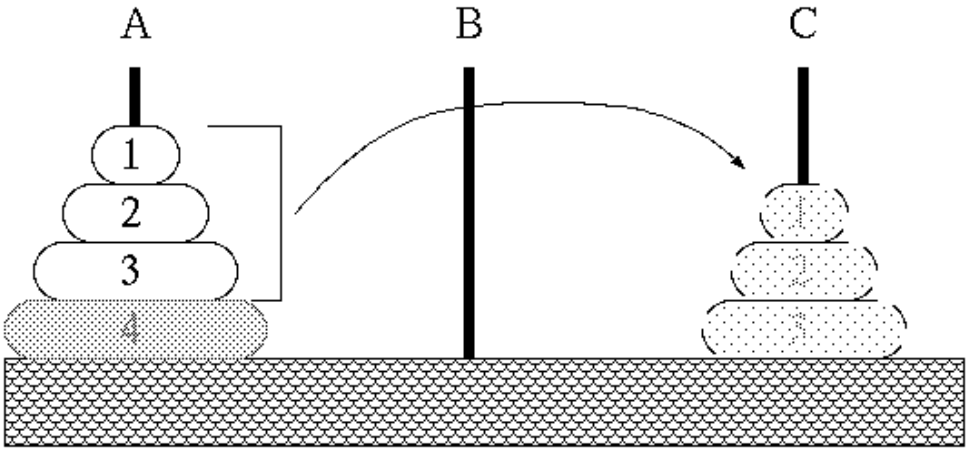
次に自分で、4をBに移動する(下図)。



そして、もう一度達人を連れて来て1から3をBに移動してもらう。



こんなムシのいい考え方でよいのだろうか。よいのである。連れて来られた達人は次のように考える。



「1から3をAからCに動かせばいいのか。うーむ。寝てる間に誰かが1から2をAからBに移動してくれれば楽だなあ。そしたら私は3番をCに移して、最後にその人に1から2をCに戻してもらえばおしまいだ」

このようにして、

- 「4枚移動しなさいといわれたときに3枚だけ誰かにやってもらう」
- 「3枚移動しなさいといわれたときに2枚だけ誰かにやってもらう」

を繰り返していくと、最終的に

- 「2枚移動しなさいといわれたときに1枚だけ誰かにやってもらう」
- 「1枚移動しなさいと誰かにいわれる」

となり、最後に頼まれた人は1枚だけ移動すればよくなり、何も考えなくても移動できることになる。

◇Rubyプログラムへ

上記の考え方をRubyメソッドにしよう。

- 積み重なった枚数
- それらが元々ある棒の名前
- 移動先の棒の名前
- 一時的に利用してよい棒の名前

を受け取り、移動手順を解くメソッド hanoi を考える。重要な部分をことばで書くと次のようになる。

```
# n    = 枚数
# from = 元の棒の名前
# to   = ゴールの棒の名前
# work = 一時的に利用してよい棒の名前
def hanoi(n, from, to, work)
  if 1枚か?
     「n番の円盤を from から to に移動しなさい」
```

```
    と表示する(かんたん)
  else                    # 2枚以上の場合
    まず n-1 だけを誰かに頼んで、一時待避棒に動かしてもらう
    そして「n番の円盤を from から to に移動しなさい」と表示
    最後に n-1 を誰かに頼んで、一時待避棒から目的棒に動かしてもらう
  end
end
```

これを実際にRubyプログラムに直すと完成する。

練習問題 14

1. 生まれた西暦を入力させ、年齢の再帰的定義に従い現在の年齢を計算し、出力するプログラムage.rbを作成せよ。当日が誕生日を過ぎているかどうかは考慮しなくてよい。なお、現在の西暦年はTime.now.yearで得ることができる。
2. ハノイの塔を解くプログラムを日本語化したものを基にして、実際に動く Rubyプログラム hanoi.rb を作成せよ。
3. Dir.entries(*パス名*)とすると *パス名* で指定したディレクトリに含まれるファイル全てのファイル名を要素に持つ配列が返される。これを利用してカレントディレクトリ以下全てのディレクトリからaで始まるファイル全てを探して出力するプログラムafind.rb を作成せよ。作成に当たっては以下のことを参考にせよ。
    - Dir.entries(*パス名*) で得られるファイル名の配列の先頭の2つは "." , ".." であり、これらは無視してよい。
    - Dir.entries(*パス名*)から得られた要素がディレクトリかどうかは
      `test(?d, パス名+"/"+要素)`
      で調べられる(ディレクトリならtrueが返る)。
    - 指定したディレクトリのファイル一覧を出力するメソッドをdef afind(path) と定義したとする。そのディレクトリ内にディレクトリが見つかったら、そのディレクトリに対し再帰的にafindメソッドを呼ぶ。

第13講 オブジェクト指向とクラス定義

◆変数のスコープ

◇例題設定

次のような簡単なお財布プログラムを考える。

1. 最初の所持金は1000円
2. ユーザからの入力を繰り返し、その数を所持金に足し続ける(マイナスなら減る)。

最も簡単に作ると以下のようになる。

inout.rb

```
#!/usr/bin/env ruby
fund = 1000
while true
  printf("残高%d円です。何円出しますか(負の数は入金, 0で終了): ", fund)
  inout = gets.to_i
  break if inout == 0           # ifを文の後ろにつけると1行で書ける
  fund -= inout
end
```

実行例は以下のとおり。

```
% ./inout.rb
残高1000円です。何円出しますか(負の数は入金, 0で終了): 30
残高970円です。何円出しますか(負の数は入金, 0で終了): 50
残高920円です。何円出しますか(負の数は入金, 0で終了): -100
残高1020円です。何円出しますか(負の数は入金, 0で終了): [C-d]
```

入出金は -= の1語で書けるが仮にこれが複雑な処理だと仮定して、メソッドを使うように以下のように書き改めてみた。

inout-bad.rb

```
#!/usr/bin/env ruby
fund = 1000

def purse(withdraw)     # 引き出し額を受け取る
  fund -= withdraw      # その分だけfundから引く……つもりだが
end                     # メソッドを囲む壁があり変数名などはこれを越えられない

while true
  printf("残高%d円です。何円出しますか(負の数は入金, 0で終了): ", fund)
  inout = gets.to_i
  break if inout == 0   # (ifを文の後ろにつけると1行で書ける)
  purse(inout)
end
```

動かすとエラーになる。

```
% ./inout-bad.rb
```

```
残高1000円です。何円出しますか(負の数は入金, 0で終了): 40
./inout-bad.rb:5:in `purse': undefined method `-' for nil
(NoMethodError)

  fund -= withdraw
       ^
        from ./inout-bad.rb:12:in `
'
```

これは変数fundがメソッドの中では隠蔽されて見えないことが原因である。メソッドには、それを囲む壁のようなものがあり、その中と外では変数やメソッドの名前は別空間で管理され、共有できない。

したがって、*purseメソッドの外で代入したfund変数は、メソッド内部では使えない。*

一般的にほとんどのプログラミング言語の変数には*スコープ* が存在する。スコープとはある場所で設定した変数が有効な範囲のことであり、 Rubyなど多くの言語ではメソッド(関数)の中と外では世界が違うものとして処理される。

◇改良案

def purse....end 内部では、外にある fund 変数にアクセスできない。メソッドは、一つの完結した処理を定義するものであり、その部分だけで明確に動きが分かることが重要である。

メソッドを用いて正しく処理するための解決策は2つ。

1. 重要な変数の値をメソッドの引数と返却値でやり取りする
2. クラス定義を用いて変数と手続きをカプセル化する

2のオブジェクト指向的アプローチは次項で説明する。

1の方法を以下に示す。

inout-byarg.rb

```
#!/usr/bin/env ruby
fund = 1000

def purse(total, withdraw)      # 今の総額と引き出し額を受け取る
  total - withdraw              # その分だけ総額から引く
end

while true
  printf("残高%d円です。何円出しますか(負の数は入金, 0で終了): ", fund)
  inout = gets.to_i
  break if inout == 0           # ifを文の後ろにつけると1行で書ける
  fund = purse(fund, inout)     # 総額と支出をpurseに渡し、結果を受け取る
end
```

この書き方で、purse メソッドの独立性は高くなる。たとえばメソッドの外側でfund変数の名前を変えたとしても purseメソッド自体は一切書き換えなくてもよい。ただ、この例のように何らかの状態(この場合残高)を保持しつつ、入れたり出したりと行った様々な付随処理をまとめて面倒を見るかたまり(装置）を考えるとより効率的である。

そのための考え方の一つ、オブジェクト指向を次の節で説明する。

◇練習問題

次のプログラムは変数のスコープが外れていてうまく動かない。 *変数を引数で渡す形にする* ことで、これを修正せよ。

varerr.rb

◆オブジェクト指向

◇お財布プログラムの例題

引き続きお財布プログラムを考える。

財布にまつわるお金のやり取りは以下のようにまとめられる。最初は「入れるのはお金だけ」で考える。

- 財布に入れるものはお金である。
- 財布は使い始めてから使うのをやめるまでの命である。
- 使い始めるときは最初にいくらか入れた状態で始まる。
- 財布に関する操作は「お金を入れる」と「出す」である。
- 入っている額を超えない範囲で出すことができる。

オブジェクト指向では、このような性質を持ったものを*クラス*というくくりで考え、そのクラスには以下のような値や手続きがあると整理して考える。

- 「お財布」を表すクラス名を Purse とする。
- そこに属するのは「中に入る金額」と「入れる操作」と「出す操作」である。
- それらを順に変数:fund、メソッド:add、メソッド:sub とする。
- 最初にお財布を導入するときの初期金額を設定できる。

これをRubyのコードに直すと以下のようになる。

purse.rb

```
#!/usr/bin/env ruby
class Purse
  def initialize(start=0)       # コンストラクタの定義
    # initializeという名前でメソッド定義しておくと、初期化されるときに
    # 自動的に呼ばれる(それゆえ通常は定義する)。
    # 生成時に呼ばれるメソッドのことを「コンストラクタ」という。
    # @で始まる変数はインスタンス変数といいクラス内ですべてのメソッドから
    # 参照できるもの。@がない変数はメソッド間を越えられない。
    @fund = start        # 引数を指定しなかったら0になる
```

```
  end
  def add(amount)
    if amount < 0
      STDERR.printf("マイナスはだめよ(%d)\n", amount)
    else
      @fund += amount
      printf("%d円入れました。\n", amount)
    end
  end
  def sub(amount)
    if amount > @fund
      STDERR.printf("そんなに入っていません(残高: %d)\n", @fund)
    else
      @fund -= amount
      printf("%d円出しました。\n", amount)
    end
  end
  def howmuch()         # 現在の残高を返す
    @fund
  end
end
# ここまでがクラス定義本体
```

クラス定義も定義しただけでは何も起きず、利用しなければ意味がないのはメソッド定義と同じである。上の purse.rb を利用するプログラムを示す。

purse-use.rb

```
#!/usr/bin/env ruby
require_relative "purse.rb"             # 同じディレクトリあるときは require_relative

puts "=== 今日からこの財布を使おう。5000円入れておこう: Purse.new(5000)"
saifu = Purse.new(5000)                 # .newで定義したクラスの実体を生成する。
# 生成するときに initialize メソッドが呼ばれる。

puts "=== 1598円の買い物をしよう。"
puts "=== 2000円出すぞ: saifu.sub(2000)"
saifu.sub(2000)
puts "=== おつりを402円もらった。しまおう: saifu.add(402)"
saifu.add(402)
puts "=== いまいくらあるんだろう: saifu.howmuch"
printf("残り%d円です\n", saifu.howmuch)
```

実行例を示す。

```
% ./purse-use.rb
=== 今日からこの財布を使おう。5000円入れておこう: Purse.new(5000)
=== 1598円の買い物をしよう。
=== 2000円出すぞ: saifu.sub(2000)
2000円出しました。
=== おつりを402円もらった。しまおう: saifu.add(402)
402円入れました。
=== いまいくらあるんだろう: saifu.howmuch
残り3402円です
```

◇クラスとインスタンス

ここで示した財布は「お金だけ出し入れできる」財布で、そのようなものはいくらでも作れる。ある性質の集合体を設計したものを *クラス*といい、製品の仕様書に相当する。

クラスは単なる仕様書だが、そこから実際に動くものとして生み出されたもののことを*インスタンス*（実体）といい、実際に作られた製品に相当する。一つの仕様書から需要に応じてたくさんの製品が作られるように、一つのクラスからいくらでもインスタンスを作ることができる。プログラムで実際に動くものとして利用するのはインスタンスである。インスタンスをプログラム中で保持する値のことを*オブジェクト* という。

上のプログラム purse-base.rb の後半の意味を示す。

- `saifu = Purse.new(5000)`
  定義したPurseクラスの性質を持つインスタンスを一つ生成する(.new)。このとき引数として5000が渡され、それがクラス定義にある initializeメソッドに渡される。残高変数 @fund に5000が格納された状態でこの財布の利用が始まる。
- `saifu.sub(2000)`
  saifu変数にはPurseクラスから生まれたオブジェクトが入っている。このオブジェクト内にあるsubメソッドを呼ぶ。
- `saifu.add(402)`
  同様にオブジェクト内にあるaddメソッドを呼ぶ。

◇複数のインスタンス(オブジェクト)

売り物の財布なら、同じ財布を持っている人は世の中にたくさんいる。同じものではあるが、中に入っているものはそれぞれ違う。人間も、全人類同じゲノムから生まれるが、一人として同じ人は存在しない。一つのクラスから生まれるインスタンスはすべて違う個体で、独立した情報を持ち、それに従った動きをする。

```
chiharu_saifu = Purse.new(5000)         # 千春さんが財布を新しくして5000円から開始
hibiki_saifu  = Purse.new(8000)         # 響さんが財布を新しくして8000円から開始
yu_saifu      = Purse.new(1000)         # 優さんが財布を新しくして1000円から開始
```

いずれも、全く同じ設計の財布だが、別個体として存在し続ける。

chiharu_saifu
@fund変数　*5000*（残高）
addメソッド　入れる処理
subメソッド　出す処理
howmuchメソッド　残高を返す処理

hibiki_saifu
@fund変数　*8000*（残高）
addメソッド　入れる処理
subメソッド　出す処理
howmuchメソッド　残高を返す処理

yu_saifu
@fund変数　　*1000*（残高）
addメソッド　　入れる処理
subメソッド　　出す処理
howmuchメソッド　残高を返す処理

これまで使った配列(Array)やハッシュ(Hash)も同様で、 Ruby本体で定義されたクラスを我々は色々な変数に代入して利用している。

◇カプセル化

クラス定義(classからendまで)の内側で使用された変数はその外側からは一切見えない。これを*変数の隠蔽* といい、プログラム同士が同じ変数名やメソッド名(まとめてシンボルという)で競合して異常動作をする可能性をなくせる。シンボルを隠蔽してプログラムの一定の部分を独立性を高めることを*カプセル化*といい、複数人、あるいは将来別のプログラムから利用することを見越したプログラムを作るときには欠かせない。

◇継承

既に定義されたクラスの基本機能は踏襲しつつ、機能や性質を追加したい場合に*継承* を使う。

たとえば、Purseクラスは現金を入れることしか想定していない。この機能を保持したまま、「カードも入れられるようにしたい」とする場合、同じようにクラス設計すると効率が悪い。

purse-with-card.rb

```
require_relative "purse.rb"             # Purseクラスの定義を読み込む

class PurseWithCard < Purse             # Purseを親として引き継いでクラス定義
  def initialize(money=0, card=[])      # コンストラクタ(.newで呼ばれる)
    super(monkey)                       # 親(Purse)の同じ名前のメソッド(initialize)を呼ぶ
    @card = card                        # 独自のインスタンス変数を導入
  end
  def putinCard(card)
    @card << card
    printf("%s をしまいました。\n", card)
  end
  def drawCard(card)
    @card.delete(card)
    printf("%s を取り出しました。\n", card)
  end
  def myCards()
    @card.join(",")
  end
end
```

カードを複数枚配列で保持できるインスタンス変数 @card、カードの出し入れとリスト返す処理をする3つのメソッド、 putinCard, drawCard, myCards が追加される。

このように、あるクラスを拡張して別のクラスを作ることで、徐々に複雑さを増した設計をすることができる。

上記のクラス PurseWithCard を利用したプログラム例を示す。

purse-inherit.rb

実行例:

```
% ./purse-inherit.rb
=== 今日からこの財布を使おう。5000円入れておこう: Purse.new(5000)
=== 1598円の買い物をしよう。
=== 2000円出すぞ: saifu.sub(2000)
2000円出しました。
=== おつりを402円もらった。しまおう: saifu.add(402)
402円入れました。
=== いまいくらあるんだろう: saifu.howmuch
残り3402円です
=== 次は4000円のものをカードで買おう!
=== このお店は手持ちのカードで買えますか: Pizza,Mustard
=== じゃあPizzaカードで!
Pizza を取り出しました。
=== じゃあPizzaカードで!
=== やったー、手持ちが足りないのに買えた! 帰ろう。
...
=== ただいま。あ、カードがない! 店に忘れた。
Mustard
```

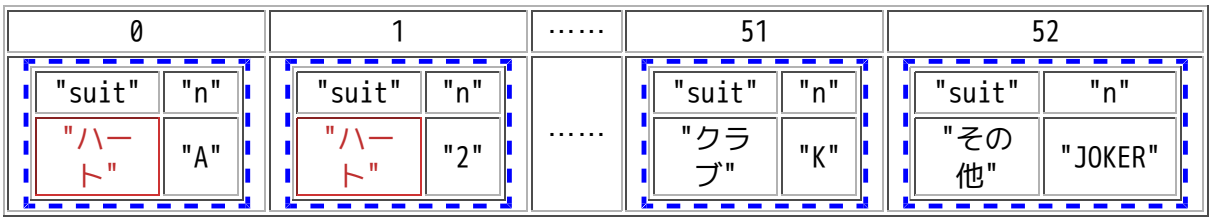
第13講 オブジェクト指向とクラス定義

◆オブジェクト指向

これからもっと大規模なプログラミングに取り組んでいく場合、オブジェクト指向プログラミングの考え方が重要となる。本格的なオブジェクト指向の考え方を全て理解するのは難しいので、ここではその入口だけ触れてみよう。

◇データの内部形式への高依存問題

前回作成したブラックジャックプログラムを改良することを考えよう。前回は52枚のトランプカードを表現する形式を「配列の中のハッシュ」にした。つまり、



のようなイメージとなる。実際にはこの形式にしておいて困ることは当面考えられないが、ここでは仮に

効率が悪いのでデータ形式を変える必要が生じた

ものとして話を進めよう。

配列の中のハッシュをやめて、全て通し番号管理 に変えたい。通し番号管理では、

[0]	[1]	[2]	……	[50]	[51]	[52]
22	3	27	……	19	33	4

というデータ格納形式となる。通し番号管理に変えるには、まず、最初に52 枚のカードを作成するメソッドを変える必要がある。これには、もともとの bj.rbにあった createCardsHashArray メソッドを置き換えて、通し番号を生成するものにすればよい。たとえば、以下のようになるだろう(前々回のもの と同じ)。

```
def createCardsArray()
  card=Array.new                # 空の配列を作る
  0.upto(51) do |x|             # ジョーカー不要なので51まで
    card << x
  end
  card                          # 最後に card配列 を呼び主に返す
end
```

今回やろうとしている改造は、カードの内部構造を変えるものなので、カードを作成するメソッドを変更するのは当然であることが分かる。しかし、構造を変えたからには、それを利用する部分にも影響が

出る。まず、シャッフルするメソッドを考えよう。

シャッフルするメソッドは、

```
def shuffle(a)
  srand
  0.upto(a.length-1) do |i|
    j = rand(a.length)    # 交換相手をランダムに選ぶ
    w = a[i]
    a[i] = a[j]
    a[j] = w
  end
  a
end
```

となっていた。これは仮引数として a を受け取ってはいるものの、それ(a)がどんな構造になっているかはまったく関知していない。たんに配列だと思って値を交換しているに過ぎない。よってこのメソッドは、カードの内部保存形式を変えても全く影響が出ない。

続いて、ブラックジャックのゲームの進行部分を調べてみよう。ブラックジャックでは、カードのスートは全く無視して、数のみを見ている。数を見るのは得点を計算する部分と、画面に表示する部分である。つまり、 dispCard, point 両メソッドに関して、下線で示した部分を書き換えなければならない。

```
def dispCard(card)
  printf("%s の %s\n", card["suit"], card["n"])
end


def point(arr)
  ace = 0               # "A"の枚数
  sum = 0               # 合計点
  for card in arr       # arr配列全ての要素に対して繰り返す
    n = card["n"]       # card["n"]で数を表すvalueが得られる
    case n              # nの値によって場合分け
    when "A"            # "A"ならば…
      ace += 1          # 枚数カウンタを1増やしておく
      sum += 11         # まずは11点として加算
    when "10", "J", "Q", "K"
      sum += 10         # 絵札と10は10点
    else                # それ以外(つまり2…9)はそのものを整数にして足す
      sum += n.to_i
    end # when終わり
  end # for終わり
  while sum > 21 && ace > 0 # 21より大きく、かつ、aceが1以上なら
    sum -= 10           # 10を引く
    ace -= 1            # aceを1つ使ったので減らす
  end
  sum
end
```

さらにもう一点プログラミングに神経を遣わなければならないところがある。それはカードを配るところで、コンピュータに配るところ

```
com << cards[cardNo+=1]
```

と人間に配るところ

```
player << cards[cardNo+=1]
```

が、プログラム中何箇所も登場する。これには以下のような問題点が内在する。

1. cardsが配列でなくなった場合(たとえばハッシュのとき）には、該当部分を全て書き換えなければならなくなる。
2. 変数cardNoをいちいち書くのが面倒(変数名を忘れそう)。
3. cardNoという変数をどこか別の場所で変更したらゲームがおかしくなる。
4. 52枚全部引いたとき(cardNo==52に達したとき)どうするか、という処理をカードを引く箇所全てに書かなければならない。

結局このプログラムを、「通し番号形式」に変えるために書き換えなければならないところをまとめると bj-numbering.rbとなる。変更点の多さに着目せよ。

bj-numbering.rb

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

def createCardsArray()
  card=Array.new                  # 空の配列を作る
  0.upto(51) do |x|               # ジョーカー不要なので51まで
    card << x
  end
  card                            # 最後に card配列 を呼び主に返す
end

def shuffle(a)
  srand
  0.upto(a.length-1) do |i|
    j = rand(a.length)    # 交換相手をランダムに選ぶ
    w = a[i]
    a[i] = a[j]
    a[j] = w
  end
  a
end

def getSuite(n)          # 通し番号からスートに変える
  s = n/13
  ["ハート", "スペード", "ダイヤ", "クラブ"][s]
end

def dispCard(card)
  num = card%13 + 1               # 通し番号から数に変える
  cardN = case num
          when 1
            "A"
          when 11
            "J"
          when 12
            "Q"
          when 13
            "K"
          else
            num
          end
  printf("%s の %s\n", getSuite(card), cardN)
```

```
end


def point(arr)
  ace = 0                # "A"の枚数
  sum = 0                # 合計点
  for card in arr        # arr配列全ての要素に対して繰り返す
    n = (card%13)+1      # 13で割った余りに1足したものが数
    case n               # nの値によって場合分け
    when 1               # "A"ならば…
      ace += 1           # 枚数カウンタを1増やしておく
      sum += 11          # まずは11点として加算
    when 10, 11, 12, 13
      sum += 10          # 絵札と10は10点
    else                 # それ以外(つまり2…9)はそのものを整数にして足す
      sum += n
    end # when終わり
  end # for終わり
  while sum > 21 && ace > 0 # 21より大きく、かつ、aceが1以上なら
    sum -= 10            # 10を引く
    ace -= 1             # aceを1つ使ったので減らす
  end
  sum
end

# カードを作ってシャッフルしたものを cards配列に格納
cards=shuffle(createCardsArray)

# カードを数える変数。-1で初期化
cardNo=-1
   :
   :
（以下は変更点がないので省略）
   :
```

この問題を整理すると以下のようにまとめられる。

- カードの内部構造をゲームの進行部分で意識しなければならないのはあとあと融通が利かないのであまりよくない。
- カードの通し番号のようなちょっとずれただけでゲームが台無しになる繊細な情報をゲームの進行部分で何回も変更しなければいけないようなプログラムは後で改良するときにとてもたいへんで間違えやすくなる。

このような場合には、カードを使ってゲームを作るときに、

> カードってどんな処理ができるものなの?

ということに着目して考える。ブラックジャックゲームを作った経験からは、

- カードはシャッフルされた状態で始まる
- 1枚ずつ順番に配られる
- スートと数を取り出すことができる

ことが最低限必要である。逆に、この条件を満たせば内部構造が、ハッシュのハッシュだろうが、通し番号の配列だろうが、どんなものでも構わない。

ここで今まで、ArrayやHashを利用した経験を思い出そう。Array(配列)を使っているときには、我々はただ単に要素を追加したり取り出したりするだけでよく、要素が今何個入っているかとかは普段は意識しなくてよかったし、並べ換えるときはsortメソッド任せで実際に並べ換えの具体的手順はいっさい知らなくてよかった。Arrayクラスのメソッドが勝手にやってくれた。

そこでカードゲームでも同じように、トランプのカードにかかわるいろいろな処理を持ったCardクラスがあったらいいなと考える。言い換えると、

- Cardクラスの値を一個作るとシャッフルされたカードが用意される
- 1枚ずつ配ってくれるメソッドがある
- 配られたカードを渡すとスートを教えてくれるメソッドがある
- 配られたカードを渡すと数を教えてくれるメソッドがある

のような便利なクラスが使えたらいいのに、と考える。

Rubyによるオブジェクト指向プログラミングでは、そのような性質を満たすものを新しい クラス として宣言し、必要な処理をメソッドとして定義し、別のプログラムでそれを利用する。具体的には、

1. カードそのものを表現する Card クラスを定義する
2. そのクラスの値(インスタンス)を生成し、ゲーム進行部分で利用する

という流れとなる。

◇Cardクラスを利用したプログラム

以上のことをふまえてブラックジャックを「Cardクラスを使う」方式に作り直す。手順は、

1. Cardクラスを定義する
2. Cardクラスを使うゲーム進行部を作る

の2段階となる。1番をcard.rbに、2番を bj2.rbに作成する。

card.rb

```
#
#  「カード」を表現するCardクラスを定義する
#
class Card                          # クラス名は必ず大文字で始める
  # クラス定義におけるinitializeという名前のメソッドは、
  # そのクラスの値(インスタンス)がnewされるときに自動的に
  # 呼ばれるメソッド。
  # 通常initializeメソッドは必ず定義する。
  def initialize()
    # @で始まる変数はインスタンス変数といい、クラスの中で
    # しか利用できないつまりクラス定義の外からは一切いじる
    # ことも見ることもできないので後で変えるかもしれない
    # データ構造はインスタンス変数に閉じ込めるのがポイント
    @card = Array.new           # 全てのカードを保存する配列
    @now  = -1                  # 今何枚目かをしまう変数
    for suit in ["ハート", "スペード", "ダイヤ", "クラブ"]
      @card << {"suit" => suit, "n" => "A"}
```

```
      # ここで代入しているのがハッシュ
      2.upto(10) do |n|
        @card << {"suit" => suit, "n" => n.to_s}
      end
      for n in ["J", "Q", "K"]
        @card << {"suit" => suit, "n" => n}
      end
    end
    shuffle
  end

  def shuffle()                 # @cardをいじるので引数は要らない
    srand
    0.upto(@card.length-1) do |i|
      j = rand(@card.length)    # 交換相手をランダムに選ぶ
      w = @card[i]
      @card[i] = @card[j]
      @card[j] = w
    end
    @now = -1                   # カードの順番-1にしておく。
  end

  def next()                    # つぎの1枚を引く、メソッド
    if @now >= @card.length
      shuffle()
    end
    @card[@now += 1]
  end

  # クラスメソッドの定義
  # クラスメソッドはそのクラスのインスタンス全体に共通のメソッド
  # クラスメソッドは　クラス名.メソッド名 で定義する
  def Card.suit(card)           # スートを得るメソッド
    card["suit"]
  end
  def Card.n(card)              # 数を得るメソッド
    card["n"]
  end

end
```

続いて、上記card.rbを利用したブラックジャック定義プログラムbj2.rbを示す。先頭にある *require* は、別ファイルに書かれたRuby プログラムを取り入れるためのものである。

bj2.rb

```
#!/usr/local/bin/ruby
# coding: utf-8
# ブラックジャック進行部

# Cardクラス定義ファイルを読み込む
require 'card.rb'       # これでCardクラスが使えるようになる

# ブラックジャックに必要なメソッドの定義
# カードのスートや数を調べるときは、Cardクラスのメソッド
# Card.suit() と Card.n() を利用している。
def dispCard(card)
  # Cardクラスのクラスメソッドを使ってスートと数を得る
  printf("%s の %s\n", Card.suit(card), Card.n(card))
end

def point(arr)
  ace = 0               # "A"の枚数
```

```
  sum = 0               # 合計点
  for card in arr       # arr配列全ての要素に対して繰り返す
    n = Card.n(card)    # クラスメソッド経由で数を得る
    case n              # nの値によって場合分け
    when "A"            # "A"ならば…
      ace += 1          # 枚数カウンタを1増やしておく
      sum += 11         # まずは11点として加算
    when "10", "J", "Q", "K"
      sum += 10         # 絵札と10は10点
    else                # それ以外(つまり2…9)はそのものを整数にして足す
      sum += n.to_i
    end # when終わり
  end # for終わり
  while sum > 21 && ace > 0 # 21より大きく、かつ、aceが1以上なら
    sum -= 10           # 10を引く
    ace -= 1            # aceを1つ使ったので減らす
  end
  sum
end

# Cardクラスのインスタンスを生成(既にシャッフルされている)
cards=Card.new

# コンピュータの持ちカードを格納する配列
com=[]
com << cards.next # cardsに属するnextメソッドで次の1枚を引くようにする
com << cards.next

# プレイヤーの持ちカードを格納する配列
player=[]
player << cards.next
player << cards.next

# 最初の手持ちを表示する
puts "わたしの手:"
# コンピュータの1枚目を表示
puts "  [ ??? ]"
# コンピュータの2枚目を表示
dispCard(com[1])        # 2枚目は com[1] に入っている
# プレイヤー側
puts "あなたの手:"
dispCard(player[0])
dispCard(player[1])

# プレイヤーが満足するまで引かせる
while true
  STDERR.print "もう一枚引きますか?(yかnで) "
  answer = gets
  if /^y/i =~ answer
    player << cards.next
    puts "もう一枚引きます。引いたカードはこれです。"
    dispCard(player[-1])
  else
    break               # while true を抜ける
  end
end

# コンピュータが16以下なら引き続ける
while point(com) <= 16
  puts "わたしはもう一枚引きます"
  com << cards.next
end

# いよいよ勝負
```

```
puts "勝負!"
puts "わたしの手:"
for c in com            # コンピュータの手持ちカード全て繰り返す
  dispCard(c)
end
puts "あなたの手:"
player.each do |c|      # プレイヤーの手持ちカード全て繰り返す
  dispCard(c)
end

# 得点を比べる
cp, pp = point(com), point(player)
printf("わたし %d : %d あなた\n", cp, pp)

if cp >= 22             # 22以上なら0点ということにしてしまう
  cp = 0                # (コンピュータ)
end

if pp >= 22             # 22以上なら0点ということにしてしまう
  pp = 0                # (プレイヤー)
end

if pp > cp              # コンピュータより点が高ければ
  puts "あなたの勝ちです。参りました!"
else                    # そうでなければ(引き分け含む)
  puts "わたしの勝ちです。べろべろばあ…"
end
```

ゲーム進行部でカードを配るときに、カード番号を逐一管理しなくてよくなる。また、カードの内部保存構造を変えるのが容易になる。実際に変えてみよう。カードを通し番号管理にしてみる。この場合、Cardクラスの定義だけを変えるだけでよい。通し番号形式に変えたプログラム card.rbが以下のものである。上記のものからの変更点を下線で示す。ブラックジャックゲーム進行部(bj2.rb)は、以前と全く同じままでよいことに注目せよ。

card.rb(通し番号バージョン)

```
# # 「カード」を表現するCardクラスを定義する # 通し番号方式バージョン # class Card
def initialize() @card = Array.new # 全てのカードを保存する配列 @now = -1 # 今何枚
目かをしまう変数 0.upto(51) do |x| # ジョーカー不要なので51まで @card << x end
shuffle end def shuffle() srand 0.upto(@card.length-1) do |i| j =
rand(@card.length) # 交換相手をランダムに選ぶ w = @card[i]
      @card[i] = @card[j]
      @card[j] = w
    end
    @now = -1                   # カードの順番
  end

  def Card.suit(card)
    s = card / 13
    ["ハート", "スペード", "ダイヤ", "クラブ"][s]
  end
  def Card.n(card)
    n = card%13 + 1
    case n
    when 1
      "A"
    when 11
      "J"
    when 12
      "Q"
    when 13
```

```
        "K"
      else
        n.to_s
      end
    end

    def next()
      if @now >= @card.length
        shuffle()
      end
      @card[@now += 1]
    end
  end
```

◆クラス定義

ブラックジャックのプログラムを、Cardクラスを利用するように変えたことで、データ構造の修正がしやすくなるとともに、ブラックジャック以外のカードゲームを作るときの労力が削減できることは想像できただろうか。

オブジェクト指向の考え方の入口に進むために、実際にクラスを設計していく場合の基本的な考え方を見ていこう。

◇クラス定義の仕方

クラス定義は *class* 文によって行なう。

```
class クラス名
  ……定義本体……
end
```

定義したクラスには、プログラミングのときに利用したい対象物の持つ内部情報や、必要な処理手順を埋め込んでいく。たとえば、トランプゲームを作るときには、トランプのカードそのものが必要となる。これに必要な 52(または 53)枚のカードのデータを変数として、シャッフル処理・1枚ずつ配る処理などをメソッドとして定義する。これらは既に作成したとおりである。

定義したクラスを利用する場合には、クラス名の後に .new を付けて新規に生成する。生成されたもののことをそのクラスの*インスタンス* という(インスタンスは「実体」の意味)。

配列を思い浮かべよう。Array クラスのインスタンスを Array.newで生成すれば、そこに自由な配列データを代入していけた。それと同様、カードを表す Card クラスを定義して、Card.new によって生成されたものは1組のトランプカードセットとして使えるようになる。

クラスの中で定義したメソッドは、原則としてそのクラスのインスタンスからしか利用できない。たとえば、

```
class Card
  def next()
    ...
  end
end
```

として、Cardクラス内にnextメソッドを定義した場合は、

```
x = Card.new
x.next
```

のように、Cardクラスの値であるxから nextメソッドを呼ぶことになる。

◇クラス定義の考え方

実用的なプログラミングでは、日常にあるものを計算機上で表現する場合が多い。そのようなときに、実際に対象物を取り扱うのに必要な典型的な手順がいくつか考えられるような場合、それはクラス定義にするとよい。

たとえば、五目並べゲームを作る場合、必要なのは碁盤と碁石のセットである。それをRubyの変数でどのように表現するか、また、石を置いたりするときに変数をどう変更するかを考える。これをできるだけ抽象的にしておいて、五目並べを作るときにも、囲碁を作るときにも利用できるようにすることを目標にする。たとえば、碁盤と碁石の場合、考えられる処理は以下のようになるだろう。

- 何も置いていない碁盤を用意する(new)
- 指定した座標に黒石を打つ
- 指定した座標に白石を打つ
- 指定した座標の石を取り除く
- 指定した座標に何が置いてあるか教えてくれる
- 現在の盤面を表示する
- 盤面をリセットして全部取り除く

他にもあるかもしれないが、この程度の処理を考えておけば碁盤を利用する色々なゲームで利用できそうである。ということで、碁盤を実現する Gobanクラスを考えると以下のような骨組になる。

```
class Goban
  def initialize()      # newされたときに自動的に呼ばれる
    # 碁盤を初期化する
    # 盤面を配列とかで作ったりする
    # 黒石、白石の個数を保存する変数を初期化したりする
  end
  def putBlack(x, y)
    # 座標（x, y）に黒を置くメソッド
    # 黒石の残り数を1減らしたりするといいかもね
    # 置ける場合は残りの打点数を、置けない場合は false を返す。
  end
  def putWhite(x, y)
    # 座標（x, y）に白を置くメソッド
    # 白石の残り数を1減らしたりするといいかもね
    # 置ける場合は残りの打点数を、置けない場合は false を返す。
  end
  def remove(x, y)
    # 座標（x, y）の石を取り除くメソッド
    # 取り除いたのが黒か白かに応じて残り数を復活させる
  end
  def at(x, y)
```

```
    # 座標（x, y）に何があるかを返す
    # "白" か "黒" か nil か、かな(nilは何もない)
  end
  def displayGoban()
    # 現在の盤面を画面表示するメソッド
    # forとかwhileで表示するんだろうね
  end
  def reset
    # 全ての盤面情報をリセットして
    # 黒石、白石の個数を保存する変数を初期化したりする
  end
end
```

これを使うと、五目並べゲームは

```
board = Goban.new
while true
  print "どこに打ちますか? (x, y): "
  line = gets.chomp!
  if /(\w), *(\d)/ =~ line      # A, 5 とか入れてもらうはず
    x, y = $1, $2               # x="A", y="3" とかになるはず
  else
    redo
  end
  if !board.putBlack(x, y)      # 人間の手を置く
    puts "そこには置けないよ!"
    redo
  end
  if check5(board)              # 5目並んでるか調べるメソッド
    puts "あなたの勝ちです!"
    break
  end
  x, y = thinkComputer(board)   # COM思考ルーチン(難しいのでまだ無理ね!)
  board.putWhite(x, y)          # コンピュータの手を置く
  if check5(board)              # 5目並んでるか調べるメソッド
    puts "わたしの勝ちです!"
    break
  end
end
```

というような流れで作ることができるだろう。もちろん、プログラムの中に登場する未知のメソッドは自分で作らなければならず、それはかなり難しいものになるだろう。しかし、Goban クラスを定義して、それを利用したプログラムにすることで、碁盤を配列で表現してもハッシュで表現しても、通し番号で表現しても、あるいはもっとすばらしい内部構造で表現しても、五目並べのゲーム進行部分は一切修正せずに済むことになる。さらには、囲碁プログラムを作る場合にも同じクラス定義を利用でき大幅な作業量削減につながる。

◇Rubyにおけるクラス定義

Rubyでのクラス定義に関しては以下の決まりがある。

- クラス名は必ず英大文字で始める
- *クラス名*.new によってクラスの実体(インスタンス）が1つ生成される
- クラス内で @ 記号で始まる変数を利用すると、クラスのインスタンス内のみで有効なものになる

- initialize という名前のメソッドはインスタンスが生成されるときに必ず呼ばれるので、このメソッドで必要な初期化を行なう
- クラスメソッドは *クラス名*.メソッド名 という名前で定義する。クラスメソッドは、プログラムの任意の箇所で利用することができる。

◇結局クラス定義って…?

　クラス定義、に代表されるオブジェクト指向の考え方は、現在では本格的なプログラミングを行なうときには必須となってくるものである。その考え方を深く学びたい場合は、Ruby言語、またはJava言語を深く学習するとよい。

第14講 CGIプログラム

◆CGI入門

HTML文書内にあるデータ入力窓とそれらを受け渡すスクリプト名を書いておき、ボタンを押すとそのスクリプトに入力値が渡るようなしくみを*CGI*(Common Gateway Interface)という。

◇CGIのしくみ

CGIのデータ入力窓の集合を*入力フォーム*という。以下の入力フォームに値を入力、あるいは選択し「OK」ボタンを押してみよう。

名前↓
さんの誕生日は?
1997 年 月 日
OK reset

上の入力窓に入れられた値は全て*入力名*という特別な変数でCGIスクリプトに送られる。自分で作るプログラムではその変数を受け取りそれに応じた処理をした上で、結果を表すHTML文書を出力する。

```
[taro]さんの誕生日は?
[1992]年[ 3]月[1]日
[OK][reset]
```

→

```
「Rubyスクリプト」
値を受け取り
HTMLを出力
```

→

```
<html>
<head>
<title>結果発表!</title>
</head>
</body>
<p>........</p>
</body>
</html>
```

◇CGIの作り方

CGIは、利用者にデータを入力してもらうためのHTML文書とそこから送られて来るデータを受け取って処理をするプログラムファイルを作る。さらに、Webサーバに対して「ここでCGIプログラムを利用する」ことの宣言を行なう必要がある。

1. CGI利用宣言 → .htaccessファイルの作成
2. 入力フォームのHTML文書作成
3. 処理プログラムの作成

順を追って行なう。なお、慣れたあとは2と3をまとめて1つの Ruby プログラムで行うほうが楽であるが、 HTMLとCGI実行プログラムは分けたほうが理解しやすいので以下の説明は分ける方法で記述する。

◇CGIプログラムの利用宣言

「この場所でCGIを利用する」ためのディレクトリを作成し、そこに .htaccessという名前のファイルを作成する。ここではたとえば、 ~/public_html/mycgi というディレクトリを作成し、そこでCGI利用宣言をするものとする。まず

```
% cd ~/public_html
% mkdir mycgi
```

としてディレクトリを作成する。次にEmacsで ~/public_html/mycgi/.htaccess ファイルを新規に開き、以下の内容を記述して保存する(コピーペーストしてよい)。

```
AddHandler cgi-script   .rb
Options +ExecCGI
AddType "text/html; charset=utf-8" .html
```

1行目と2行目により、このディレクトリに作成する .rb という名前で終わるファイルはCGIスクリプトと認識される。3行目の記述により、 HTMLファイルの文字コードが UTF-8 であることを宣言する。RubyプログラムをUTF-8で作成するので、HTMLファイルもそれに合わせる。

◇HTML文書の構成

次は入力フォームを構成するHTML文書を作る。入力フォームは form要素を使って記述する。form要素は

```
<form method="メソッド" action="./スクリプト">
 ～～～
</form>
```

のように記述する。メソッドの部分は GET または POST のどちらかを指定する。文章など、長いデータを入力させたいときはPOSTを指定する。

以下のform要素例は

- namaeという入力名で最大40字の文字列を
- bloodという入力名で A, AB, B, O いずれかの選択肢を

入力させ、./btype.rb というスクリプトに入力値を渡すためのHTML記述である(該当部分抜粋)。

```
<form method="POST" action="./btype.rb">
<p>お名前: <input name="namae" type="text" maxlength="40"><br>
血液型: <select name="blood">
  <option>A</option>
  <option>AB</option>
  <option>B</option>
  <option>O</option>
 </select><br>
<input type="submit" value="OK">
<input type="reset" value="reset"><br></p>
</form>
```

上記の入力フォームのみを含むHTML文書の例を btype.html に示す(ソース)。

次節で、./btype.rb の作り方を示す。

◇CGIスクリプトの構成

CGIスクリプトとなるRubyプログラムでは、最低限以下の内容を記述して作成する。

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

require 'cgi'
c = CGI.new(:accept_charset => "UTF-8")
print "Content-type: text/html; charset=UTF-8\n\n"
```

変数cはどんな名前でもよい。このあとに、HTML文書となり得る文字列を順次出力していく。

require は、プログラムに必要な外部モジュールを現在のプログラムに取り込む記法で、

```
require 'cgi'
```

はCGIプログラムを組むのに便利なクラスが定義された cgi.rb を取り込む宣言である。これで以下のようにCGIクラスが使えるようになる(CGI.newの部分)。

CGIプログラムに渡された入力値を取り出すには、CGI.new を受け取った変数のハッシュ値として取り出す。以下の例はform入力されたパラメータ namae と blood を単にHTML文書形式で出力するRubyプログラムである。

btype.rb

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

require 'cgi'
c = CGI.new(:accept_charset => "UTF-8")
print "Content-type: text/html; charset=UTF-8\n\n"

name = c["namae"]  # フォームに記入されたnamaeの値を取り出す
bt   = c["blood"]  # フォームに記入されたbloodの値を取り出す

# 「<<EOF」とすると、次に見つかる「EOF」まですべてを文字列として処理する
# これを「ヒアドキュメント」という
print <<EOF
<!DOCTYPE html>
<html>
<head><title>Blood type</title></head>
<body>

<h1>#{name}さんの血液型</h1>

<p>#{name}さんは#{bt.upcase}型ですね!</p>
</body>
</html>
EOF
# ↑このEOFまでがすべて文字列として処理される。CGI等で便利。
```

プログラム中現れる << は*ヒアドキュメント* といい、その次の行から、 <<の直後に書かれた単語が見つかる行までをまとめて 1つの文字列として括る記号である。改行文字を含む長文文字列を扱いたいときに便利な記法であり、文字列中にダブルクォートやシングルクォートを自由に入れることもできるた

め、ダブルクォートを多用することの多い HTML 文の出力には有用である。

さらにこの文字列中で利用している #{...}という記法はその部分をRubyの式としてその値に置き換える働きを持つ。たとえば、

```
print "1+2は#{1+2}です\n"
```

とすると、

```
1+2は3です
```

と出力される。HTML出力文に変数結果を入れたい場合に #{...} は有用である。

このCGIを動かす例が以下の入力フォームである。

お名前:
血液型: A
OK reset

◇処理の切り替え

少し進んで、入力した値により出力を切り替える例を作成してみよう。入力されうる値をキーとして、対応する文字列をハッシュで登録しておき、メッセージ出力部分をハッシュの値を引くことによって切り替える。

お名前:
血液型: A
OK reset

btype2.rb

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

require 'cgi'
c = CGI.new(:accept_charset => "UTF-8")
print "Content-type: text/html; charset=UTF-8\n\n"

name = c["namae"]
bt   = c["blood"]

kekka = {                       # A, B, AB, O に対応する値をHashで設定する
  "A"   => "えーっすなあ",
  "AB"  => "だぶる!",
  "B"   => "びびっとね",
  "O"   => "おーすげー",
}
kekka.default = "人間なの??"    # Hashのデフォルト値(キーがない場合の値)

bt.upcase!      # 念のため大文字に変えておく

# ここ↓の <<EOF の部分が次の行からEOF行手前までの文字列に置き換わる
printf(<<EOF, bt, kekka[bt])    # 8行下の%sに入る
<!DOCTYPE html>
<html>
```

```
<head><title>Blood type</title></head>
<body>

<h1>#{name}さんの判定結果</h1>

<p>%s型のあなたは%s</p>
</body>
</html>
EOF
```

◇CGIプログラムのデバッグ

> プログラム作成作業を進めるに当たって、ここの手順を取らないと非常に苦労する。

簡単なRubyプログラムでも、CGIとして動かすには困難がともなう。なぜなら、プログラムを起動して出てくるエラーメッセージなどが見えないからである。文法エラーなどもWebブラウザ経由では見ることができない。

Web上で動かすプログラムは以下の手順で段階的に作成・デバッグする。

> 1. ターミナルで直接動かしてエラーをなくす
> 2. ローカルPC(目の前で使っているPC)で動かして詳細デバッグ
> 3. 本番サーバで動かして最終調整

■(0)ターミナルで起動できるかだけテスト

CGIプログラムを保存したディレクトリに移動して、作成したプログラム(例: btype.rb)を起動する。

```
% ./btype.rb
(offline mode: enter name=value pairs on standard input)
[Ctrl-D]
```

このように入力値を求めてくれば単純な文法エラーがないことが確認できる。これすら出ない場合はすぐ修正する。C-d を押して入力修了を指示すると、HTML文が出てくるはずで出てくれば最低限の準備はできているので次に進む。

■(1)ターミナルでテスト

cgi.rb モジュールを利用したプログラムはCGI経由だけでなく、ターミナル上でもデバッグできる。上記の btype.rb の場合は、HTMLからプログラムに渡す入力名として namae, blood を利用している。nameに「太郎」を、bloodに「A」を代入した状態でデバッグする場合以下のように起動する。

```
% echo 'name=太郎&blood=A' | ./btype.rb
```

つまり、入力名の値を

'*入力名$_1$*=*値$_1$*&*入力名$_2$*=*値$_2$*&*入力名$_3$*=*値$_3$*&...'

の形式でechoコマンドで出力し、実行プログラムに送り込めばよい。まとめると、以下のようなプログ

ラム起動で確認する。

```
% echo '入力名1=値1&入力名2=値2&入力名3=値3' | ./作成中のプログラム.rb
```

正しい(と思われる)HTML文が出力されたらローカルPCのWebサーバでテストする 2の手順に進む(手順3に先に進んで詰まったら次の2に戻るのでもよい)。

■(2)ローカルWebサーバでのテスト

本来Webサーバは専用サーバで動かすものだが、Rubyには簡易Webサーバ機能があり手軽に動かすことができる。

webserv.rb

保存したwebserv.rbを起動すると以下のようなメッセージが出る:

```
% chmod +x webserv.rb
% ./webserv.rb
http://localhost:1234/ファイル名
で接続してください
```

「ファイル名」の部分を作成中のCGIファイルにしてブラウザで開く。たとえばbtype.htmlを作成中の場合はブラウザで

```
http://localhost:1234/btype.html
```

を開く(1234は人によって違う)。もしエラーが出ると、webserv.rb を起動したターミナルにエラーメッセージが出るので自分の作ったプログラムに出された部分を選んで確認し、修正する。

■(3)正式WebサーバのURLで閲覧してテスト

以上のような起動方法で、正しくHTML文書が出力されたら正式Webサーバの URLで開く。

◇formで使える入力用要素

HTML文書の form 要素内で、ユーザからの入力を促すための主な要素を実例とともに示す。

以下の説明脇にある入力フォームに実際の値を入れて［送信］ボタンを押すと、この例にある入力名がどのように渡るかの結果が示される。実際に、いろいろな値を入れて試してみること。

■input要素

なんらかの値を入力させる場合に、input要素に入力方法と入力名を指定する。入力方法は type属性で、入力名は name 属性で指定する。以下に入力方法の種別と使用例を示す。他の属性も付けた例を示すが、最低限必要なのは name である。 type を省略すると text と見なされる。また、例中にある label 要素は入力に対応するラベルを結び付けるものである。

以下の例では，「送信」ボタンを実際に押すことで入力名がどのように Ruby プログラムに渡るかが分かるようになっている。

text

1行だけの文字列入力。

```
<p><label>メッセージをどうぞ:
<input name="v_text" type="text" value="Hello" size="40">
</label></p>
```

入力名 v_text に初期値 "Hello" を入れて40字分の枠を作る。

メッセージをどうぞ: Hello [送信]

初期値を省略すると空欄になる。

password

text と同様だが、入力文字をアスタリスクで隠す。

```
<p><label>パスワード:
<input name="v_password" type="password" size="20" maxlength="12">
</label></p>
```

入力名 v_password に最大文字数12字のパスワード入力を入れる。

パスワード: [送信]

ただし、パスワードの値がCGIプログラムに渡されるときは *暗号化などは一切されない* ので、盗聴されても構わないものでしか使えない。

checkbox

チェックボックスを作る。ONのときの値とOFFのときの値を value 属性で指定する。

```
<p><label><input name="v_checkbox_1" type="checkbox"
 value="morning">朝御飯OK</label><br></p>
<p><input name="v_checkbox_2" type="checkbox"
 value="night">夕飯OK</p>
```

「朝御飯OK」がチェックされていたら入力名 v_checkbox_1 を "morning" に「夕飯OK」がチェックされていたら入力名 v_checkbox_ に "night" を代入させる。label 要素はチェックボックスとラベル文字列をグループ化し、ラベル文字列のクリックでチェックが入るようになる。「朝御飯OK」はlabel要素で結びついているので文字列クリックでよいが「夕飯OK」はボタンをクリックしないとチェックが入らない。

☐ 朝御飯OK

☐ 夕飯OK

[送信]

チェックされないときは空文字列 "" が代入される。

radio

ラジオボタンを作成する。name属性で同じ入力名名を持つものどうしがグループ化され、どれかひとつをチェックすると他はクリアされる。

```
<p>
<label><input name="v_radio" type="radio" value="1">地球人</label>
<label><input name="v_radio" type="radio" value="2">宇宙人</label>
<label><input name="v_radio" type="radio" value="3">仙人</label></p>
```

○地球人 ○宇宙人 ○仙人

たとえば「仙人」を選ぶと入力名 v_radio に "3" が代入される。どれも選択されていない場合は空文字列が代入される。

一度押してしまったラジオボタンをクリアさせるには次の reset ボタンを配置しておく。

submit, reset

入力値の送信またはリセットボタンを作成する。

```
<p><input type="submit" value="[送信]">
   <input type="reset" value="[リセット]"></p>
```

[送信] [リセット] ←これを押すと上記の入力値がすべてリセットされる。

hidden

隠れ変数を設定する。CGIプログラムに必ず渡したい値を見えないところで指定する。

```
<input name="v_hidden" type="hidden" value="第3ステージ">
```

上記を書いてもWebページには何も表示されないが CGIプログラムには入力名 v_hidden に「第3ステージ」という値が代入される。

■select要素

選択肢を提示してそのうち1つまたは複数を選ばせるには selectとoptionを組み合わせる。

```
好きな季節
<select name="v_select">
 <option>春</option>
 <option value="Summer" selected>夏</option>
 <option value="Autumn">秋</option>
 <option value="Winter">冬</option>
</select>
```

好きな季節 [夏 ⌄]

(value指定のない「春」を選択すると「春」そのものが値となる)

このように、selected を付けておいたオプションが選択初期値となる(上の場合「夏」)。 selected を付けなければ先頭項目が選択された状態で始まる。

select に multiple 属性を付けると複数選択が可能になる。

```
好きな曜日
```

```
<select multiple name="v_select_multi">
 <option value="Sunday" selected>日</option>
 <option value="Monday">月</option>
 <option value="Tuesday">火</option>
 <option value="Wednesday">水</option>
 <option value="Thursday">木</option>
 <option value="Friday">金</option>
 <option value="Saturday" selected>土</option>
</select>
```

好きな曜日 日 月 火 水

Firefoxで複数選択するには Ctrlクリック で行なう。

[送信] [リセット]

■テキストエリア

長文を入力させるためには textarea を用いる。入力エリアの行と桁数はそれぞれ rows、cols 属性で指定する。

```
<textarea name="v_textarea" cols="60" rows="5">
これを消してメッセージをどうぞ
</textarea>
```

これを消してメッセージをどうぞ

[送信] [リセット]

■その他のフォーム

フォーム制御の詳細は HTMl Standard 4.10 フォーム が参考になる。

◇練習問題

次のRubyプログラムはinput要素に値を入力してOKを押すと、次の画面に進む、を繰り返すものである。

cgiloop.rb（これと同時に cgiloop.css も保存する）

```
#!/usr/bin/env ruby
# -*- coding: utf-8 -*-
require 'cgi'
c = CGI.new
input = c["name"]
myname = File.basename($0)
# $0はプログラムのフルパス名、File.basename($0)するとファイル名部分のみ取り出す
```

```
print <<~EOF
        Content-type: text/html; charset=utf-8

        <!DOCTYPE html>
        <head><title>CGIループ</title>
        <link rel="stylesheet" type="text/css" href="cgiloop.css">
        </head>
        <body>
        <h1>CGIループだよ</h1>
EOF

if !input.empty? then
  printf("<p class=\"shout\">%sさんの明日にいいことがありますように!</p>\n",
         input)
end
print <<~EOF
        <form action="#{myname}" method="POST">
         <p>お名前をどうぞ:</p>
         <p><input type="text" name="name">
         <input type="submit" value="OK">
         <input type="reset" value="Reset">
        </form>
        </body></html>
EOF
```

1. このプログラムを ~/public_html/mycgi/ (またはcgi-scriptが有効になっているディレクトリ)に保存して chmod +x cgiloop.rb し、ブラウザでURLアクセスして挙動を確かめよ。
2. input 要素の type="text" で名前を入力させる部分を type="checkbox" に変えて何かを入力させるものを作成し挙動を確認せよ。
3. 同様に type="radio" に変えたものを作成し挙動を確認せよ。
4. select要素で複数の選択肢から1つを選ぶように変えたものを作成し挙動を確認せよ。
5. select要素で複数の選択肢から複数を選べるように変えたものを作成し挙動を確認せよ。
6. textarea要素を使って長文を入力させ、それをメッセージとして表示できるプログラムを作成し、挙動を確認せよ。

◆検索主体のCGIプログラム

整理されて格納された情報から，必要なものを検索表示する機能はそれを必要とするできるだけ多くの人が利用できる状態にしておくことが有用である。そのために，Webブラウザさえあれば利用できるCGIプログラムの形にするのは有効である。

ここでは，以前作成した履修科目データベース検索をCGI化してみる。

◇コンソールプログラムのCGI化

コマンドラインで起動し，入出力をすべてコマンドラインで完了させるプログラムのことをコンソールプログラムという。コンソールプログラムで値を利用者から読み取るときは主に gets を利用しその結果を変数に代入した。以下がその一例である。

```
STDERR.print("年齢を入力して下さい: ")
age = gets.to_i
```

このような形式のものをCGI化するには，入力に関る部分を CGIパラメータから受け取るように変え，出力する部分を HTML 形式で書き出すように変えればよい。 csv-search.rb では，「開講時期」と「担当教員」の2つの検索条件を入力している。前者を入力名 jiki，後者を入力名 kyoi で受け渡すことを見越してHTMLフォームを記述すると以下のようになる。

```
<form method="POST" action="db-cgi.rb">
 <p>検索条件を入力して下さい。</p>
 <table border="1">
  <tr><td>開講時期</td><td><select name="jiki">
                             <option value="">------ (指定なし)</option>
                             <option>春学期</option>
                             <option>秋学期</option>
                            </select></td></tr>
  <tr><td>担当教員</td><td><input name="kyoi" size="8"></td></tr>
 </table>
</form>
```

この記述により，ブラウザには以下のような入力フォームが出る。

検索条件を入力して下さい。

開講時期	------ (指定なし)
担当教員	

これにより入力される値を入力名 jiki，kyoi から受け取るようなRubyプログラムは以下のようになる。

```
require 'cgi'
c = CGI.new(:accept_charset => "UTF-8")
term = c["jiki"]
whom = c["kyoi"]
```

以上のような置き換えに注意し，結果出力をHTML形式で行なえば CGI化は完了する。以下の書き換え例では，結果を桁揃えして読みやすくなるよう table にして出力している。

search-cgi.rb

```
#!/usr/bin/env ruby
# coding: utf-8
Encoding.default_external = 'utf-8'     # UTF-8のCSVファイルを読むため
require 'cgi'
require 'csv'
c = CGI.new(:accept_charset => "UTF-8")

#データベース読み取り処理は全く同じ
csv = CSV.read("syllabus.csv", headers: true)
db  = csv               # あとで全データを使う場合にそなえcsv変数は温存

# 検索パターンはHTMLフォームへの入力値を取得する
term = c["jiki"]
whom = c["kyoi"]

if term > ""                    # 開講時期指定に何か文字列を入れたなら
  ptn = Regexp.new(term)        # 文字列を正規表現に変換
  db = db.select {|row| ptn =~ row["開講時期"]}
```

```
end
if whom > ""                    # 担当教員指定に何か文字列を入れたなら
  ptn = Regexp.new(whom)        # 文字列を正規表現に変換
  db = db.select {|row| ptn =~ row["担当教員"]}
end

print("Content-type: text/html; charset=utf-8

<!DOCTYPE html>
<html>
<head><title>検索結果</title>
<style type=\"text/css\">
<!--
 tr.head {background: cyan;}
 th {width: 8em;}
 -->
</style>
</head>
<body>")
puts("<h1>該当科目一覧</h1>")
puts("<table border=\"1\">")
db.each {|row|
  printf("<tr class=\"head\"><td colspan=\"2\">%s</td></tr>\n",
         row["科目名"])
  row.each {|key, value|
    printf("<tr><th>%s</th><td>%s</td></tr>\n", key, value)
  }
}
puts("</table>")
puts("</body>
</html>")
```

上の例(下線部)にあるように、Rubyプログラムで " " で囲まれた文字列で、HTMLの文としての " " を出すときは、直前にバックスラッシュを追加してエスケープし、 Rubyプログラムとしての文字列がそこで終わらないように気をつける。

以下の入力フォームによって実際に検索できる。

検索条件を入力して下さい。

開講時期	------ (指定なし)
担当教員	

検索 リセット

◇入力フォームの自動出力

search-cgi.rb の利用にはこのプログラムを呼ぶためのHTML文書を用意する必要があった。その中で，「開講時期」の検索オプションとして「春学期」と「秋学期」の値を選択肢に提示した。この場合選択肢が2つなので問題ないが，これが多数になると option 要素一覧を用意するのが困難になる。また，データに選択肢が増えるたびに HTMLのフォームも修正する必要があるため，互いに食い違いが発生する危険もある。

そのため，データの値に関連する選択肢を入力フォームに入れるような場合は入力フォーム自体もデータの値を管理するCGIプログラムで生成するほうが望ましい。

たとえば，ここで例に挙げている履修科目データベースは， 1科目分の情報を持つハッシュにキー "開講時期" を与えて取り出せるものすべてを集めれば選択肢一覧が得られる。 search-cgi.rb 中の変数 db から「開講時期」の値一覧を得るには以下のようにすればよい。ヘッダオプション(headers) を true に設定した CSV オブジェクトでは添字に見出し名をキーとして設定すると列の値が配列として得られる。つまり、

```
db["開講時期"]
```

は ["春学期", "秋学期", "春学期"] となる。さらに uniq メソッドを組み合わせると重複をすべて削った一覧が得られるので

```
db["開講時期"].uniq
```

によって開講時期の値一覧の重複のない配列が得られる。これを応用し option リストにするには以下のようにすると効率的である。

```
db["開講時期"].uniq.collect {|j| "<option>#{j}</option>"}.join("\n")
```

これにより

```
<option>春学期</option>
<option>秋学期</option>
```

という文字列が得られる。

以上ことをふまえつつ，以下の方針でHTML文を出力する。

```
1. データベースを読み込む
2. 自分自身を呼ぶform文を出力する
3. formの値が入力されていたら検索結果を出力
```

これをプログラム化したものを示す(実行例)。この例ではprint文のところで <<*区切り* のような記法を用いている(*ヒアドキュメント*)。これは、 *区切り* が行の先頭に登場する部分までをひとつの文字列と見なす記法で、複数行に渡る長い文字列を扱うのに便利である。

form-search-cgi.rb

```
#!/usr/bin/env ruby
# coding: utf-8
Encoding.default_external = 'utf-8'
require 'csv'
require 'cgi'
c = CGI.new(:accept_charset => "UTF-8")

#データベース読み取り処理は全く同じ
db = CSV.read("syllabus.csv", headers: true)
opts = db["開講時期"].uniq.collect {|j| "<option>#{j}</option>"}.join("\n")

# HTMLヘッダと入力フォーム出力
print <<EOF
Content-type: text/html; charset=utf-8

<!DOCTYPE html>
<html>
<head><title>科目検索</title>
<style type="text/css">
```

```
<!--
 tr.head {background: cyan;}
 th {width: 8em;}
 -->
</style>
</head>
<body>
<form method="POST" action="form-search-cgi.rb">
 <h1>科目検索</h1>
 <p>検索条件を入力して下さい。</p>
 <table border="1">
  <tr><td>開講時期</td><td><select name="jiki">
                            <option value="">------ (指定なし)</option>
EOF

print(opts)     # これで自動生成した option 一覧が出る

print <<EOF
</select></td></tr>
  <tr><td>担当教員</td><td><input name="kyoi" size="8"></td></tr>
 </table>
 <p><input name="ok" type="submit" value="検索">
 <input name="cl" type="reset" value="リセット"></p>
</form>
EOF
# 検索パターンはHTMLフォームへの入力値を取得する
term = c["jiki"]
whom = c["kyoi"]

if term>"" or whom>""           # 検索条件の最低1つが入力されていたら
  if term > ""                  # 開講時期指定に何か文字列を入れたなら
    ptn = Regexp.new(term)      # 文字列を正規表現に変換
    db = db.select {|row| ptn =~ row["開講時期"]}
  end
  if whom > ""                  # 担当教員指定に何か文字列を入れたなら
    ptn = Regexp.new(whom)      # 文字列を正規表現に変換
    db = db.select {|row| ptn =~ row["担当教員"]}
  end

  puts("<h2>該当科目一覧</h2>")
  printf("<p>検索語: 開講時期=[%s]，担当教員=[%s]</p>", term, whom)
  puts("<table border=\"1\">")
  db.each {|row|
    printf("<tr class=\"head\"><td colspan=\"2\">%s</td></tr>\n",
           row["科目名"])
    row.each {|key, value|
      printf("<tr><th>%s</th><td>%s</td></tr>\n", key, value)
    }
  }
  puts("</table>")
end
puts("</body>\n</html>")
```

◆紙芝居のようにページを進めるCGI

CGIプログラムは，起動してHTMLデータを出力したら終了する。ユーザがHTML文書を目にするときにはプログラムの実行は終了している。これはつまり，フォーム文で入力した値はすぐに消えてしまうことを意味する。

ここでは，1つのCGIプログラムで紙芝居のように順次移り変わる Webページを生成する方法の例を示

す。

◇hidden変数の活用

1つのCGIプログラムに複数のWebページ出力を管理させるには，現在の状態を示す変数をhidden変数に埋め込んで，次に起動されるCGIプログラムでその値を参照して出力するHTMLを切り替えればよい。

たとえば，以下のような流れで進むWebページ遷移を考える。

(1)		(2)		(3)
こんにちは	→	やあやあ	→	さようなら

このような場合，表示すべきページ番号をhidden変数で管理する。利用者の見えないところで値の受け渡しのできるhidden変数が便利である。 CGIプログラムでは以下のような方針で出力内容を切り替える。

1. 出力すべきページの指定がない場合 → 1ページ目を出力
2. 出力すべきページの指定がある場合 → そのページを出力

ページ番号とそのページでの出力内容を以下のようなハッシュで定義しておく。

```
page = {
  "1" => "こんにちは",
  "2" => "やあやあ",
  "3" => "さようなら",
}
```

出力すべきページ番号は，入力名 page で管理するものとすると， HTMLでは以下のような form を記述すればよい。

```
<form method="POST" action="page2go.rb">
 <input type="hidden" name="page" value="ページ番号">
</form>
```

page2go.rb

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

require 'cgi'
c = CGI.new(:accept_charset => 'utf-8')

page = {
  "1" => "こんにちは",
  "2" => "やあやあ",
  "3" => "さようなら",
}
p = c["page"]
if p==""
  p="1"                          # page変数指定がなければ "1" とする
end
name = c["name"]
```

```
puts "Content-type: text/html; charset=utf-8

<!DOCTYPE html>
<html>
<head><title>Page #{p}</title></head>
<body>
<h1>Page #{p}</h1>
<form method=\"POST\" action=\"page2go.rb\">"

# ページ番号による切り替え処理
case p
when "1"
  nextpage = "2"
when "2"
  nextpage = "3"
when "3"
  nextpage = nil                # 「次のページ」はなし
end
if name > ""                    # nameが定義されていたら呼びかけ文出力
  printf("<p>%sさん!</p>\n", name)
end
printf("<p>%s</p>\n", page[p])  # ページ番号に応じたメッセージ出力

if nextpage
  if name == ""                 # 名前入力がなければ名前入力フォーム
    puts('<p>お名前: <input type="text" name="name" size="10"></p>')
  else
    # 既に入力されていたら hidden 変数に保存
    printf("<input type=\"hidden\" name=\"name\" value=\"%s\">\n", name)
  end
  printf("<input type=\"hidden\" name=\"page\" value=\"%s\">\n", nextpage)
  puts('<input type="submit" value="次へ">')
end

puts "</form>
</body>
</html>"
```

◆データを永続させるCGI

アンケートなどの処理では過去に登録された値全てを累積的に保存しておき、統計処理をしたい。ところがCGIでは、 CGIスクリプトが呼び出されて終了する間しか変数の値は生きていない。つまり、<form>のあるWebページで入力された値は CGIスクリプトの終了とともに消えてしまう。入力名に入れられた値を残すためには、そのときの変数などの状態をファイルに保存して、次の起動時にそれを読み込むようにしなければならない。

Rubyでは、現在の変数の値をそのままの形でファイルに保存することが容易にできる。これにはPStoreクラスを使う。

◇PStoreクラス

PStoreクラスを用いると数値、文字列、配列、ハッシュ、どんな値でも直接ファイルに保存できる。文字列を保存することはこれまでも open でできたのでPStoreで保存しても面白くない。ここではハッ

シュの値を保存する例を示す。以下の例は変数xを介して保存させる。

```
require 'pstore'
x = PStore.new("値保存ファイル")
x.transaction do
  x["foo"] = x["foo"] || Hash.new       # x["foo"] が空なら新規ハッシュ代入
  保存したい変数 = x["foo"]
     :
     :
end
```

実際に動く短い例を見ていこう。

◇PStoreクラスの利用法

最初はCGIではなく普通のRubyスクリプトで、途中で設定した変数を保存させるようにしていく例を見てみよう。「名前」と「一言」を読み込んで出力するだけのプログラムを作ってみる。あとから複数の値のペアを追加することを見越して、ハッシュを利用する。

word.rb

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

word = Hash.new

print "名前は?: "
name = gets.chomp
print "ひとこと: "
word[name] = gets.chomp

for who, wd in word     # または word.each do |who, wd|
  printf("%sさんのひとこと「%s」\n", who, wd)
end
```

最後で、ハッシュに含まれるキーと値のペアを全て出力しているが、プログラムを一度動かしても1つのキーと値しか登録されないので何度実行しても1人の言葉しか出てこない。

```
% ./word.rb
名前は?: taro
ひとこと: hoge
taroさんのひとこと「hoge」
% ./word.rb
名前は?: hanako
ひとこと: やほー
hanakoさんのひとこと「やほー」
% ./word.rb
名前は?: John
ひとこと: Hey, Jude
Johnさんのひとこと「Hey, Jude」
```

これをPStoreを使って、そのときのハッシュ全体の値を別ファイル（以下の例ではword.db)に保存して毎回読み込むようにする。

word-pstore.rb

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

```
require "pstore"
x = PStore.new("word.db")
x.transaction do
  x["word"] = x["word"] || Hash.new     # x["word"] ||= Hash.new でも可
  word = x["word"]

  print "名前は?: "
  name = gets.chomp
  print "ひとこと: "
  word[name] = gets.chomp

  for who, wd in word
    printf("%sさんのひとこと「%s」\n", who, wd)
  end
end
```

このようにすると、以前に起動された値を word.db ファイルに自動的に保存し、前回の値を引き継ぐことができる。

```
% ./word.rb
名前は?: taro
ひとこと: hoge
taroさんのひとこと「hoge」
% ./word.rb
名前は?: hanako
ひとこと: やほー
hanakoさんのひとこと「やほー」
taroさんのひとこと「hoge」
% ./word.rb
名前は?: John
ひとこと: Hey, Jude
Johnさんのひとこと「Hey, Jude」
hanakoさんのひとこと「やほー」
taroさんのひとこと「hoge」
```

◇CGIでの利用

word-pstore.rbをCGIにしたものを示す。次節で述べるが、このCGIを保存するディレクトリは別に作成する必要がある。

word-pstore-cgi.rb

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

myname="word-pstore-cgi.rb"
require "cgi"
c = CGI.new(:accept_charset => "UTF-8")
require "pstore"
x = PStore.new("data/word.db")  # 別ディレクトリにする

print "Content-type: text/html; charset=UTF-8\n\n"

print "<!DOCTYPE html>
<html>
<head><title>Word</title></head>
<body>"

# 値入力フォームもこのCGIで出力する。
```

```
# formのactionをこのCGIプログラムに指定している。
# (mynameはこのスクリプト名)
printf("<form method=\"POST\" action=\"./%s\">\n", myname)
print ' <p>
おなまえ: <input name="name" maxlength="40"><br>
ひとこと: <input name="word" maxlength="80"><br>
<input type="submit" value="GO">
<input type="reset" value="reset">
</p></form>'

x.transaction do
  x["word"] ||= Hash.new
  word = x["word"]
  if c["name"] > "" && c["word"] > ""
    name = c["name"]
    word[name] = c["word"]
  end
  print "<pre>\n"
  for p, w in word
    # フォーム入力値を出力するときは必ず CGI.escapeHTML() する
    person = CGI.escapeHTML(p)
    wrd = CGI.escapeHTML(w)
    printf("%sさんのひとこと「%s」\n", person, wrd)
  end
  print "</pre>"
end
puts "</body></html>"
```

◇データを保存するCGIの実行

CGIプログラムは自分のユーザ権限で動かない。Webサーバプログラムは違うユーザ権限で動いているため、自分の保有するディレクトリに書き込むことがそのままではできない。CGIプログラムにデータファイルを書かせるためには当該ディレクトリを他人でも書き込めるように設定しておかなければならない。

データを書き込むディレクトリを別途作成する。作成中の CGI プログラムが ~/public_html/mycgi であると仮定するとその中に data/ ディレクトリを作成する。

```
% mkdir ~/public_html/mycgi/data
% chmod 1777 ~/public_html/mycgi/data
```

chmodの 1777 は、誰にでも書き込みできるが既存ファイルを消すのはファイル所有者のみ、という属性をもつディレクトリに設定することを意味する。 ~/public_html/mycgi/ に word-pstore-cgi.rb を保存し chmod +x しておく。

実際に動かす例を word-pstore-cgi に示す。

◇実用CGIスクリプトへ

■自分用データベース

PStoreクラスを利用して、Webで入力できる自分用のデータベースを作ろう。

ここでは、何かの飲み物を飲んだときの感想を登録できるデータベースを作る。入力名として、

飲み物の名前	item
感想	comment

を利用する。また、入力フォームの出力と、現在登録されているデータの出力を両方とも行なうようなCGIプログラムとする。大まかな構成は以下のようになる。

1. HTMLヘッダ等の出力
2. データ入力用フォームの出力
3. その時点で何か値が入力されていたらそれをハッシュに追加
4. 既存データの値一覧を出力

これをプログラム化した例が cancoffee.rb である。

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

require "cgi"
require "pstore"
myname = "cancoffee.rb"

c = CGI.new(:accept_charset => "UTF-8")
item = c["item"]
cmt  = c["comment"]
time  = Time.now                # 時刻を保持するTimeクラス代入。nowは現時刻

print 'Content-type: text/html; charset=UTF-8

<!DOCTYPE html>
<html>
<head><title>飲んだものメモ</title>
<link rel="stylesheet" type="text/css" href="simple.css">
</head>
<body>
<h1>飲み物メモ</h1>
'                               # HTTPヘッダと冒頭部分
db = PStore.new("data/coffee.db")
db.transaction do               # PStoreは db.transaction do ... end で使う
  db["root"] ||= Hash.new
  data = db["root"]             # ここまではおきまり

  if item >"" && cmt > ""       # 名前とコメント、両方値があるなら登録
    data[item] = [time, cmt]    # 今日の日付とコメント
  end

  # フォーム出力
  printf("<form method=\"POST\" action=\"./%s\">\n", myname)
  print '<p>
飲んだもの: <input name="item" type="text" maxlength="40"><br>
コメント <br>
<textarea name="comment" cols="40" rows="5">
</textarea><br>
<input type="submit" value="OK">
<input type="reset" value="reset"><br>
</p><hr>'

  # 既存のコメント出力(キー毎)
  print "<dl>\n"                # 定義環境開始
  for i in data.keys.sort{|x, y|
      data[y][0] <=> data[x][0] # 日付の新しい順にソート
```

```
    }
    day = data[i][0]            # 第0要素が日付
    msg = data[i][1]            # 第1要素がコメント、それぞれ取り出す
    printf(" <dt>%s</dt>\n", i) # キー(つまり飲んだものの名前)
    printf(" <dd>記載日: %s<br>\n", day.strftime("%Y年%m月%d日"))
    printf("     %s</dd>\n", CGI.escapeHTML(msg))
  end
  print "</dl>\n"               # 定義環境終了
end                             # db.transaction 終わり

print "</form><hr></body>\n</html>"
```

実際に登録してみよう。

■項目修正が容易なデータベース

cancoffee.rb では既存項目を入力すると、古い項目のコメントが消され、新しく入力されたコメントに置き換えられる。これにより後で修正することは可能だが、入力ミスしたものなどを直す場合、 既存の項目の値を初期値として代入した状態 で入力フォームが出力されていると都合がよい。そのように直したものを cc2.rb に示す。

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

require "cgi"
require "kconv"                 # utf-8への変換のため(toutf8メソッド)
require "pstore"
myname = "cc2.rb"

c = CGI.new(:accept_charset => "UTF-8")
item = c["item"]
cmt  = c["comment"]
time  = Time.now                # 時刻を保持するTimeクラス代入。nowは現時刻

edit_item = ARGV[0].to_s.toutf8 # 修正モードの場合の修正項目名
oldcomment = ""                 # 修正モードの場合の既存コメント

print 'Content-type: text/html; charset=UTF-8

<!DOCTYPE html>
<html>
<head><title>飲んだものメモ</title>
<link rel="stylesheet" type="text/css" href="simple.css">
</head>
<body>
<h1>飲み物メモ</h1>
'                               # HTTPヘッダ
db = PStore.new("data/coffee.db")
db.transaction do               # PStoreは db.transaction do ... end で使う
  db["root"] ||= Hash.new
  data = db["root"]             # ここまではおきまり

  if item >"" && cmt > ""       # 名前とコメント、両方値があるなら登録
    data[item] = [time, cmt]    # 今日の日付とコメント
  end

  if edit_item > "" && data[edit_item]
    oldcomment = data[edit_item][1]
  end
  # フォーム出力
```

```
  printf("<form method=\"POST\" action=\"./%s\">\n", myname)
  print '<p>
飲んだもの: <input name="item" type="text" value="' +
    edit_item + '" maxlength="40"><br>
コメント <br>
<textarea name="comment" cols="40" rows="5">' + oldcomment + '
</textarea><br>
<input type="submit" value="OK">
<input type="reset" value="reset"><br>
</p><hr>'

  # 既存のコメント出力(キー毎)
  print "<dl>\n"                # 定義環境開始
  for i in data.keys.sort{|x, y|
      data[y][0] <=> data[x][0] # 日付の新しい順にソート
    }
    day = data[i][0]            # 第0要素が日付
    msg = data[i][1]            # 第1要素がコメント、それぞれ取り出す
    # <a href="./cc2.rb?ITEM">ITEM</a> を出力
    printf(" <dt><a href=\"%s?%s\">%s</a></dt>\n", myname, i, i)
    printf(" <dd>記載日: %s<br>\n", day.strftime("%Y年%m月%d日"))
    printf("     %s</dd>\n", CGI.escapeHTML(msg))
  end
  print "</dl>\n"               # 定義環境終了
end                             # db.transaction 終わり

print "</form><hr></body>\n</html>"
```

◆CGIへの画像の投稿

blogサーバなど、文章だけでなく写真等のファイルをポストできるものがある。これにはHTMLフォーム input 要素の、type="file" を利用する。

◇multipart/form-data

文字列や選択オプションを送信するフォームはたとえば以下のように記述した。

```
<form method="POST" action="./hoge.rb">
 一言メモ: <input name="var" size="40"><br>
 <input type="submit" value="SEND!">
 <input type="reset" value="reset">
</form>
```

ファイルの中味をPOSTするには、form の enctype 属性を multipart/form-data に変える。

```
<form method="POST" enctype="multipart/form-data" action="./hoge.rb">
 一言メモ: <input name="var" size="40"><br><br>
 画像を選んでね: <input name="image" type="file">
 <input type="submit" value="SEND!">
 <input type="reset" value="reset">
</form>
```

HTMLで multipart/form-data を指定して入力させた値を受け取るCGIスクリプト、すなわちRubyプログラムは、CGI変数の受け取りを以下のように変更する。

```
require "cgi"
```

```
cgi = CGI.new(:accept_charset => "UTF-8")

memo = cgi["var"].read                  # CGI変数 var の受け取り(text)
photo_filename = cgi["image"].original_filename # ファイル名の受け取り
photo_size = cgi["image"].size          # 送信ファイルサイズ取得
photo_data = cgi["image"].read          # 送信ファイルデータの受け取り
```

このように、実際のCGIフォームでの送信値を得るには read メソッドを介して行なう。type="file" だけでなく type="text" も read メソッドで取得することに注意。

読み取った画像データをファイルに保存するにはたとえば以下のようにする。

```
open("ファイル名", "w") do |out|
  out.write cgi["image"].read
end
```

保存するファイル名は上書きに留意しつつ適切に決める。なお、ファイルを保存するディレクトリはWebサーバの権限で書き込めるようにしておく。そのため、画像データを書き出すディレクトリはあらかじめ決めて chmod 1777 しておくとよい。

◇画像投稿の実例

1MB以下の画像ファイル(JPEGまたはPNG)を投稿してみよ。

投稿画像選択: 参照... ファイルが選択されていません。
一言説明: 送信 reset

ソース: filepost.rb

◇複数画像の投稿

html5に対応したブラウザのみの利用となるが、type="file" の input 要素に multiple 属性を付けると、複数のファイル選択をさせることが可能となる。

この場合、Rubyプログラムで受け取るCGI変数は、値を配列で受け取れるように CGI#params を利用する。たとえばHTML側で、

```
<input name="memo" type="text">
<input name="memo" type="text">
```

のように同じCGI変数 memo で複数の値を送信することもできる。これをRubyプログラム側で処理するには

```
require 'cgi'
cgi = CGI.new(:accept_charset => "UTF-8")
v_memo = cgi.params["memo"]
```

のようにする。v_memo[0] で1つ目のinputフォーム値が、 v_memo[1] で2つ目のinputフォーム値が受け取れるようになる。これは他のtypeでも同様である。ただし、 enctype="multipart/form-data" の場合にはそれぞれ v_memo[0].read, v_memo[1].read のようにさらに read メソッドを呼んで実際のCGI変数値が取得できる。

以上のことを利用して、複数のファイルをまとめて投稿させ、それらのサムネイル画像を生成し、サムネイルから元の画像にリンクした Webページを自動生成するCGIプログラムの例を以下に示す。実際に複数の画像を与えてみよ。

投稿画像選択(複数画像OK): 参照... ファイルが選択されていません。
タイトル:
本文:
送信 reset

ソース: multipost.rb

◆CGI学習時の生成AI利用のコツ

Web上で動くアプリケーションは多種多様で、検索やGAIの結果もそれに応じて多様すぎる。普通に問い合わせても理解できないものが含まれていてそこからの応用が効かない。

ここで学習しているCGIではWebプログラムの基本原理を知るために、なるべく基本的な処理だけで済むものを利用しているので、GAIへの問い合わせにもそれを前提としたものを先に示すとよい。

◇問い合わせ例

以下の順番で問い合わせしてみるとよい。

1. 自己紹介する

   こんにちは、出席番号uso800の椎寺愛です。しいちゃんと呼んでね。

2. 前提知識と今の状況を伝える

   いま、RubyでCGIをならっています。ほとんどRubyの文法を知りません。
   これをベースに拡張したいです。
   《ここに btype.html などCGI授業の最初の方のプログラムを貼り付ける》

3. 結果が出るのでわからない点・不満な点を修正してもらう。

◇応用問い合わせ例

その他、いくつかの拡張をしたいときの前置き例を示す。

■データ保存系

「データを保存できるようにできますか」と普通に問い合わせて得られるプログラムを実行しても、ファイルに書き込みできないエラーが発生する。「データを永続させるCGI」の例をしっかり理解して、Webサーバプロセス(つまり他人)が*書き込みできる* ようにする。これを前提として質問に入れる。

> *（前に得られた答に続けて）*
> このプログラムに入力してくれたものを保存できるようにしたいです。
> 保存するファイルは
> mkdir -m 1777 db
> したディレクトリに入れるようにしてください。

得られた答にわからないものが含まれていたら以下のように聞き直す。

> *（前に得られた答に続けて）*
> PStoreを使うようにできますか。

■デザイン系

Web出力を見栄えのよいものにしたいとき、「画面をきれいにして」だけだとかなり複雑なCSSがあてがわれる。またCGIプログラム内に埋め込まれると、あとで修復が大変になる。シンプルなCSSを別ファイルに作ってもらうように問い合わせる。

> このプログラムの出力画面を少しおしゃれにしてください。
> CSSを後でいじりたいので、シンプルなCSSにして、ファイルは
> main.css でお願いします。

よい例を示してくれるはずだが、あとで自分で直せるように説明してもらう。

◇その他

GAIに全部生成してもらうとゴール到達まで却って時間がかかる。問題を細分化して質問して使い方を説明してもらって、それまで作ったプログラムに組み込むようにしたほうがよい。

また、問い合わせた答が、習っていないものにそれた場合は質問を続けず、その質問を編集し直して話題がそれないような一言を追加してやり直す。それた答が残ると、次の話題も外れたままになりやすい。

第15講　チーム課題作成

◆システムの設計

　公益に資する情報システムを作成する流れを俯瞰した上で地道に着手する。以下、いずれにおいても重要なことは *具体性* で、「困っている」とか「便利」とか曖昧なレベルを超えた発案をする。

◇係と仕事の分割

　個人個人、以下の4つのうちから「主担当」となるもの1つ、「副担当」となるもの1つを選ぶ。チーム内でなるべく人数の偏りがないようにする。

- プログラム作成
- データ作成(情報提示系) or ストーリー作成(体験提示系)
- 画面作成（HTML, 画像, 動画）
- 納品物作成(プレゼンシート + レジュメ + ポスター)
  納品物はいずれもPDF化してチームWebに置いてリンクする

係を決めるためにまず全員で要件定義を行なう。そののち、作成するものについて4つの仕事に分けて進める。

1. 要件定義(全員で)
2. 画面設計
3. 機能設計
4. データ設計
5. 材料構築

■テーマの選定

　まず題材を決める必要がある。題材については

- 既存のサービスに*ないもの*
- 既存のサービスでは*行き届かない部分*が明らかにあるもの
- *受益者層*がはっきりしているもの
- 自分達が集める/作ることが*できる*情報か(実行可能性/規模/権利関係)
- 具体的成果物を作る*動機*を高く持てるものか

などを勘案する。

■要件定義

- 対象は「誰」の「どういう状況」
    - 実例を「Aさん」、「Bさん」の2パターン以上の物語で書いてみる。
    - 「具体的」にするのはその問題に関係する部分のみ(本筋無関係のところの具体性は脱線の元なので控える)。
    - 脱線しても否定はせず元に戻す働きかけをする。
- 何を提示する? 「情報」or「体験」?
    - 「情報」の場合: 実例を5個以上挙げてみる。
    - 「体験」の場合: 動きを雑な絵に書いてみる。

以上が決まったら係ごとの作業に移る。

■画面設計

- 何を入力させて
- 何を出力するか
- どこを見やすくするか
- 何枚の画面になるか
- デッサンする

データ設計者から5個のサンプルデータをもらって作成する。

■機能設計

- 入力値をどう綺麗にするか(文字列か数値か)
- 出力の書式と文法の具体例
- 具体的にどう動かすか

データ設計者から5個のサンプルデータをもらって作成する。

■データ設計

- 必須項目は何か
- 綺麗な形で格納する具体例
- どうやって手分けするか具体的手順

5個のサンプルデータをまず作り他の係に渡す。

■材料

- 探す：オープンデータを探す
- 作る：自由に使えるツールを利用して作る(誰でも手軽に作れるようにする)

以後の作業が台無しにならないよう「著作権」に厳重注意して進める。

◇作業手順

1. テーマの選定(2つに絞る) *15分*
2. ストーリーの作成 *30分*
3. プロトタイプの作成
    - プログラム係：予備データで動くものを作る
    - 画面係：システムの入口と操作する入力用Webページ
    - データ係：予備データを1人20個以上
    - 納品物作成：他の係の手伝い

■本日の作業

上記の2番まで行なう。

■授業終了後の作業

本日の課題へ

練習問題解答

解答として示すプログラムは題意を満たすひとつの例である。同等の動きを示すプログラムはいくらでもあるので、多くの可能性を探し出して欲しい。

◆練習問題 2

1. hw.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
print "世界のみなさんこんにちは!\n"
print "世界のみなさんこんにちは!\n"
print "世界のみなさんこんにちは!\n"
```

print "〜〜\n" のかわりに puts "〜〜"でも良い。

2. 修正後のgatagata.rb

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

x = gets.chomp.to_i
if x > 0
  printf("%d は正の数ね\n", x)
end
```

Emacsで編集している場合は、全ての行で[TAB]（またはC-i)をタイプして揃える。

◆練習問題 3

1. イ、8

   ロ、8

   ハ、20

   ニ、"999"

2. イ、0回。ifの比較で 5 > 5となり不成立となるため。

   ロ、5回。xの値が 5, 4, 3, 2, 1 のときに内部を実行する。一般的に*N*回繰り返したい場合は

   ```
   x=N
   while x>0
    :
   end
   ```

   または、

   ```
   x=0
   while x<N
    :
   end
   ```

   とループ形成する。これらのやり方はwhile構文のある他の言語でも同じ。Rubyの場合はさら

に、

```
N.times do |i|
 :
end
```

や、

```
for i in 1..N
 :
end
```

などの便利な構文が豊富にある。

ハ、50回。0～99の範囲の偶数の個数だけ実行する。

3. イ、
```
if y%400 == 0 then
  puts "閏年"
end
```
ロ、
```
if y%4 != 0 then
  puts "閏年でない"
end
```
ハ、
```
if y%4 == 0 && y%100 != 0 then
  puts "閏年"
end
```
二、
```
if y%4 != 0 || y%100 == 0 && y%400 != 0 then
  puts "閏年でない"
end
```

◆練習問題 4

1. gomasuri.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
n = gets.to_i
i=0
while i < n
  puts "きみは天才!"
  i += 1
end
```

2. homegoroshi.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
while true
  puts "きみは天才!"
end
```

3. rectangle.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
STDERR.print "長方形の面積を求めます。\n"
STDERR.print "底辺は?(cm): "
width = gets.to_i
STDERR.print "高さは?(cm): "
height = gets.to_i
printf("面積は%d平方cmです\n", width*height)
```

STDERR.はなくても本質的間違いではない。以下のプログラムも同様。

4. jetcoaster.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
STDERR.print "恐れ入りますが身長をcmで入力してください: "
tall = gets.to_i
if tall >= 120 && tall <= 180 then
  print "搭乗できます\n"
else
  print "申し訳ありませんが搭乗できません\n"
end
```

この問題の場合は、比較で利用している不等号に= はあってもなくても構わない。

5. kingin.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
STDERR.print "現在の年齢は?: "
age = gets.to_i
STDERR.print "今年の西暦は?: "
year = gets.to_i
while age < 100
  age += 1
  year += 1
  printf("%d年であなたは%d歳です。\n", year, age)
end
```

6. shopping.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
total = 0
while true
  STDERR.print "商品名は?: "
  item = gets.chomp
  if item == "q" then
    break
  end
  STDERR.print "おいくら?: "
  price = gets.to_i
  printf("%-14s %5d円\n", item, price)
  total += price
end
print "お買い上げありがとうございました。\n"
printf("合計は%6d円です。\n", total)
```

◆練習問題 5

1. イ、/./ → 1, 2, 3, 4

   ロ、/e./ → 1, 2, 3, 4

   .は「任意の1字」なのでeの文字の後に何か文字があればその行は全てマッチする。

   ハ、/the/ → 3

   1行目のTheはTが違うのでマッチしない。

   ニ、/the/i → 1, 3

   大文字でもよくなるので今度は1行目のTheもマッチする。

ホ、/[A-Z]/ → 1, 2, 4

大文字のある行全てにマッチする。

ヘ、/^[A-Z]/ → 1

行頭が大文字の行にマッチする。

ト、/[^A-Z]/ → 1, 2, 3, 4

大文字以外のものは全ての行にある。

チ、/[a-z][A-Z]/ → 2

小文字のあとに大文字が来るのは2行目の OggVorbis のみ。

リ、/\d/ → 1, 2

ヌ、/\d*/ → 1, 2, 3, 4

*ならば直前の\dが0回でも構わない。よって全てにマッチする。

ル、/\d+/ → 1, 2

数字が1つ以上あれば良い。

ヲ、/\d*[A-Z]/ → 1, 2, 4

\d* は数字がなくても良いのだから大文字のある行全てにマッチする。

ワ、/\d.*[A-Z]/ → 1, 2

数字の見つかった場所より後に大文字があれば良い。

カ、/(.)\1/ → 2, 4

同じ文字が繰り返し現れるのは2行目のOggVorbis(のg)と 4行目のbuffer(のf)。

2. 実際に実行してみよ。

◆練習問題 6

1. pts-sum.rb

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

pts = [198, 37, 66, 28, 33, 45, 46, 32, 2, 99]
sum=0
for x in pts
  sum += x
end
printf("%d\n", sum)
```

2. pts-avg.rb

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

pts = [198, 37, 66, 28, 33, 45, 46, 32, 2, 99]
sum=0
for x in pts
  sum += x
end
printf("%f\n", sum.to_f/pts.length)
```

整数だけなのでそのまま計算すると割り算も整数化される。 to_f で浮動小数点数に変換してから割り算する。

3. pts-minmax.rb

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

pts = [198, 37, 66, 28, 33, 45, 46, 32, 2, 99]
sum=0
min = max = pts[0]
for x in pts
  sum += x
  if min > x then
    min = x
  elsif max < x then
    max = x
  end
end
printf("平均値 %f\n", sum.to_f/pts.length)
printf("最大値 %d\n", max)
printf("最小値 %d\n", min)
```

4. tac.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
buffer = []
n=0
while str = gets
  buffer[n] = str
  n += 1
end
for l in buffer.reverse
  print l
end
```

もっとも短く書くと

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
print readlines.reverse
```

となる。

5. register.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
name = []                       # 商品名を配列にいれる
value = []                      # 価格を配列にいれる
total = 0                       # 価格の合計
nitem = 0                       # 読み取った商品の数
while true
  STDERR.print "商品名は?: "
  item = gets
  if item == nil then           # 入力終わりなら gets はnilを返す
    break
  end
  name[nitem] = item.chomp!
  STDERR.print "おいくら?: "
  price = gets.to_i
  value[nitem] = price
  total += price
  nitem += 1
end
```

```
i=0
puts ""                          # 改行するだけ
print "お買い上げありがとうございました。\n"
while i < nitem
  printf("%-14s %5d円\n", name[i], value[i])
  i += 1
end
puts "---------------------------"
printf("%-13s %6d円です。\n", "合計は", total)
```

◆練習問題 7

1. fcat.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
STDERR.print "見たいファイルは?: "
fn = gets.chomp
open(fn, "r") do |f|
  while fline=f.gets
    print fline
  end
end
```

2. fcat2.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
STDERR.print "見たいファイルは?: "
fn = gets.chomp
if test(?r, fn) then
  open(fn, "r") do |f|
    while fline=f.gets
      print fline
    end
  end
else
  STDERR.print "そんなファイル読めないよ\n"
end
```

3. fcat3.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
STDERR.print "見たいファイルは?: "
fn = gets.chomp
begin
  open(fn, "r") do |f|
    while fline=f.gets
      print fline
    end
  end
rescue
  STDERR.print "そんなファイル読めないよ\n"
end
```

4. sushikuine.rb

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

sum = 0
```

```
menu = []
price = []
i = 0

open("price.txt", "r") do |neta| # price.txtを読み込みモードで開く
  while line = neta.gets         # ファイルから1行ずつ読む
    if /(\S+)\s+(\d+)円/ =~ line # (数字)円 というパターンがあれば
      menu[i]  = $1              # $1 は商品名
      price[i] = $2.to_i         # $2 が「数字」に当てはまるので整数化
    end
    i += 1
  end
end

while true
  j=0
  while j<i
    # 商品番号と、ネタ名、価格を出力
    printf("%3d %s - %d\n", j, menu[j], price[j])
    j+=1
  end

  print "御注文は(番号で入れてね、q で終了)?: "
  line=gets.chomp!              # ユーザの入力を line に入れる
  if line == "q"                # "q" を入力したら
    break                       #    終了
  end
  number = line.to_i            # 整数(メニュー番号)にする
  if number < 0 || number >= i
    puts "そんなメニューねぇでガス"
    redo
  end
  printf("はぁ〜い、「%s」一丁\n", menu[number])
  sum += price[number]
end

print "おあいそでガス\n"
printf("全部で %d 円でガス。まいどっ\n", sum)
```

5. parking.rb

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

begin
  open("parking.tmpl", "r") do |pr|
    open("parking.txt", "w") do |pw|
      while pl=pr.gets
        if /%DATE%/ =~ pl then
          pw.puts Time.now.strftime("%Y年%m月%d日")
        else
          pw.print pl
        end
      end
    end
  end
  STDERR.puts "本日分の駐車券を作成しました。"
  STDERR.puts "parking.txtを印刷して利用してください。"
rescue
  STDERR.puts "parking.tmplを用意したうえで、。"
  STDERR.puts "parking.txtに結果が書き出せるようにしておいてください。"
  STDERR.puts "よろしう。"
end
```

6. customreg.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
name = []                       # 商品名を配列にいれる
value = []                      # 価格を配列にいれる
total = 0                       # 価格の合計
nitem = 0                       # 読み取った商品の数

begin
  # テンプレートファイルを開く
  open("register.tmpl", "r") do |tp|
    while true
      STDERR.print "商品名は?: "
      item = gets
      if item == nil then               # 入力終わりなら gets はnilを返す
        break
      end
      name[nitem] = item.chomp!
      STDERR.print "おいくら?: "
      price = gets.to_i
      value[nitem] = price
      total += price
      nitem += 1
    end

    i=0
    puts ""                             # 改行するだけ
    while template=tp.gets
      if /%SUMMARY%/ =~ template
        # テンプレートファイルに %SUMMARY% という行があったら
        # その部分の代わりに買い物情報を出力
        while i<nitem
          printf("%-14s %5d円\n", name[i], value[i])
          i += 1
        end
        puts "- "*17
        print "お買い上げありがとうございました。\n"
        printf("合計は%6d円です。\n", total)
      elsif /%DATE%/ =~ template then
        puts Time.now.strftime("%Y年%m月%d日")
      else
        print template
      end
    end
  end
rescue
  STDERR.puts "register.tmplを用意してからプログラムを起動してください。"
end
```

◆練習問題 8

1.

10進数	2進数	16進数	7進数
2006	11111010101	7d5	5563
42	101010	2a	60
3054	101111101110	bee	11622
194	11000010	c2	365

2.

J	a	p	a	n	!
0x4a	0x61	0x70	0x61	0x6e	0x21

◆練習問題 9

1. last1c.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
while l=gets
  print l[-2..-1]
end
```

2. hyphen.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
while l=gets
  l.chomp!
  print l+"-"*(80-l.length)+"\n"
end
```

3. randq.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
m = ["世界で一番高い山は?", "日本で一番高い山は?",
  "エトセトラ、を表す3文字の略語は?",
  "高速道路で料金を自動的に払うシステムの略称は?"]
srand()
puts m[rand(m.length)]
```

4. qpod.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
question=[]
answer=[]
n = 0
begin
  open("quiz.dat", "r") do |q|
    while line=q.gets
      question[n] = line.chomp
      answer[n] = q.gets.chomp
      n+=1
    end
  end
rescue
  STDERR.puts "quiz.datに問題&解答集を用意してね"
  exit(0)
end

i=0
while i<n
  printf("問題: %s(答を見るには[Return])\n", question[i])
  gets                          # [Return]で先に進む
  printf("答: %s\n", answer[i])
  i+=1
end
```

5. shuffle.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
x = (0..52).collect
i=0
srand()                         # 乱数の種を初期化
while i<x.length
```

```
  t=rand(x.length)              # 交換相手を探す
  tmp=x[i]
  x[i] = x[t]
  x[t] = tmp
  i += 1
end
p x                             # xを配列のまま出力
```

6. qpod-shuffle.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
question=[]
answer=[]
n = 0
begin
  open("quiz.dat", "r") do |q|
    while line=q.gets
      question[n] = line.chomp
      answer[n] = q.gets.chomp
      n+=1
    end
  end
rescue
  STDERR.puts "quiz.datに問題&解答集を用意してね"
  exit(0)
end

# シャッフルする
srand()
i=0
while i<question.length
  t=rand(question.length)
  # まず、question[i]とquestion[t]を交換
  tmp=question[i]
  question[i] = question[t]
  question[t] = tmp
  # answer[i]とanswer[t]を交換
  tmp=answer[i]
  answer[i] = answer[t]
  answer[t] = tmp
  i+=1
end

i=0
while i<n
  printf("問題: %s(答を見るには[Return])\n", question[i])
  gets                          # [Return]で先に進む
  printf("答: %s\n", answer[i])
  i+=1
end
```

7. janken.rb

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

janken = ["グー", "チョキ", "パー"]
while true
  STDERR.puts "じゃんけんしましょ、"
  STDERR.print "あなたの手は? [グーなら0、チョキなら1、パーなら2]: "
  j = gets
  if j == nil then
    break
  end
  human=j.to_i
```

```
  if human < 0 || human > 2 then
    STDERR.puts "ちゃんと出してね。0, 1, 2のどれかだよ。"
    redo
  end
  com = rand(3)
  printf("あなた「%s」 :  「%s」わたし\n",
         janken[human], janken[com])
  if human==0 && com==1 || human==1 && com==2 || human==2 && com==0 then
    puts "があ〜ん負けちゃったー! 強いのね。"
  elsif human==com then
    puts "おっと引き分けだ。仲良しこよし。"
  else
    puts "やったー、勝ったー! よわよわじゃん。"
  end
  STDERR.print "もういちど"
end
STDERR.puts "\nあらもうおしまい? さようなら!"
```

8. jankenstat.rb

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

win=lose=draw=0                 # 勝ち、負け、引き分け
janken = ["グー", "チョキ", "パー"]
while true
  STDERR.puts "じゃんけんしましょ、"
  STDERR.print "あなたの手は?"
  STDERR.print "[グーなら0、チョキなら1、パーなら2]: "
  j = gets
  if j == nil then
    break
  end
  human=j.to_i
  if human < 0 || human > 2 then
    STDERR.puts "ちゃんと出してね。0, 1, 2のどれかだよ。"
    redo
  end
  com = rand(3)
  printf("あなた「%s」 :  「%s」わたし\n",
         janken[human], janken[com])
  if human==0 && com==1 ||
      human==1 && com==2 ||
      human==2 && com==0 then
    puts "があ〜ん負けちゃったー! 強いのね。"
    win += 1
  elsif human==com then
    puts "おっと引き分けだ。仲良しこよし。"
    draw += 1
  else
    puts "やったー、勝ったー! よわよわじゃん。"
    lose += 1
  end
  STDERR.print "もういちど"
end
STDERR.puts "\nあらもうおしまい? さようなら!"
printf("あなたの戦績: %d勝%d敗%d分け\n",
       win, lose, draw)
```

目次へ

◆練習問題 10

1. 数学的関数*f(x)*に相当するメソッド。

```
def f(x)
  x**2+3*x-1
end
```

2. rectangleメソッド。

```
def rectangle(width, height)
  width*height
end
```

3. leapメソッド。

```
def leap(y)
  if y%4!=0 || y%100==0 && y%400!=0 then
    365
  else
    366
  end
end
```

このロジックは、閏年判定に用いられる。

4. tagifyメソッド。

```
def tagify(element, content)
  "<"+element+">"+content+"</"+element+">"
end
```

5. pre.rb

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

def tagify(element, content)
  "<"+element+">"+content+"</"+element+">"
end

while sentence=gets
  printf("%s\n", tagify("pre", sentence.chomp!))
end
```

6. sflメソッド。

```
def sfl(arr)
  tmp=arr[0]
  arr[0]=arr[-1]
  arr[-1]=tmp
  arr
end
```

◆練習問題 11

1. プログラムの前半と後半に分けて例を載せる。前半にはエラー処理を記述する。

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

if ARGV[0] == nil then
  STDERR.puts "出力回数を指定してください。"
  STDERR.puts "例: ./nhello.rb 5"
  exit(1)
end
```

後半は、ARGV[0]を整数に直して(to_i)、その回数だけ puts "Hello, world!" する。様々な方法

があるので以下のその1〜その5のどれかを選択する。

```
# その1
ARGV[0].to_i.times do
  puts "Hellow, wowrld!"
end
# その2
ARGV[0].to_i.downto(1) do |i|
  puts "Hellow, wowrld!"
end
# その3
1.upto(ARGV[0].to_i) do |i|
  puts "Hellow, wowrld!"
end
# その4
for i in 1..(ARGV[0].to_i)
  puts "Hellow, wowrld!"
end
# その5
1.step(ARGV[0].to_i, 1) do |i|
  puts "Hellow, wowrld!"
end
```

2. copy.rb

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

if ARGV[1] == nil
  STDERR.puts "ファイルをコピーします。"
  STDERR.puts "./copy.rb コピー元 コピー先"
  exit(1)
end

# ARGV[0]がコピー元ファイル。読めるかチェック。
if test(?r, ARGV[0]) then
  open(ARGV[0], "r") do |src|
    open(ARGV[1], "w") do |dst|
      while s=src.gets
        dst.print s
      end
    end
  end
end
```

3. backup.rb

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

if ARGV[0] == nil
  STDERR.puts "ファイルをバックアップします。"
  STDERR.puts "./backup.rb ファイル名"
  exit(1)
end

# ***** (A) *****
# ARGV[0]がコピー元ファイル。読めるかチェック。
if !test(?r, ARGV[0]) then
  STDERR.printf("%s ファイルは読めません\n", ARGV[0])
  exit(2)
end
n=1                     # 拡張子。1からスタート。
while true
  # 元の名前とnからバックアップファイル名を生成
  backup = sprintf("%s.%d", ARGV[0], n)
```

```
  if !test(?r, backup)  # 存在しなければ
    break               # ループを抜ける
  end
  n += 1
end

open(ARGV[0], "r") do |src|
  open(backup, "w") do |dst|
    while s=src.gets
      dst.print s
    end
  end
end
STDERR.printf("[%s] --> [%s]\n", ARGV[0], backup)
# ***** (B) *****
```

4. 前問のbackup.rbのソース中にある「***** (A) *****」の部分に

```
while ARGV[0]
```

を、「***** (B) *****」の部分に

```
  ARGV.shift
end
```

を追加すれば良い。(A)と(B)で囲まれた部分は、適宜インデントし直すこと。

◆練習問題 12

1. pricelist.rb

```
#!/usr/local/bin/ruby
# coding: utf-8
prlist={}
while entry=gets
  if /(\S+)=(\d+)/ =~ entry
    prlist[$1] = $2.to_i
  end
end

p prlist
```

2. aefreq.rb

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

freq=Hash.new(0)

while s=gets
  for ch in s.chomp!.split(//)
    freq[ch] += 1
  end
end

puts "---+--------"
for ch, count in freq do
  if /[a-e]/ =~ ch
    printf(" %s | %3d\n", ch, count)
 end
end
puts "---+--------"
```

◆練習問題 13

1. kokugo-rank.rb のうち、

```
for student in point.keys.sort {|x, y|
    point[x][0] <=> point[y][0]
  }.reverse
```

の部分を、以下のように変えれば良い。

```
for student in point.keys.sort {|x, y|
    point[x][1] <=> point[y][1]
  }.reverse
```

2. 前問の該当部分を以下のように変えれば良い。

```
for student in point.keys.sort {|x, y|
    point[x][0]+point[x][1] <=> point[y][0]+point[y][1]
  }.reverse
```

ただし、この場合sortメソッドが並べ換えのときに何度も足し算することになり効率が良くない。

このような場合は、データの読み込み時にあらかじめ合計値を計算し、配列要素として追加しておくのがよい。そうするためには、kokugo-rank.rbの

```
if /(\S+)\s+(\d+)\s+(\d+)/ =~ yline
    point[$1] = [$2.to_i, $3.to_i]
end
```

の部分を、

```
if /(\S+)\s+(\d+)\s+(\d+)/ =~ yline
    point[$1] = [$2.to_i, $3.to_i, $2.to_i+$3.to_i]
end
```

と変更し、並べ換えの部分を

```
for student in point.keys.sort {|x, y|
    point[x][2] <=> point[y][2]
  }.reverse
```

とすればよい。

3. jslash.rb

```
#!/usr/local/bin/ruby
# coding: utf-8
line=[]
while l=gets
  line << l.chomp!
end
puts line.join("/")
```

4. wordsplit.rb

```
#!/usr/local/bin/ruby
# coding: utf-8
while l=gets
  l.chomp!
  for word in l.split(/[., \t\n]+/)
    if word != "" then
      puts word
    end
  end
end
```

5. freq.rb

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

freq=Hash.new(0)

while s=gets
  for ch in s.chomp!.split(//)
    freq[ch] += 1
  end
end

puts "---+--------"
for ch, in freq.keys.sort{|x, y| freq[y] <=> freq[x]}
  if /[a-z]/ =~ ch
    printf(" %s | %3d\n", ch, freq[ch])
  end
end
puts "---+--------"
```

6. wordfreq.rb

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

freq=Hash.new(0)

while l = gets
  for word in l.chomp!.split(/[,. \t]+/)
    if word != ""
      freq[word] += 1
    end
  end
end

puts "--------------+--------"
for word in freq.keys.sort{|x, y| freq[y] <=> freq[x]}
  if /[a-z]/ =~ word
    printf(" %-12s | %3d\n", word, freq[word])
  end
end
puts "--------------+--------"
```

◆練習問題 14

1. age.rb

```
#!/usr/local/bin/ruby
# coding: utf-8
def age(thisyear, birthyear)
  if thisyear==birthyear
    0
  else
    1+age(thisyear-1, birthyear)
  end
end

STDERR.print "生まれた西暦年は?: "
printf("誕生日後は%d歳ですね\n",
       age(Time.now.year, gets.to_i))
```

2. hanoi.rb

```
#!/usr/local/bin/ruby
```

```
# coding: utf-8

def hanoi(n, from, to, work)
  if n == 1
    printf("%d番の円盤 を %s から %s に移動しなさい\n", n, from, to)
  else
    hanoi(n-1, from, work, to)
    printf("%d番の円盤 を %s から %s に移動しなさい\n", n, from, to)
    hanoi(n-1, work, to, from)
  end
end

if ARGV[0] == nil
  STDERR.puts "ハノイの塔の最初の板の数を指定してください。"
  STDERR.puts "例: ./hanoi.rb 5"
  exit 0
end

n=ARGV[0].to_i
hanoi(n, "A", "B", "C")
```

3. afind.rb

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

if ARGV[0] == nil
  ARGV.push(".")
end

def afind(path)
  # [2..-1]で0番目、1番目の要素を除去した配列が得られる
  for f in Dir.entries(path)[2..-1].sort
    if test(?d, path+"/"+f)
      # fがディレクトリなら再帰的に検索
      afind(path+"/"+f)
    end
    if /^a/ =~ f then
      printf("%s/%s\n", path, f)
    end
  end
end

for dir in ARGV
  afind(dir)    # コマンドライン引数全部に対して繰り返す
end
```

◆練習問題 15

1. 13.6のwordfreq.rbの

```
l.chomp!
```

の部分を、

```
l.chomp!.downcase
```

に変えれば良い。

2. msg1.rb

```
#!/usr/local/bin/ruby
# coding: utf-8
ff="msg1.txt"
```

```
if !test(?r, ff)
  STDERR.printf("%s にメッセージを入れといてね\n", ff)
  exit(1)
end

STDERR.print "戦士よ、汝の名は?: "
name=gets.chomp

open(ff, "r") do |m|
  while msg=m.gets
    print msg.gsub(/%NAME%/){name}
  end
end
```

3. pos.rb

```
#!/usr/local/bin/ruby
# coding: utf-8
name = []                       # 商品名を配列にいれる
value = []                      # 価格を配列にいれる
total = 0                       # 価格の合計
nitem = 0                       # 読み取った商品の数


STDERR.print "ポイント倍率は?: "
pointrate=gets.chomp

begin
  # テンプレートファイルを開く
  open("register.tmpl", "r") do |tp|
    while true
      STDERR.print "商品名は?: "
      item = gets
      if item == nil then               # 入力終わりなら gets はnilを返す
        break
      end
      name[nitem] = item.chomp!
      STDERR.print "おいくら?: "
      price = gets.to_i
      value[nitem] = price
      total += price
      nitem += 1
    end

    i=0
    puts ""                             # 改行するだけ
    while template=tp.gets
      if /%SUMMARY%/ =~ template
        # テンプレートファイルに %SUMMARY% という行があったら
        # その部分の代わりに買い物情報を出力
        while i<nitem
          printf("%-14s %5d円\n", name[i], value[i])
          i += 1
        end
        puts "- "*17
        print "お買い上げありがとうございました。\n"
        printf("合計は%6d円です。\n", total)
      else
        template.gsub!(/%DATE%/){
          Time.now.strftime("%Y年%m月%d日")
        }
        template.gsub!(/%P%/){pointrate}
        print template
      end
    end
  end
rescue
  STDERR.puts "register.tmplを用意してからプログラムを起動してください。"
end
```

customreg.rbとの違いを下線 で示した。

4. exam.rb

```
#!/usr/local/bin/ruby
```

```
# coding: utf-8

pts={}
summary = gets.chomp.split(/,/) # 科目名リスト
summary << "合計"        # 科目名の末尾を「合計」に
nsubj = summary.length-1 # 科目数はsummaryの要素数-1

while line=gets
  x = line.split(/,/)   # x[1]～x[-1]に各科目の点が入る(文字列)
  name = x[0]           # x[0]には氏名
  pts[name] = x[1..-1]  # pts[name]にx[1]～x[-1]をコピー
  sum = 0               # 合計点用
  0.upto(pts[name].length-1) do |j|
    # 数値に変換しておく
    pts[name][j] = pts[name][j].to_i
    sum += pts[name][j]
  end
  pts[name] << sum      # 合計点を配列末尾に追加
end

printf(" %-14s ", summary[0])
for s in summary[1..-1]
  printf("| %4s ", s)   # 科目名のヘッダ出力
end
puts ""
puts "-"*16+"+------"*nsubj     # 区切り線
# 合計点の高い順にソート
for i in pts.keys.sort{
    |x, y| pts[y][-1]<=>pts[x][-1]
  }
  printf(" %-14s ", i)
  for p in pts[i]
    printf("| %4d ", p)
  end
  puts ""
end
puts "-"*16+"+------"*nsubj     # 区切り線
```

5. exam.rbの末尾に以下の内容を追加すれば良い。

```
# 各科目の平均を計算
sum=Array.new
avg=Array.new
0.upto(nsubj-1) do |i|
  sum[i]=0.0
  # 人数分可算
  for student in pts.keys
    sum[i] += pts[student][i]
  end
  avg[i] = sum[i]/pts.length
end

stddev=Array.new
# 各科目の標準偏差を計算
0.upto(nsubj-1) do |i|
  v=0.0         # 分散の計算
  for student in pts.keys
    v += (pts[student][i]-avg[i])**2
  end
  # (標本数-1)で割ったものの平方根が標準偏差
  stddev[i] = Math.sqrt(v/(pts.length-1))
end

printf(" %-14s ", "平均")
for p in avg
```

```
  printf("|%5.1f ", p)
end
puts ""
printf(" %-14s ", "標準偏差")
for p in stddev
  printf("|%5.1f ", p)
end
puts ""
puts "-"*16+"+------"*nsubj     # 区切り線
```

◆練習問題 16

1. 5pan.rb

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

menu=["あんパン", "クリームパン", "メロンパン",
"うぐいすパン", "ずんだパン"]

srand
0.upto(menu.length-1) do |i|
  j = rand(menu.length)
  tmp = menu[i]
  menu[i] = menu[j]
  menu[j] = tmp
end

puts "今日は、"
print menu.join("を食べて、\n")
puts "を食べてしめましょう。"
```

2. その都度ogg123コマンドを呼ぶ場合。

ogg-shuffle.rb

```
#!/usr/local/bin/ruby
# coding: utf-8
srand
0.upto(ARGV.length-1) do |i|
  j = rand(ARGV.length)
  tmp = ARGV[i]
  ARGV[i] = ARGV[j]
  ARGV[j] = tmp
end

for o in ARGV
  system("ogg123", o)
end
```

ogg123コマンドにoggファイルを全て渡す場合。

ogg-shuffle.rb

```
#!/usr/local/bin/ruby
# coding: utf-8
srand
0.upto(ARGV.length-1) do |i|
  j = rand(ARGV.length)
  tmp = ARGV[i]
  ARGV[i] = ARGV[j]
  ARGV[j] = tmp
end
```

```
system("ogg123 "+ARGV.join(" "))
```

付録

◆本書に関する情報

本書に関する最新情報、練習問題の解答等

> http://www.gentei.org/~yuuji/support/lcr/

から得られるようにしてある。

◆各種ツールの設定情報

Unix、並びにそれと同等の環境があることを前提として各種ツールの導入・設定方法を簡潔に示しておく。

◇Rubyのインストール

どんなシステムでもRuby自身のソースプログラムを持って来てコンパイルする手順は共通で進めることができる。

1. Rubyのソースアーカイブの入手

   http://www.ruby-lang.org/に、Ruby本体のダウンロードページへのリンクがあるのでそこに進み、

   ruby-*バージョン番号*.tar.gz

   を入手し、適当なディレクトリにコピーする。

2. Rubyのコンパイルとインストール

   以下の手順でコンパイルとインストールを行なう。% 記号は一般ユーザでの作業を、#記号はスーパーユーザでの作業を意味する。

```
% gzip -dc ruby-バージョン番号.tar.gz|tar xpf -
% cd ruby-バージョン番号
% ./configure && make
(ここでスーパーユーザになる)
# make install
```

一度ログアウトして入り直せばrubyコマンドが使えるはずである。

```
% ruby -v
ruby 1.8.2 (2004-10-25) [i686-netbsd2]
```

make作業をしたディレクトリで

```
% find . -name ruby-mode.el
```

するとEmacsでRubyプログラムを編集するモードの設定ファイルが見つかる。このファイルをEmacs用の

site-lisp ディレクトリにコピーする。 site-lisp ディレクトリは通常、emacsコマンドのあるディレクトリからみて、../share/emacs/site-lisp に存在する。たとえば、 emacsコマンドが /usr/local/bin にあるなら、 site-lisp ディレクトリは /usr/local/share/emacs/site-lisp になる。

◇Emacsの設定

Emacsの場合は ~/.emacs ファイルに

```
(set-language-environment "Japanese")
(setq-default buffer-file-coding-system 'japanese-iso-8bit)
(setq auto-mode-alist
      (cons (cons "\.rb$" 'ruby-mode) auto-mode-alist))
(autoload 'ruby-mode "ruby-mode" "Ruby Editing Mode" t)
```

と書いておけばとくに指定のない限り保存漢字コードが日本語EUCとなり、 Rubyプログラム編集時に自動的に ruby-mode が有効となる。

◇ターミナルソフトの設定

KTermの場合のみ示す。日本語EUCコードの利用を前提とする場合は、 ~/.Xdefaultsに以下の行を追加する(※)。

```
KTerm*KanjiMode:                euc
```

※ xrdbを利用している場合はさらに
xrdb ~/.Xdefaultsする必要がある。

目次へ