Rubyで始めるプログラミング

久木田　水生

2011年後期 情報科学演習

1　プログラミングの要素

この節では単純な例を挙げて，プログラミングにおいて本質的な要素を見ていこう．

任意の正整数はその素因数に一意的に分解できる．そこで，任意の正整数を素因数に分解する手続きを以下のように定義しよう．

アルゴリズム **

素因数分解

入力： 正整数 n

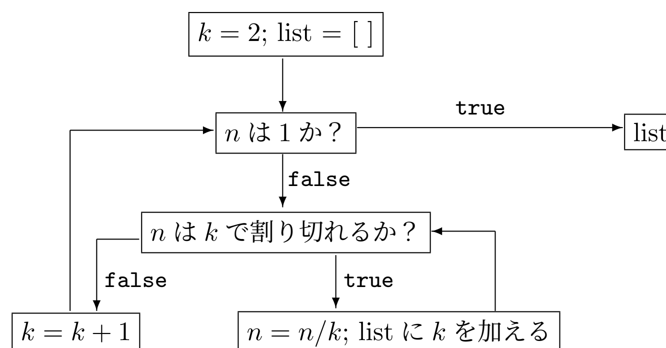
出力： n の素因数からなる配列 list

手続き：

最初に k を 2，list を空の配列としておく．n が 1 になるまで次の作業を繰り返し，n が 1 になったら list を出力する．n が k で割り切れるときは k をリストに加え，n の値を n/k に更新する．n が k で割り切れないときは，k の値を $k+1$ に更新する．

****

この手続きは以下のフローチャートによって表現される．



以上の手続きを `prime_factors` と呼ぶことにする．たとえば `prime_factors(126)` は配列 `[2, 3, 3, 7]` を出力する．

この例には以下の要素が含まれている．

- 整数
- 整数からなる配列
- 整数や配列を表す変数の導入，変数の参照，変数の値の更新
- ある整数が別の整数で割り切れるかどうかを調べる手続き
- 整数に 1 を加える手続き
- 配列に要素を加える手続き
- ある条件に従った作業の流れの分岐
- ある作業を繰り返し
- 手続きの定義と使用

以下でこれらについて説明しよう．

1.1 オブジェクト

オブジェクトとは，プログラミングにおいて扱われる対象，「もの」のことである．プログラムは通常，特定の個別的なオブジェクトではなく，何らか種類に属する任意のオブジェクトを扱っている．オブジェクトの種類のことを私たちは**クラス**，**型**，あるいは**タイプ**と呼ぶ．

私たちはアルゴリズムの各ステップにおいて，どの型のオブジェクトを扱っているのかということに関して自覚的でなければならない．というのも異なる型のオブジェクトに対しては異なる取り扱いが必要だからである．

上の例では整数と配列という型のオブジェクトが扱われた．プログラミングで扱われるオブジェクトには他に，小数，文字列，真偽値などがある[*1]．

文字列については 2 節，整数については 3 節，配列について 6 節，真偽値については 4 節で，より詳しく扱う．また新しい型のオブジェクトを定義する方法は 8 節で扱う．

1.2 変数

特定のオブジェクトに言及するために使われるのが**変数**である．これはオブジェクトにつけられた名前であると言ってもよい．プログラミングにおいてはすべてのことが言語によって表現されるため，あらゆるオブジェクトの取り扱いは，変数を通じて行われる．変数については 2.2 節で扱う．

[*1] 明示的にされていないが，実際には上の例で真偽値型のオブジェクトも使われている．

1.3　メソッド，手続き

オブジェクトに対して施される操作を**メソッド**（method），あるいは**手続き**（procedure）という．ある型のオブジェクトに対して適用できるメソッドは決まっている．その型のオブジェクトに適用できないメソッドを適用しようとするとエラーが起こる．

メソッドは，それを適切な型のオブジェクトに適用した結果として，何らかの出力を返すものとそうでないものがある．メソッドが出力する値を，そのメソッドの**返り値**または**戻り値**という．

1.4　流れ制御

流れ制御とはアルゴリズムの流れを制御するために使われる特殊な命令である．フローチャートにおいては，命令と命令の間の関連付けを表わす矢印によって，流れ制御が表わされている．流れ制御には大きく言って**条件分岐**と**ループ**の二種類がある．

流れ制御については 3.2 節で扱う．

1.5　手続きの定義と使用

ある仕事を遂行するアルゴリズムを表現するプログラムが書けたら，今度はそれに名前をつけて，別なプログラムの中で使うことが出来る．

実際のプログラミング言語では，あるメソッドを構成し，それに名前をつけることもプログラムの内部で行われる．どんな大きなプログラムでも，少しずつメソッドの定義を積み重ねることによって作られる．数学で言えば，大きなプログラムは定理，そのプログラムの中で使われるために定義されるメソッドは補助定理にあたる．

上の例ではある整数が別の整数で割り切れるかどうかを判定する手続きが使われた．しかしこの手続きは言語によってはあらかじめ用意されていない可能性がある．しかしほとんどのプログラミング言語には，ある整数を別の整数で割ったときの余りを計算する手続きと，ある整数が別な整数と等しいかどうかを判定する手続きが用意されている．これらを組み合わせることである整数が別の整数で割り切れるかどうかを判定する手続きを作ることが出来る．

なお，上の例のアルゴリズムは，Ruby では次のように記述することが出来る．

```
def divide?(n, k)
  n % k == 0
end

def prime_factors(n)
  k = 2
  list = []
  while n != 1
    if divide?(n, k)
      list.push(k)
```

```
      n = n / k
    else
      k = k + 1
    end
  end
  list
end
```

上のプログラムで使われているメソッドを説明しよう．

- x `%` y は，整数 x を整数 y で割った余りを値として持つ．
- x `/` y は，整数 x を整数 y で割った商を値として持つ．
- x `==` y は，整数 x が整数 y と等しいときに `true` を，そうでないときに `false` を値として持つ．
- x `!=` y はその逆．
- `def...end` はメソッドの定義である．これは次の形で使われる．

  `def` メソッドの名前 (変数 1, ..., 変数 n)
  　　定義の内容
  `end`

  上では `divide?`と `prime_factors` という二つのメソッドが定義されている．定義されたメソッドを使うときには

  メソッドの名前 (入力値 1, ..., 入力値 n)

  という形で使う．この式は定義の内容に現れる変数 i に入力値 i を代入した式を表す．たとえば `divide?(6, 2)` は

  `6 % 2 == 0`

  を表す（したがってこの式は `true` を値として持つ)．
- `if...else...end` は条件分岐である．これは次の形式で使われる．

  `if` X
  　Y
  `else`
  　Z
  `end`

  X が `true` を値に持つとき，Y が実行され，その結果が値として返される．X が `false` であるとき，Z が実行され，その結果が値として返される．
- `while ... end` はループである．これは次の形式で使われる．

  `while` X
  　Y
  `end`

  X が `true` を値として持つ間，Y が繰り返し実行される．Y の実行の後にはそのたびに X が評価される．
- V `=` X は変数に値を指定する命令である．ここで V は変数，X は任意の式を表す．この命令が実行されると，それ以降，プログラムにおいて V という変数は X の値を表すことになる．

メソッドの定義については 5 節でより詳しく扱うが，その前にもメソッドを定義しながらプログラムを書いていく．

問題 1.1. (1) `prime_factors(210)` はどのように計算され，どのような値を返すか．

(2) `prime_factors` が実際に素因数分解の手続きになっていることを確かめよ．

(3) 次のプログラムを実行するとどうなるか．

```
while 1 == 1
  puts "Hello!"
end
```

ただし `puts "..."`は文字列「...」を画面に表示する命令である．

(4) 二つの正整数 n，m の最大公約数を求める手続きをフローチャートで表せ．

(5) `%` を使わずに `x` を `y` で割った余りを計算する手続きをフローチャートで表せ．（`+`，`-`，`<=`，`==`などは使ってよい．）

(6) 次の手続きをフローチャートで表せ．

入力：正整数 n

出力：正整数の配列 list

手続き：

list を $[n]$ としておく．n が 1 になるまで以下を繰り返す．n が偶数ならば $n = n/2$ とし，そうでなければ $n = (3n+1)/2$ とする．list に n を加える．

またこのアルゴリズムに従って，入力 7，9，85 に対する出力をそれぞれ求めよ[*2]．

1.6 Ruby を使うには

Ruby でプログラムを書くときは，拡張子として「.rb」を持つファイルをテキスト・エディタで編集する．Ruby のプログラムを実行するときは，コマンドプロンプトで実行したいプログラム・ファイルの置かれているフォルダに移動して，

ruby ファイル名.rb

と命令する．またはプログラム・ファイルの置かれているフォルダに移動して

irb

と命令する．こうすると interactive ruby（IRB）というプログラムが起動する．IRB ではキーボードから入力された式が即座に Ruby のインタプリタによって評価されその値が表示される．ここで

[*2] この手続きが任意の正整数に対して終了するかどうかは知られていない．これを「シラキュース問題」という．

load ”ファイル名.rb”

と命令すると，指定されたプログラム・ファイルが読み込まれる
　コマンドプロンプトは「スタート」＞「すべてのプログラム」＞「アクセサリ」から起動する．フォルダの移動は

cd フォルダ名

によって行う．ただしこの方法では現在のフォルダの子フォルダにしか移動できない．現在のフォルダの親フォルダに移動するには

cd..

と打ち込む．現在のフォルダの子フォルダをすべて表示するには

dir

と打ち込む．別のドライブに移動するには

ドライブ名:

とだけ打ち込めばよい．これはどこのフォルダからも有効な命令である．京大の端末ではマイドキュメントのフォルダは「M」というドライブになっているためコマンドプロンプトで

M:

と打ち込めばマイドキュメントに移動することが出来る．
　コマンドプロンプトで日本語入力モードに切り替えるには [alt] + [半角/全角] を押す．元に戻す時は [半角/全角] を押す．

問題 1.2. コマンドプロンプトから IRB を起動し，キーボードから以下の式を入力せよ（入力した後に [Enter] を押す）．

(1) `5 - 2 * 3 + 1`

(2) `9 / 4`

(3) `9 % 4`

(4) `0 / 0`

(5) `x`

(6) `x = 3`

(7) `x * x`

(8) `x = 5`

(9) `x * x`

(10) `x == 5`

(11) `x == 3`

(12) `greeting = "Hello!"`

(13) `puts greeting`

(14) `puts greeting.downcase`

(15) `puts greeting.upcase`

(16) `puts greeting.capitalize`

(17) `puts greeting[1..8]`

(18) `while x > 0 do puts greeting; x = x - 1 end`

(19) `x`

(20) `ymo = ["Hosono", "Takahashi", "Sakamoto"]`

(21) `ymo[0]`

(22) `ymo[1]`

(23) `ymo[2]`

(24) `ymo[3]`

(25) `ymo[-1]`

(26) `ymo[1..2]`

問題 1.3. 以下のプログラムを書いた Ruby ファイルを作り，IRB で読み込み使用しなさい.

(1) `prime_factors`

(2)

```
def gcm(x, y)
  if x <= y
    measure = x
  else
    measure = y
  end
  while not(divide?(x, measure) && divide?(y, measure))
    measure = measure -1
  end
  measure
end
```

(3)

```
def syracuse(n)
  list = [n]
  while n != 1
    if divide?(n, 2)
      n = n / 2
    else
```

```
      n = (3 * n + 1) / 2
    end
    list.push(n)
  end
  list
end
```

2 文字列と変数

本節では，文字列を扱う基本的なメソッドをいくつか紹介する．それと同時に変数の扱い方を覚えよう．

文字列に対する操作において，半角文字と全角文字に関して扱いが異なる場合がある．そこで以下では半角文字のみからなる文字列を考えることにする．

文字列とは文字が並んだものである．文字には様々なものがあるが，ここでは半角アルファベット，半角数字，半角空白，および記号

```
!   ,   -   .   (   )   :   ;   ?   [   ]   ^   _   {   }
```

だけを考えることにしよう．特殊な文字列として，含まれる文字が 0 個の文字列というものも考える．

2.1 文字列の出力と入力

コンピュータが何かの仕事を行ったとして，その結果が何らかの仕方で表示されなければ，私たちはコンピュータの内部で何が行われたのかを知ることはできない．従ってコンピュータに，処理の結果を画面に表示させる方法が必要である．このために `print` というメソッドが用意されている．たとえばコンピュータに「Hello!」と表示させてみよう．それにはプログラムに

```
print "Hello!"
```

と書けばよい．このプログラムを実行させるとコンピュータの画面には

```
Hello!
```

と表示される．つまり `print` の後ろの二重引用符「`"`」ではさまれた部分がそのまま表示されるのである[*3]．それでは

```
print "Hello!"
print "Good-bye!"
```

と書くとどうなるだろう．この実行結果は

```
Hello!Good-bye!
```

[*3] Ruby で文字列を表すための引用符は他にもあるが，ここではそれらについては触れないことにする．

となる．print は明示的な改行記号がなければ改行せずに文字列を表示する．文字列を独立した行として表示したい時は puts というメソッドが使える．

```
puts "Hello!"
puts "Good-bye!"
```

と書けば，実行結果は

```
Hello!
Good-bye!
```

となる．

コンピュータに入力を与える方法としては gets メソッドがある．例えばプログラム

```
gets
```

を実行するとコンピュータはキーボードからの入力待ちの状態になる．ここで例えば「Hello!」と打ち込み，Enter を入力するとコンピュータは gets の値として「Hello!」という文字列を受け取る．ただしこのプログラムはそれ以上のことは何もしない．コンピュータが受け取った値を画面に表示させたければ

```
puts gets
```

と書けばよい．このプログラムを実行すると，最初は入力待ちの状態になり，そこで「Hello!」と入力し [Enter] を押すと，画面には「Hello!」という出力が現れる．この実行結果は次のようになる（キーボードからの入力はイタリックで表すことにする）．

Hello!

`Hello!`

最初の「*Hello!*」はキーボードからの入力，二つ目「Hello!」はコンピュータが出力したものである．なお gets はキーボードからの入力を [Enter] が押されるまで受け取る．最後の [Enter] も gets で取得した文字列には含まれている．最後の [Enter] を取り除くには chomp という命令を使う．

問題 2.1. (1) IRB で gets と puts を使用してみなさい．

(2) 次のプログラムを書いた Ruby ファイルを作り，実行しなさい．

```
def ask_name
  puts "What is your name?"
  puts "Hello, " + gets.chomp + "!"
end
ask_name
```

2.2 変数

同じ文字列を何度も使いたい場合がある．そのときはその文字列に名前をつけて記憶しておくと便利である．例えば「Hello! How are you?」という文字列に s という名前をつけることにしよう．プログラミングの用語では，オブジェクトにつけられる名前を**変数**と呼ぶ．ある変数を導入して，その値としてあるオブジェクトを与えることを，その変数をそのオブジェクトで**初期化する**という．変数 s を導入して，文字列「Hello! How are you?」によって初期化する時には次のように書く．

```
s = "Hello!  How are you?"
```

こうしておけば，同じプログラムの中で s は常に「Hello! How are you?」という文字列を指すことになる（ただしこの名前が他のオブジェクトに付け換えられるまで）．

```
s = "Hello!  How are you?"
puts s
puts s
```

というプログラムの実行結果は

```
Hello!  How are you?
Hello!  How are you?
```

となる．また gets で得た文字列によって変数を初期化することもできる．そのときはプログラムに

```
s = gets
```

と書く．既に使った変数に異なる値を代入することもできる．例えば，

```
s = "Hello!"
puts s
s = "Good-bye!"
puts s
```

の実行結果は

```
Hello!

Good-bye!
```

である．

変数に変数の値を代入することもできる．例えば最初に s1 という変数に何かの値を入れておいて，その値を別な変数 s2 に渡すというように．

```
s1 = "Hello!"
s2 = s1
puts s1
puts s2
```

の実行結果はもちろん

```
Hello!
Hello!
```

である．注意しなければならないのは変数 s2 は変数 s1 を指しているのではないということである．変数 s2 は変数 s1 を使って初期化されているが，その後は s2 は s1 と独立に「Hello!」という文字列を直接指している．このことをはっきりさせるために次のプログラムを実行した結果がどのようになるかを考えてみよう．

```
s1 = "Hello!"
s2 = s1
s1 = "Good-bye!"
puts s2
```

2 行目で s2 は s1 によって初期化されている．s1 はこの時点で「Hello!」という文字列を値として持つ．従って s2 は「Hello!」という文字列を値として持つことになる．その後，3 行目で s1 の値は「Good-bye!」に変更されているが，そのことは s2 に値に影響を与えない．従ってこのプログラムを実行すると

```
Hello!
```

が出力される．

2.2.1　変数に対する制限

変数として使える文字列には次のような制限がある．

- 半角アルファベット，半角数字，「_」のみからなる．
- 先頭の文字が数字でない．
- 先頭の文字が大文字のアルファベットでない．
- 予約語として指定されていない．

プログラミング言語の**予約語**とは，その言語の中で意味が固定されている文字列であり，それを変数として使うことはできない．Ruby では例えば以下の文字列が予約語である．

```
if  else  elsif  end  unless  while  do  until  true  false  nil  def
```

問題 2.2. 以下の文字列のうち，変数として使えないものはどれか．またそれが使えないのはなぜか．

```
4oranges   VQie8a2   _var   int?   nil0   elsif
```

2.3　文字列に対するメソッド

メソッドとはオブジェクトに対する操作，作業である．メソッドは作業の結果を値として持つ．この値のことをメソッドの**返り値**または**戻り値**，あるいは単に**値**と呼ぶ．またメソッドはその返り値を**返す**と言われる．

Ruby では文字列を操作する様々なメソッドが与えられている．その内の基本的なものを紹介しよう．以下で s，t は任意の文字列を，i, j は整数を表すものとする．

`s + t`

s と t を合わせた文字列を返す．従って

```
"abcd" + "efgh"
```

は文字列「abcdefgh」を返す．

`s * i`

s を i 回並べた文字列を返す．従って

```
"abc" * 3
```

は文字列「abcabcabc」を返す．

`s.length`

文字列の長さを整数で返す．文字列の長さとは，その文字列に含まれる文字の数を指す[*4]．従って

```
"Hello!  How are you?".length
```

は整数 19 を返す．空白も 1 文字としてカウントされることに注意．

`s.index(t)`

s の中で文字列 t が最初に現れる位置を整数で返す．ただし文字列の中でのある文字列の位置とは，その文字列よりも前にある文字の数とする．従って先頭の文字列の位置は 0 になることに注意せよ．もし t が s の中に現れないときは nil を返す[*5]．従って s が「abcde」という文字列だとすると，

```
s.index("abc")
s.index("e")
s.index("ac")
```

[*4] ただし半角文字は 1 個で 1 文字，全角文字は 1 個で 2 文字にカウントされる．混乱を避けるために，このテキストでは半角文字だけを扱うことにする．

[*5] nil は Ruby の予約語で例外的な値であることを表すときに用いられる．

はそれぞれ，0，4，nil を返す．

`s.index(t, i)`

s の中で位置 i 番目以降で文字列 t が最初に現れる位置を返す．もし t が s の中で位置 i 番目以降で現れないときは nil を返す．

`s < t   s <= t   s > t   s >= t   s == t`

それぞれ，辞書配列で s が t に対して，より前であるとき，より前であるかまたは等しいとき，より後であるとき，より後であるかまたは等しいとき，等しいときに true を返し，そうでないときに false を返す[6]．

ここでの辞書配列は，小文字よりも大文字が優先される点で通常の辞書配列と異なる．主な文字を優先順位の高い順に並べると以下のようになる．

```
! , - . ( ) 0 9 : ; ? A Z [ ] ^ a z { }
```

また半角スペースは最も優先順位が高い．従って，

```
"abc" < "abcd"
"Hello," < "Hello."
"Abc" < "ab"
"Za" < "aa"
"3" < "three"
"123" < "23"
" 23" < "123"
```

はすべて true を返す．

`s[i..j]`

s の位置 i 番目から j 番目までの文字列を返す．また i と j に同じ数字を入力すると i 番目の一文字だけを返す．文字列の長さを超える指定範囲は無視される．また，i が文字列の長さを超える値であるときには nil を返す．従って

```
"abcde"[1..3]
"abcde"[2..10]
"abcde"[5..7]
"abcde"[6..7]
```

はそれぞれ文字列「bcd」，文字列「cde」，文字列「」，および nil を返す．

[6] true と false は予約語で，主に条件文などにおいて使われる特殊な値である．

`s.upcase` `s.downcase` `s.capitalize`

それぞれ s に含まれるアルファベットをすべて大文字にした文字列，s に含まれるアルファベットをすべて小文字にした文字列，s の先頭のアルファベットを大文字にし，他のアルファベットをすべて小文字にした文字列を返す．これらはアルファベット以外の文字に対しては何もしない．従って，

```
"aPpLe?".upcase
"aPpLe?".downcase
"aPpLe?".capitalize
```

はそれぞれ文字列「APPLE?」,「apple?」,「Apple?」を返す．

問題 2.3. 以下の表現がどのような値を返すか答えよ．

(1) `"Hello" * 3`

(2) `"Good morning!" + "How are you?"`

(3) `"How do you do?".index("do")`

(4) `"How do you do?".index("do", 5)`

(5) `"John" < "Jack"`

(6) `"Good morning!"[3..5]`

(7) `" apple".capitalize`

2.3.1 式と評価

プログラムの中で意味を持つ表現，例えば，

```
"abc"
s = "abc"
puts "abc"
(("abc" + "def") * 3).length
```

などの表現を**式**と呼ぶ．任意の式は何らかの値を持つ．式の値を求めることを，その式を**評価する**と言う．プログラムとは式の集まりである．プログラムの実行とはそれらの式を実際に評価することであると言っても良い．

2.3.2 メソッドの適用の順序

一つの式の中に複数のメソッドの適用が現れるとき，そのメソッドが適用された順序を示す必要がある場合がある．例えば

```
"ab" + "cd" * 3
```

という式について考えよう．この場合，算数の場合と同じように，`*`が先に評価される．従って上の式の値は文字列「abcdcdcd」である．もしも`"ab" + "cd"`を先に評価して，その値に`* 3`を適用したいときには

```
("ab" + "cd") * 3
```

というように括弧を使って，メソッドの適用の順番を明示しなければならない．この式を評価した結果は「abcdabcdabcd」になる．

XXX`.`xxx または XXX`[`xxx`]` の形のメソッドは `+` や`*`よりも優先的に実行される．また XXX `==` YYY，XXX `>` YYY などは上のどれよりも優先順位が低く，より後に実行される．

例 2.4. 変数 `s` と `t` はそれぞれ文字列「abcdefg」,「hijklmn」を値として持つものとする．以下の式がどのような値を持つかを考えてみよう．

(1) `s[2..5]`

先頭の文字を 0 番目と考えるから，2 番目から 5 番目の文字列は「cdef」である．

(2) `t[4..5] + s[2..5]`

`t[4..5]` は文字列「lm」を値に持つ．また `s[2..5]` は「cdef」を値に持つ．よってこの式の値は「lmcdef」である．

(3) `(s + t).length`

`s + t` は文字列「abcdefghijklmn」を値に持つ．この長さは 14 である．よってこの式の値は整数 14 である．

(4) `s.index("fg")`

先頭の文字を 0 番目と考えるので，`s` の中で「fg」が現れる位置は 5 番目である．従ってこの式の値は 5 である．

(5) `(s * 2).index("fg", s.length)`

`s.length` の値は整数 7 である．7 番目以降で `(s * 2)` の中に文字列「fg」が現れるのは 12 番目である．従ってこの式の値は 12 である．

問題 2.5. 変数 `a` と `p` はそれぞれ文字列「apple」,「pineapple」を値として持つものとする．以下の式はどのような値を返すか答えよ．

(1) `(a[0..1] + a[2..4]) == p[4..8]`

(2) `a[1..2] * p[3..4].length`

(3) `a[p.index(a)..p.index(a)]`

(4) `a[3..4].upcase + p[0..2].capitalize`

問題 2.6. 変数 `s` は文字列「aBcDeF」を値として持っているとする．この変数とこの節で学んだメソッドのみを使って以下の値を持つ式を書け．

(1) 文字列「cDe」

(2) 文字列「Abc」

(3) 文字列「abEF」

(4) `true`

(5) `false`

(6) 整数 18

(7) 整数 50

(8) `nil`

3 整数の演算と流れ制御

この節では整数についての基本的な演算メソッドを学ぶ．それと同時に `if` と `while` を使った流れ制御の方法を学ぶ．

3.1 整数の四則演算と比較

以下で `i`，`j` は任意の整数を値に持つとする．

- `i + j`
  `i` と `j` の和を返す．
- `i * j`
  `i` と `j` の積を返す．
- `i - j`
  `i` から `j` を引いた差を返す．
- `i / j`
  `i` を `j` で割ったときの商（整数）を返す．ただし `j` は 0 以外の整数でなければならない[*7]．例えば `5 / 2` の返り値は 2 である．
- `i % j`
  `i` を `j` で割ったときの余りを返す．ただし `j` は 0 以外の整数でなければならない．例えば `5 % 2` の返り値は 1 である．
- `i < j`　`i <= j`　`i > j`　`i >= j`　`i == j`
  それぞれ `i` が `j` に対して，より小さいとき，以下であるとき，より大きいとき，以上であるとき，等しいときに `true` を返し，そうでないときには `false` を返す．

[*7] `j` の値が 0 のときにこの式を評価させようとすると，エラーが生じる．

小数点以下の計算をさせるには最初から整数ではなく浮動小数点数というクラスのオブジェクトを使う[*8]．ここでは整数の計算のみを扱う．

文字列に対するメソッドと同様に，上の演算のメソッドにも実行の際の優先順位がある．優先的に実行される順にグループ分けすると，

1. `*` `/` `%`
2. `+` `-`
3. `<` `<=` `>` `>=` `==`

となる．同じグループのメソッドであれば左から優先的に実行される．従って例えば式

```
5 + 3 % 2 * 8 < 4 - 9 / 3 + 2
```

は

```
(5 + ((3 % 2) * 2)) < ((4 - (9 / 3)) + 2)
```

として評価される．

問題 3.1. 以下の式はどのような値を返すか．

(1) `9 % 4 * 2`

(2) `-3 + 6 * 4 - 3`

(3) `9 / 3 * 3 % 5 <= 5 - 3 * 3`

(4) `(-2 + 4) / 3 > 5 + -3 * 2`

(5) `"cat".length * 3 == ("cat" * 3).length`

(6) `"apple".index("l") + "orange".length / 2`

3.2 流れ制御

3.2.1 if

ある条件が成立しているか否かに従って，その後の処理の流れを分岐させる必要がある場合がある．たとえば任意の整数に対してその絶対値を与えるというアルゴリズムを考えよう．これは次のように記述できる．

アルゴリズム **

絶対値　任意の整数に対してその絶対値を返すアルゴリズム．

[*8] クラスとはオブジェクトの分類であって，メソッドはそれぞれのクラスに対して定義されている．

`+` `*` `<` `<=` `>` `>=` `==`

などのメソッドは，整数に対しても文字列に対しても定義されている．しかしこれらは異なるクラスに対して定められた異なるメソッドであることに注意しなければならない．

入力：整数 i

出力：i の絶対値 $|i|$

手続き：i が 0 または正の整数ならば i を返し，そうでない場合は $-i$ を返す．

****

このようなアルゴリズムを実行するために，if を使うことができる．if は次のような形で使われる．

```
if X
  Y
else
  Z
end
```

X，Y，Z には任意の式が入る[*9]．このプログラムは次のように実行される．まず式 X が評価される．その値が true であれば Y が実行され，その返り値がこの式全体（if から end まで）の返す値となる．もし X の値が false または nil だった場合は Z が実行され，その返り値がこの式全体の返す値となる[*10]．

これを使うと整数 i の絶対値を返すメソッドは次のように定義できる．

```
def abs(x)
  if x >= 0
    x
  else
    -x
  end
end
```

X の値が true の時には Y を実行し，そうでない場合には何もしないという命令をしたいときには次のように書けばよい．

```
if X
  Y
end
```

インタプリタが X を false または nil と評価したときは，インタプリタはそれ以上何も行わない．この場合 if から end までの式全体の値は nil になる．

3 個以上の条件によって分岐を行いたいときには elsif を使うことができる．これは次のように使われる．

```
if X1
  Y1
elsif X2
```

[*9] 式を複数個，セミコロンか改行で区切って並べたものも式と見なされる．その場合，最後に評価された式の値がその式の値と見なされる．例えば式

```
3 + 5 == 8; "abc" + "xyz"; 5 * 8
```

の値は 40 である．

[*10] 実際には Ruby においては X が true の場合だけでなく，false または nil 以外のどんな値のときにも Y が実行される．しかしここでは簡単のために，X は true か false か nil のいずれかの値を持つ式であるものと考える．

Y_2
`elsif` X_3
Y_3
$\vdots$
`elsif` X_{n-1}
Y_{n-1}
`else`
Y_n
`end`

例 3.2. 変数 byear，bmonth，bday はそれぞれ，ある人の誕生年，誕生月，誕生日を値として持つとする．また変数 cyear，cmonth，cday にはそれぞれ現在の年月日を値として持つとする．このとき次のアルゴリズムによってその人の満年齢を計算することができる．

```
if cmonth > bmonth
  cyear - byear
elsif cmonth < bmonth
  cyear - byear - 1
elsif cday >= bday
  cyear - byear
else
  cyear - byear - 1
end
```

問題 3.3. i，j は任意の整数とする．このとき次のアルゴリズムを記述せよ．

(1) i と j のうちで大きい方を返す．

(2) i と j のうちで 5 に近い方を返す．例えば i の値が 2 で j の値が 7 であれば j を返す．

3.2.2 while

ある条件が成り立っている間，ある処理を繰り返して続けるというアルゴリズムを記述するのに，while を使うことができる．これは次のような形式で使われる．

`while` X
Y
`end`

これは次のように実行される．まず X が評価されて，その値が false でも nil でもない場合は Y が実行される．その後，Y が実行される毎に再度 X が評価されて，その値が false か nil になるまで同じことを繰り返す．X の値が false か nil になった時点で処理は終了となる．なお while 式の処理が終了したときは必ず X が false または nil と評価され，Y の処理をスキップすることになるため，while 式の評価は常に nil になる．

なお while 式の評価は終了しない場合もある．例えば次のプログラムを考えよう．

```
while true
  puts "endless"
end
```

このプログラムを実行すると，コンピュータはいつまでも endless を表示することを繰り返す．このときは処理を強制的に中断させなければならない．Windows のコマンドプロンプトでは [Ctrl] + C を押すことで処理を中断させることができる．

例 3.4. i は整数，j は非負整数であるとする．このとき i の j 乗を計算する方法を考えよう．これは 1 に i を j 回かけることによって得られる．

アルゴリズム **

累乗　任意の整数 i と任意の非負整数 j に対して，i の j 乗を計算するアルゴリズム．

入力：整数 i, j

出力：i の j 乗

手続き：

(1) $k = 0, z = 1$ とする．

(2) $k < j$ ならば 3 に進む．そうでなければ z を返して手続きを終了する．

(3) $z = z * i$，$k = k + 1$ とする．2 に戻る．

****

ここでは $k < j$ という条件が成立している間は (3) を繰り返すということが行われている．(3) が実行されるたびに k の値は 1 づつ増えていくので，k は (3) が何回実行されたかを数える役割を果たしている．(3) が j 回実行されると $k < j$ が成り立たなくなるのでそこで手続きは終了する．そのとき z の値は 1 に i を j 回かけた数になっている．従ってこのアルゴリズムは i の j 乗を計算する．

このアルゴリズムを while を使って書くと次のようになる．

```
k = 0
z = 1
while k < j
  z = z * i
  k = k + 1
end
z
```

例えば i の値が 2，j の値が 3 の場合を考えてみよう．k の初期値は 0 なので，k < j の値は true．よって while ループの中に入る．z は 1 なので z * i の値 2 が新しい z の値になる．また k + 1 の値 1 が新しい k の値になる．まだ k < j の値は true なので，再び while ループの中に入る．z * i の値 4，k + 1 の値 2 がそれぞれ z，k の新しい値になる．まだ k < j の値は true なので，みたび while ループの中に入る．z * i の値 8，k + 1 の値 3 がそれぞれ z，k の新しい値になる．ここで k < j の値が false になるので while ループから抜け出す．そのときの z の値がこの式全体の値である[*11]．

例 3.5. (1) n の値は非負整数であるとする．このとき 0 から n までのすべての整数の合計を計算するアルゴ

[*11] while 式は必ず nil を返すので，最後に z を付け足す必要がある．

リズムは次のように記述できる.

```
k = 0; z = 0
while k < n
  z = z + k
  k = k + 1
end
z
```

(2) 次のような数列が与えられたとする.

1, 2, 2, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5, 5, ...

この数列の n 項目までの和を返すアルゴリズムは次のように記述できる.

```
i = n; j = 1; sum = 0
while i - j >= 0
  sum = sum + (j * j)
  i = i - j
  j = j + 1
end
sum = sum + (j * i)
```

このプログラムの返り値は，最後に評価される式 `sum = sum + (j * i)` の値に等しいが，一般に変数への代入を表す式の値は，そのときに代入される値に等しい.

(3) s は任意の文字列，t は一文字からなる文字列を値として持つ変数であるとする．このとき，s の中に t が何回現れるかを数えるアルゴリズムは次のように記述できる.

```
k = 0; z = 0
while k < s.length
  if s[k..k] == t
    z = z + 1
  end
  k = k + 1
end
z
```

文字列の最後の文字の位置は，その文字列の長さから 1 引いた整数になることに注意.

(4) s は任意の文字列であるとする．このとき s の各文字の間に「,」を挿入した文字列を返すアルゴリズムは次のように記述できる.

```
k = 0; l = s.length; t = ""
while k < l - 1
  t = t + s[k..k] + ","
  k = k + 1
end
if l > 0
  t = t + s[l - 1..l - 1]
end
t
```

問題 3.6. 以下のアルゴリズムを記述せよ.

(1) 次の数列の n 番目の項を返すアルゴリズム[12].

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...$$

(2) 次の数列の n 番目の項を返すアルゴリズム.

$$1, 2, 2, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5, 5, ...$$

(3) 次の数列の n 番目の項を返すアルゴリズム.

$$1, 1, 2, 1, 2, 3, 1, 2, 3, 4, 1, 2, 3, 4, 5, ...$$

(4) 文字列 `s` に対して，`s` を逆にした文字列を返すアルゴリズム．例えば「abcde」という文字列に対して「edcba」を返す.

4 論理演算

日本の法律では女性ならばは 16 歳以上，男性ならば 18 歳以上で結婚できることになっている．この法律を `if` を使ったプログラム風に記述すると次のようになる.

```
if A は女性
  if A は 16 歳以上
    A は結婚できる
  end
elsif A は 18 歳以上
  A は結婚できる
end
```

しかしこれはもっと短く，次のようにも表現できる.

```
if A は女性　かつ　A は 16 歳以上
    A は結婚できる
elsif A は 18 歳以上
    A は結婚できる
end
```

さらに次のようにも表現できる.

```
if （A は女性　かつ　A は 16 歳以上）　または　A は 18 歳以上
    A は結婚できる
end
```

[12] この数列はフィボナッチ数列と呼ばれる.

このように複数の条件を「かつ」や「または」でつなげることによって複雑な条件を作ることができる．このようにして作られた複雑な条件の成立不成立は，その構成部分になっている条件の成立不成立に依存する．ある複雑な条件の成立不成立を，その構成部分になっている条件の成立不成立から計算することを**論理演算**と呼ぶ．また単純な条件から複雑な条件を作るための「かつ」,「または」などの語を**論理演算子**あるいは**論理結合子**と呼ぶ．

論理演算子を適切に使うことでプログラムをより短く分かりやすく記述することができる．ここでは「かつ」,「または」,「ではない」にあたる論理演算子を学ぶことにしよう．

p，q は真偽値（true または false）を値に持つ変数であるとする*13．

- p && q
  「かつ」に対応する．p と q の値がともに true のときに true を返す．どちらかが false ならば false を返す．
- p || q
  「または」に対応する．p と q の値がともに false のときに false を返す．どちらかが true ならば true を返す．
- !p
  「ではない」に対応する．p の値が true ならば false を返す．p の値が false ならば true を返す．

これらの論理演算の規則は以下の表によって表される．

p	q	p && q
true	true	true
true	false	false
false	true	false
false	false	false

p	q	p || q
true	true	true
true	false	true
false	true	true
false	false	false

p	!p
true	false
false	true

論理結合子の適用の優先順位は，!，&&，||の順に高くなっている．従って例えば

```
!p && q || r
```

は

```
((!p) && q) || r
```

として計算される．

問題 4.1. 以下の式を評価しなさい．

(1) !((6 / 2 == 3) && (6 % 4 == 1))

*13 Ruby では論理演算の対象は真偽値に限らない．しかしここでは簡単のために真偽値に対する計算結果のみを挙げる．

(2) `!(6 / 2 == 3) && (6 % 4 == 1)`

(3) `!(("abcd".length) == "vwxyz".index("y"))`

(4) `!(true && false) && !(true) || !(true || false)`

問題 4.2. 以下の式の値が true になるような n，s の値を求めなさい（答えが一意に定まるとは限らない）．ただし n の値は非負整数，s の値は文字列であるものとする．

(1) `(s.upcase == "AB") && (s.capitalize == s)`

(2) `(n % 3 < n % 2) && (n < 10)`

(3) `(n * n == 30) || (n * n == n + n)`

(4) `!(!(n == n * n) || (n % (n + 1) == 1))`

例 4.3. 与えられた正整数が素数であるかどうかを判定するアルゴリズムを考えよう．ただし正整数 n が素数であるのは，n が 1 より大きく，1 と n 以外のいかなる整数でも割り切れないときである．n より大きないかなる正整数によっても n が割り切れないことは明らかなので，1 より大きく，n より小さなすべての正整数を調べて，そのどれもが n を割り切ることがないことが分かれば n は素数である．逆に 1 より大きく n より小さな正整数で n を割り切るものがあれば n は素数ではない．

アルゴリズム **

素数の判定　任意の正整数 n に対して，n が素数であるかどうかを判定するアルゴリズム．

入力：正整数 n

出力：n が素数なら true，そうでなければ false.

手続き：

```
if n == 1
  false
else
  k = 2; b = true
  while (k < n && b == true)
    if n % k == 0
      b = false
    end
    k = k + 1
  end
  b
end
```

****

問題 4.4. s，t は任意の文字列を，m，n は任意の非負整数を値に持つとする．このとき次の条件を満たす式を書け．

(1) s が t を含むか t が s を含むときに true，そうでないときには false を返す（ヒント：メソッド index を使えば，ある文字列に別な文字列が含まれているかどうかを調べることが出来る）．

(2) 文字列"abc"が s と t の両方に含まれているときに true，そうでないときには false を返す．

(3) s の先頭の文字が小文字であるときに true，そうでないときに false を返すただし s の長さが 0 の場合には false を返すものとする（ヒント：大文字に変換してもとの文字と変わっていれば小文字である）．

(4) s の長さが m 以上かつ n 以下であるときに true，そうでないときに false を返す．

(5) s の m 番目の文字と n 番目の文字が等しくないときに true，等しいときに false を返す．ただし m，n が s の長さ以上の値である場合のことは考えないものとする．

問題 4.5. s は任意の文字列を値に持つとする．このとき与えられた条件を満たすように以下の式の空欄に適切な式を補え．

(1) s に含まれる"a"の数と"b"の数の合計を返す．

```
k = 0; z = 0
while k < s.length
  if ( i )
    z = ( ii )
  end
end
( iii )
```

(2) s に"a"が奇数回現れていれば true，偶数回（0 回の場合も含めて）現れていれば false を返す．

```
k = 0; b = false
while k < s.length
  if s[k..k] == "a"
    b = ( i )
  end
  k = k + 1
end
( ii )
```

問題 4.6. 以下の式は何を行うアルゴリズムを表したものか．

(1)

```
k = 0; b = false
while (k < s.length - 1) && (b == false)
  if s[k..k] == s[(k + 1)..(k + 1)]
    b = true
  end
  k = k + 1
end
b
```

(2)

```
k = 0; b = true
while (k < s.length / 2) && (b == true)
  if !(s[k..k] == s[(s.length - (k + 1))..(s.length - (k + 1))])
    b = false
  end
  k = k + 1
```

```
end
b
```

5 メソッドの定義

私たちはこれまでに様々なアルゴリズムを考えてきた．さらにこれらのアルゴリズムを組み合わせて，より複雑なアルゴリズムを作ることが出来る．その際，すでに作られたアルゴリズムをメソッドとして定義し，別なアルゴリズムの中で使用することが出来る．

5.1 メソッドの定義

例えば，与えられた二つの文字列 s，t について，s が t を含むかどうかを判定するメソッドを include?を定義したいとする．これは次のように書かれる．

```
def include?(string1, string2)
  string1.index(string2) != nil
end
```

このように定義しておけば，このメソッドを呼び出すことで，ある文字列が別な文字列に含まれているかどうかの判定をすることが出来る．例えば，式

```
include?("abcd", "bc")
```

は true を返すし，

```
include?("bc", "abcd")
```

は false を返す．

一般的にメソッドは，そのメソッドが適用される対象を持つ．これをメソッドの**引数**（ひきすう）と呼ぶ[*14]．上の include?の定義では string1 と string2 がこのメソッドの引数を表している．引数はなくても良いし，複数個あっても良い．ただし引数を持たないメソッドを使うことはほとんどないだろう．

メソッドの定義の一般的な形式は次の通りである．

```
def メソッドの名前（引数）
    定義の内容
end
```

メソッドの名前は好きなように選ぶことが出来るが，変数と同様の制限がある[*15]．

[*14] 「引数」は数学の用語で，関数が適用される対象のことを指す．例えば関数 $f(x) = x^2 - 3x$ において，x に代入される値がこの関数の引数である．$f(x)$ が引数として 2 を取ったとき，$f(x)$ の値は -2 になる．

[*15] 2.2.1 節を参照せよ．

例 5.1. (1) 与えられた整数の絶対値を返すメソッド abs の定義.

```
def abs(int)
  if int >= 0
    int
  else
    -int
  end
end
```

(2) abs を用いることで，2 個の整数の間の差を求めるメソッド difference が次のように定義できる.

```
def difference(int1, int2)
  abs(int1 - int2)
end
```

difference を使うと，問題 3.3 (6) の手続きは，次のメソッドとして定義することが出来る.

```
def closer_to_five(int1, int2)
  if difference(5, int1) <= difference(5, int2)
    int1
  else
    int2
  end
end
```

3 個の引数をとり，第 2 引数と第 3 引数のうち，第 1 引数に近いものを返す手続きを，次のように定義することが出来る.

```
def closer(int1, int2, int3)
  if difference(int1, int2) <= difference(int1, int3)
    int2
  else
    int3
  end
end
```

(3) 与えられた二つの整数のうち小さい方を返すメソッド min の定義.

```
def min(int1, int2)
  if int1 <= int2
    int1
  else
    int2
  end
end
```

複雑な手続きを定義するときは，その手続きをより単純な手続きに分解して，部分的手続きをメソッドとして定義していくことが有用である．より大きなプログラムを書くときには，求める手続きが，どのような部分的手続きに分解されるかを理解することが重要な鍵となる．これはコンピュータ・プログラミングに限らず，およそあらゆる学問や技術においても重要な考え方である．

問題 5.2. 以下のメソッドを定義せよ（上で定義された abs および min を用いること）.

(1) 整数を二つ引数にとり，それぞれの絶対値の和を返すメソッド，sum_abs.

(2) 整数を二つ引数にとり，それらの和の絶対値を返すメソッド，abs_sum.

(3) 整数を二つ引数にとり，それぞれの絶対値を比較して，小さいほうを返すメソッド，min_abs.

(4) 整数を二つ引数にとり，それらのうちの小さいほうの絶対値を返すメソッド，abs_min.

(5) 整数を一つ引数にとり，その 2 乗を計算するメソッド，square.

(6) 文字列を引数にとり，その文字列の前半を返すメソッド，former_half. ただし文字列の長さが奇数のときは，前半の方が 1 文字少ないものとする．例えば former_half("abcde") の値は"ab"になる.

(7) 文字列を引数にとり，その文字列の後半を返すメソッド，latter_half. ただし文字列の長さが奇数のときは，後半の方が 1 文字多いものとする.

(8) 二つの文字列を引数にとり，一方の文字列の前半と，もう一方の文字列の後半をつなげた文字列を返すメソッド，cross_over.

5.2 メソッドの再帰的定義

数学においては，ある関数を定義する際に，その関数自体を定義の中で用いることがある．このような定義は**再帰的定義**と呼ばれる．例えば問題 3.6 で扱ったフィボナッチ数列は，通常，次のように定義される.

$$\mathrm{fib}(n) = \begin{cases} 1 & (n \leq 2) \\ \mathrm{fib}(n-1) + \mathrm{fib}(n-2) & (n > 2) \end{cases}$$

これと同様の定義が，プログラミング言語においても可能である.

```
def fib(n)
  if n <= 2
    1
  else
    fib(n - 1) + fib(n - 2)
  end
end
```

これと同じメソッドは再帰を使わなくても，次のように定義できる.

```
def fib(n)
  k = 2; i = 1; j = 1
  while k < n
    z = j
    j = i + j
    i = z
    k = k + 1
  end
  j
end
```

一般に，再帰を使わずに定義できるならば，その方が処理の速度は速い．しかし再帰を使うことには，複雑

な手続きをシンプルに記述できるという利点がある．

例 5.3. (1) 累乗の計算を行うメソッド，`exp`.

```
def exp(x, y)
  if y == 0
    1
  else
    x * exp(x, y - 1)
  end
end
```

(2) キーボードから入力された文字列を繰り返すメソッド `repeat`．ただしキーボードから「bye」と打ち込まれたときに手続きを終了する．

```
def repeat
  puts "Input a string."
  str = gets.chomp
  puts str
  if str.downcase != "bye"
    repeat
  end
end
```

問題 5.4. 以下のメソッドを，再帰的に定義せよ．

(1) 階乗の計算をするメソッド，`factorial`．ただし階乗とは次のように定義される関数である．

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

(2) 与えられた文字列をコンマで一文字ずつ区切った文字列を返すメソッド，`insert_commas`．ただし最後の文字の後にはコンマを入れない．例えば `insert_commas("abc")` は，文字列`"a,b,c"`を返す．
(3) 与えられた文字列からコンマを取り除いた文字列を返すメソッド，`remove_commas`．例えば，`remove_commas("a,b,c")` は文字列`"abc"`を返す．

6 配列

プログラミングで複数のオブジェクトを一括して扱う必要がある場合がある．そのようなときに使われるのが**配列**である．

6.1 配列の作り方と使い方

Ruby では配列は

[$x_0, x_1, ..., x_n$]

という形式で表される．配列の要素は先頭から 0 番目，1 番目，... とかぞえる．list が配列を表す変数であるとき，その k 番目の要素は

list[k]

によって表される．

配列には長さを調べる length というメソッドがある．これは list.length という形式で用いる．この式は list によって表現される配列の長さ（その配列に何個の要素が含まれているか）を表す．

例 6.1. list は整数からなる配列であるとする．

(1) list の要素の合計を求めるメソッド：

```
def list_sum(list)
  count = 0
  length = list.length
  sum = 0
  while count < length
    sum = sum + list[count]
    count = count + 1
  end
  sum
end
```

(2) list の各要素を 2 乗したものを要素として持つ新しい配列を作るメソッド：

```
def list_square_sum(list)
  count = 0
  length = list.length
  newList = []
  while count < length
    item = list[count] * list[count]
    newList.push(item)
    count = count + 1
  end
  newList
end
```

(3) 整数からなる配列の平均値を求めるメソッド．ただし配列が空の場合は nil を出力する[*16]．

```
def list_ave(list)
  length = list.length
  if length == 0
    nil
  else
    (list_sum(list).to_f) / length
  end
end
```

to_f は整数型のオブジェクトを小数型のオブジェクトに変換するメソッドである．合計が整数のままだと，

[*16] nil は例外的な値を表す特殊なオブジェクトである．

/によって小数点以下が切り捨てられるために，合計を小数に変換しておく必要がある．

(4) 非負整数 n を入力として受け取り，0 から n までの整数からなる配列を出力するメソッド．

```
def enumerate(n)
  count = 0
  list = []
  while count <= n
    list.push(count)
    count = count + 1
  end
  list
end
```

(5) 二つの配列をつなげた新しい配列を出力するメソッド．

```
def append(list1, list2)
  count = 0
  length = list1.length
  new_list = []
  while count < length
    new_list.push(list1[count])
    count = count + 1
  end
  count = 0
  length = list2.length
  while count < length
    new_list.push(list2[count])
    count = count + 1
  end
  new_list
end
```

問題 6.2. 以下のメソッドを定義せよ．

(1) 整数からなる配列を受け取り，その最大値を返すメソッド `max`，および最小値を返すメソッド `min`．ただし入力された配列が空の場合は `nil` を返す．

(2) 整数からなる配列と整数を受け取り，その整数と等しい要素をすべて除いた配列を返すメソッド `exclude`．

(3) 整数からなる配列を受け取り，その最大値と最小値を除いた要素の平均を求めるメソッド `list_ave_exclude_max_min`．ただし配列の長さが 3 未満のときは `nil` を返す．

(4) 整数からなる配列と整数を受け取り，その配列がその整数を要素として含んでいるときに `true`，そうでないときに `false` を返すメソッド `contain?`．

(5) 整数からなる配列を受け取り，重複する要素を取り除いた配列を返すメソッド `contract`．例えば `contract([0, 1, 0, 2, 3, 2, 0])` は `[1, 3, 2, 0]` を返す．

(6)

問題 6.3. 以下のメソッドを定義せよ．

(1) 配列を受け取り，その配列と逆順の新しい配列を返すメソッド `reverse`．

(2) 整数からなる配列を受け取り，要素を小さい順に並べた新しい配列を返すメソッド `ascend_order`．例え

ば ascend_order([3, -6, 2, 5, -2, 3]) は，配列 [-6, -2, 2, 3, 3, 5] を返す．

6.2 配列を変化させる

配列に対しては，(1) ある要素を別な要素に置き換える，(2) 新しい要素を付け加える，(3) 要素を削除する，という操作が可能である．

(1) については

list[k] = z

という形式の命令を使う．これは list の k 番目の要素を z によって表される要素に置き換えることを命じる．

(2) については push というプリミティヴを使う．これは配列の最後に新しい要素を付け加えると同時に，その配列自体を返すメソッドである．これは list.push(item) という形式で用いる．この式は配列 list の最後に item という新しい要素を加えた上で，その配列を出力せよ，という命令を表している．たとえば list が [0, 1] という配列であれば list.push(2) を実行すると，それ以後 list は [0, 1, 2] という配列を表すことになる．

(3) については pop というプリミティヴを使う．配列の最後の要素を配列から削除すると同時にその要素を返すメソッドである．これは list.pop という形式で用いる．たとえば list が [0, 1] という配列であれば list.pop を実行すると，それ以後 list は [0] という配列を表すことになる．

例 6.4. (1) 以下は配列と整数を受け取り，その配列からその整数によって指定された位置にある要素を削除すると同時にその要素を返すメソッドである[*17]．

```
def remove!(list,n)
  if (n >= list.length || n < 0)
    nil
  else
    list2 = []
    while list.length > n + 1
      list2.push(list.pop)
    end
    output = list.pop
    while list2.length > 0
      list.push(list2.pop)
    end
    output
  end
end
```

(2)

問題 6.5. (1) ある配列を作り，それに pop を繰り返し適用し，どうなるかを確かめなさい．

[*17] Ruby ではオブジェクトに変更を及ぼすメソッドの名前には「!」を付ける習慣がある．

(2) 次の命令を実行して，その後で list2 の値がどうなっているかを確かめよ．

```
list1 = [0, 1, 2]
list2 = list1
list1.pop
list1.push(10)
```

(3) 配列と非負整数 n とオブジェクトを受け取り，その配列のその n 番目に，そのオブジェクトを挿入するメソッド insert!．例えば list が [0, 1, 2] という配列であるとき，insert!([list, 2, true) を実行すると，list は [0, 1, true, 2] という配列になる．

(4) 配列と非負整数 n, m を受け取り，その配列の n 番目と m 番目を入れ替えるメソッド exchange!．

問題 6.6. (1) 以下の図は「パスカルの三角形」と呼ばれる．

```
                  1
               1     1
            1     2     1
         1     3     3     1
      1     4     6     4     1
   1     5     10    10    5     1
1     6     15    20    15    6     1
...
```

この図の n 行目の配列を求めるメソッド pascal_triangle を定義せよ．ただし最初の行は 0 行目と数える．例えば pascal_triangle(3) は配列 [1, 3, 3, 1] を返す．（ヒント：再帰的定義を用いる．）

(2)

7 手続きオブジェクト

Ruby ではメソッドをオブジェクトとして扱うことが可能である．オブジェクトとしてのメソッドを手続きオブジェクトと呼ぶ．手続きオブジェクトの作り方は以下の通りである．

```
Proc.new do |引数|
  手続きの内容
end
```

例えば与えられた整数を 2 乗する手続きオブジェクト square を次のように定義する．

```
square = Proc.new do |x|
  x * x
end
```

このように定義された手続きオブジェクトは，通常のオブジェクトと同様に手続きに入力として渡したり，手続きの出力として返したりすることが出来る．また手続きオブジェクトを手続きとして使用するときは `call` というメソッドを使って手続きを呼び出す必要がある．例えば上の `square` を整数 3 に適用するときは `square.call(3)` と書く．この式の値は 9 である．

手続きオブジェクトは，他のメソッドの引数になったり，戻り値になることが出来る．例えば次のメソッドは任意の手続きオブジェクトを二回実行する手続きである．

```
def do_twice(proc)
  proc.call
  proc.call
end
```

問題 7.1. 以下のメソッドを定義せよ．

(1) 1 引数で入力と出力の型が等しい手続きオブジェクトを受け取り，その手続きを二回適用させる新しい手続きオブジェクトを返すメソッド `double`.

(2) 配列と，1 引数の手続きオブジェクトを受け取り，配列の各要素にその手続きを適用した結果を並べた配列を返すメソッド `map`.

(3) 配列と 1 引数述語（真偽値を返すメソッド）を表す手続きオブジェクトを受け取り，その述語を適用したときに `true` になる要素を並べた配列を返すメソッド `filter`.

(4) 配列と 1 引数述語を表す手続きオブジェクトを受け取り，その述語を `true` にする配列の要素のうち一番最初に現れるものの番号を返すメソッド `index_of`．ただしその述語を `true` にする要素が配列に現れない場合は `nil` を返す．

8 クラスの定義

Ruby ではオブジェクトのクラスを定義することが出来る．オブジェクトは一般的に，個々のオブジェクト（インスタンスと呼ぶ）ごとに異なる可変的な内部状態を持ち，またそのクラスのオブジェクト全般に定義されたメソッドを持つ．クラスを定義するというのは，この内部状態とメソッドの組み合わせを定義するということである．

8.1 例：BankAccount クラス

クラスはしばしば現実の世界の何らかの事物をシミュレートする目的で作られる．ここでは銀行口座をシミュレートする `BankAccount` というクラスを定義することを考えよう．ここでは銀行口座は残高と暗証番号という二つの内部状態と，引き出し，預け入れ，残高照会という三つの操作を持つものとして定義する．

```
class BankAccount

  def initialize(b, p)
    @balance = b
```

```
    @pass = p
  end

  attr_accessor :balance
  attr_accessor :pass

  def withdraw(num, amount)
    if self.pass == num
      if self.balance < amount
        puts "Insufficient fund."
      else
        self.balance = self.balance - amount
        puts self.balance.to_s
      end
    else
      puts "Incorrect pass."
    end
  end

  def deposit(amount)
    self.balance = self.balance + amount
    puts self.balance.to_s
  end

  def view_balance(num)
    if self.pass == num
      puts self.balance.to_s
    else
      puts "Incorrect pass."
    end
  end

end
```

class から最後の end までがクラスの定義である．

initialize はこのクラスのオブジェクトの作り方を指定している．@balance，@pass はこのクラスのオブジェクトが balance と pass という二つの内部状態を持つことを意味する．このクラスのオブジェクトを作るときは，例えば

```
acc = BankAccount.new(100, 1973)
```

と書く．このとき変数 acc は balance として 100，pass として 1973 を持つ BankAccount オブジェクトを指すことになる．

attr_accessor :balance，attr_accessor :pass という命令は内部状態 balance と pass を，同じ名前のメソッドによって参照できるようにする命令である．この命令によって例えば上のように定義された acc に対して，acc.balance によって，その balance を参照することが出来る．

withdraw は BankAccount に対するメソッドの一つで，預金の引き出しに対応する．withdraw のコードの中の self はこのメソッドが適用されるオブジェクトを指している．また self.balance はこのメソッドが適用されるインスタンスの balance を指している．例えば

```
acc.withdraw(1973, 50)
```

という命令によって，acc の balance が 50 減らされる．

deposit と view_balance はそれぞれ預け入れと残高照会の手続きに対応するメソッドである．

問題 8.1. height と width という内部状態を持つオブジェクトのクラス Rectangler を定義せよ．ただし height と width はともに整数を値とする．また Rectangler に対して，その面積を求めるメソッド area を定義せよ．

8.2 サブクラス

普通口座には利率に応じた利息がつく．そこで BankAccount を，利率という内部状態を加えることによって拡張することを考えよう．BankAccount 自身の定義を書き変えてもよいが，ここでは BankAccount のサブクラスとして BankAccountWithInterest というクラスを定義する．

Ruby では二つのクラスの間にはスーパークラス−サブクラスという関係をつけることが出来て，サブクラスに対してはスーパークラスに対して定義されたメソッドをそのまま適用することが出来る．クラス AAA をクラス BBB のサブクラスにするには AAA の定義の冒頭で

```
class AAA < BBB
```

と書けば良い．

次のように BankAccountWithInterest を定義しよう．

```
class BankAccountWithInterest < BankAccount

  def initialize(b, p, r)
    @balance = b
    @pass = p
    @rate = r
  end

  attr_accessor :rate

  def add_interest
    interest = ((self.balance * self.rate) / 100).round
    self.balance = self.balance + interest
    puts self.balance.to_s
  end
end
```

add_interest はその口座の利率 rate に基づいて，残高に利息を加えるメソッドである．なお round は浮動小数に適用され，小数点以下を四捨五入するメソッドである．この定義によって以下のような実行結果を得る．

```
acc = BankAccountWithInterest.new(1000, 1973, 2.5)
```

```
acc.addInterest
```
1025

```
acc.add_interest
```
1051

```
acc.add_interest
```
1077

また BankAccountWithInterest は BankAccount のサブクラスであるため BankAccount に対して定義されたメソッドを使うことも出来る．

```
acc.withdraw(1973, 100)
```
977

```
acc.deposit(50)
```
1027

9 Ruby で表計算

私たちは 6 節で，配列の作り方と操作の方法を見た．ここではそれらを表計算に応用してみよう．例えば次のような表を考えよう．

	English	Math	Biology	History
John	88	71	95	80
Peter	65	45	70	95
Mary	90	89	94	58
Jane	75	90	77	63

この表の数値の部分は次のような配列の配列として表現できる．

```
[
 [88, 71, 95, 80],
 [65, 45, 70, 95],
 [90, 89, 94, 58],
 [75, 90, 77, 63]
]
```

この表を以後 table1 として言及する．

以下の用語を導入する．

- **行**：表の横の並び．上から 0 行目，1 行目,... と呼ぶ．
- **列**：表の縦の並び．左から 0 列目，1 列目,... と呼ぶ．

- (m, n) の**エントリ**：表の m 行目 n 列目の要素．このとき (m, n) をそのエントリの**位置**と呼ぶ．任意の表 table に対して，(m, n) のエントリは table[m][n] によって参照することが出来る．

9.1 列の参照

表 table の n 行目は table[n] によって参照することが出来る．表の n 列目を参照するにはどうしたらよいだろう．それには表の各行から n 番目の要素を取りだして並べた配列を作るメソッドが必要である．これを次のように定義しよう．

```
def column(table, n)
  count = 0
  length = table.length
  list = []
  while count < length
    list.push(table[count][n])
    count = count + 1
  end
  list
end
```

9.2 簡単な表計算

各行の合計を計算し，合計からなる配列を返す手続きは次のように定義できる*18．

```
def table_sums(table)
  count = 0
  length = table.length
  sums = []
  while count < length
    sums.push(list_sum(table[count]))
    count = count + 1
  end
  sums
end
```

表の縦の列の合計を求めたい場合もあるだろう．そのためのメソッドを定義しても良いが，行と列に対していちいち別のメソッドを定義するのも面倒なので，表の行と列を入れ替えた新しい表を作るメソッド exchange_series を定義しよう．例えば exchange_series(table1) は

```
[
 [88, 65, 90, 75],
 [71, 45, 89, 90],
 [95, 70, 94, 77],
 [80, 95, 58, 63]
]
```

*18 list_sum に関しては??頁を参照．

という表を新しく作って返す.

```
def exchange_series(table)
  count = 0
  length = table[0].length
  new_table = []
  while count < length
    new_table.push(column(table,count)))
    count = count + 1
  end
  new_table
end
```

これを使えば表 `table` の縦の列の合計は `table_sums(exchange_series(table))` によって求めることが出来る.

問題 9.1. (1) 表の各行の平均点を求めるメソッド `table_aves` を定義しなさい.（ヒント：30 頁の `listAve` を使う.）
(2) 指定された列の数値だけを各行ごとに合計し，その値の配列を返すメソッド `table_designated_sums` を定義しなさい．例えば `tableDesignatedSums(table1,[0, 1])` は `[183, 135, 184, 152]` を返す.（ヒント：最初に配列の指定された要素だけを合計するメソッド `list_designated_sum` を定義して，これを使う.）

9.3 エントリの抽出

表の中で特定の条件を満たすエントリに興味を持つ場合があるだろう．例えば `table` において 60 未満（不合格）のエントリは $(1,1)$（ピーターの数学）と $(2,3)$（メアリーの歴史）である．このように，表の中で一定の数値より小さいエントリの位置を教えてくれるようなメソッドあれば便利である．これを次のように定義しよう.

```
def entries_less_than(table, n)
  count1 = 0
  length1 = table.length
  list = []
  while count1 < length1
    count2 = 0
    length2 = table[count1].length
    while count2 < length2
      if table[count1][count2] < n
        list.push([count1,count2])
      end
      count2 = count2 + 1
    end
    count1 = count1 + 1
  end
  list
end
```

`entries_less_than(table1, 60)` は `[[1,1],[2,3]]` を返す.

問題 9.2. (1) 上の例にならって entries_more_than，entries_equal_to，entries_between を定義しなさい．
(2) 各行の最大値を求め，その値の配列を返すメソッド table_maxs を定義しなさい．（ヒント：31 頁の max を使う．）
(3) 表の中で最大値を返すメソッド table_max を定義しなさい．さらに最大値になっているエントリの位置を返すメソッド max_entries を定義しなさい．

9.4 表を Html で書く

表を Html 形式の表に書き換えるメソッドを定義しよう．Html での表は例えば次のように書くことが出来る．

- <TABLE border=1>と</TABLE>の間に表を書く．
- <TR>と</TR>の間に一つの行を書く．
- <TD>と</TD>の間に一つのエントリを書く．

従って例えば

39	2	894
112	234	71

という表は

```
<TABLE border=1>
<TR><TD>39</TD><TD>2</TD><TD>56</TD></TR>
<TR><TD>112</TD><TD>234</TD><TD>71</TD></TR>
</TABLE>
```

という命令によって書かれる．

Ruby の表を Html での表に変換する命令は次にように定義できる[*19]．

```
def tableToHtml(table)
  count1 = 0
  length1 = table.length
  string = "<TABLE border=1>\n"
  while count1 < length1
    count2 = 0
    length2 = table[count1].length
    row = "<TR>"
    while count2 < length2
      row = row + "<TD>" + table[count1][count2].to_s + "</TD>"
      count2 = count2 + 1
```

[*19] \n は改行記号を表す．

```
    end
    string = string + row + "\n"
    count1 = count1 + 1
  end
  string + "</TABLE>"
end
```

問題 9.3. 表を $\TeX$ の表に変換するメソッド `tableToTeX` を定義しなさい.（注意：文字列として\を表示するには"\\"と書かなければならない.）

9.5 テキストファイルへの書き出し，テキストファイルからの読み込み

この節では，作成した表のデータを残しておき，またあとで利用する方法を解説する．そのために YAML という仕組みを利用する.

YAML とはプログラミング言語におけるオブジェクトを決まったテキスト形式で記述する方式である．Ruby のオブジェクトを YAML 形式に変換するには，まずファイルの先頭に `require 'yaml'` と書いておいて，YAML 形式に変換したいオブジェクトに `.to_yaml` をつけるだけである．例えば

```
require 'yaml'
puts([0,1,2].to_yaml)
```

と入力すると

```
---
- 0
- 1
- 2
```

と表示される.

これを `yaml_test.txt` という名前のテキストファイルとして保存しよう．これは次の命令によって実行できる.

```
text = [0, 1, 2].to_yaml
file_name = "yaml_test.txt"
File::open file_name, "w" do |f|
  f.write(text)
end
```

すでにカレント・ディレクトリに同じ名前のファイルが存在する場合にはそのファイルを `text` によって上書きすることになるので注意が必要である.

次に YAML 形式で記述された Ruby のオブジェクトを, 再び Ruby オブジェクトととして復元しよう．そのためには `YAML::load` を YAML 形式の文字列に適用すれば良い．例えば `YAML::load("--- \n- 0\n- 1")` という式の値は配列 `[0, 1]` になる.

テキストファイルに書かれている文字列を受け取るには `File::read` をテキストファイル（の名前を表す文

字列）に適用する．従って例えばカレント・ディレクトリに foo.txt という名前のファイルがあるとすれば,

```
read_string = File::read("foo.txt")
```

と書くと，変数 read_string は foo.txt の中身の文字列を表すことになる．

問題 9.4. (1) 37 頁の table1 を YAML 形式に変換して，テキストファイルに保存しなさい．またそれを読みこんで元の表を復元しなさい．
(2) ファイル名（文字列）とオブジェクトを受け取り，そのオブジェクトを YAML 形式に変換してテキストファイルを作成する手続き make_file_yaml を定義しなさい．