2020 年度 前期 コンピュータ 2

担当 和地 輝仁

目 次

1　シラバス抜粋

授業概要　プログラミング言語 Ruby の基礎を学び、現代的なプログラミングの基礎的な知識を知る授業です。

到達目標

1. スコープ、クラス、関数などプログラミング言語の基礎的な概念を理解する。
2. 条件分岐や繰り返しなどの制御構造を理解する。
3. 制御構造を活用してプログラムを作成できる。

授業計画　順序を交換する場合もあるので注意すること。

1. Ruby のインストール
2. 繰り返しと条件分岐
3. クラスの初歩
4. 数値のクラス
5. 文字列のクラス
6. 配列のクラス
7. 繰り返しと配列
8. 関数定義
9. 関数定義の例
10. 再帰
11. コンテナの要素の反復操作
12. 正規表現
13. 文字列操作
14. 数値計算
15. 期末試験

成績評価　期末試験では、Ruby スクリプトを用いて与えられた課題を解決できるかどうかを評価する (50%)。各回の授業では演習問題を出題し、その提出状況と内容も評価する (50%)。原則として全ての時間の出席を求めるが、やむを得ない理由で欠席をする (した) 場合はできるだけ速やかに申し出て、指示を受けること。

備考　受講するには、コンピュータ 1 の単位を修得していることが望ましいです。

2　2020 年度前期の遠隔授業

2020 年度前期は、当面遠隔授業で行います。次章「授業のノート (Ruby 編)」は、対面授業が再開することを見据えて例年のテキストをそのまま置いてありますが、遠隔授業では変更箇所が多数あります。ここでは、これら変更箇所や、その他の注意事項をまとめます。

§1　遠隔授業の進め方

コンピュータ 1 を受講した人であれば、同様と考えてよいです。つまり、

(1) テキストを読むなどして、その日に学ぶべきことを理解する。

(2) 対応する課題のプログラムを作成し、指定されたファイル名で保存する。

(3) `http://alg.kus.hokkyodai.ac.jp/comp2/comp2.html` から、コンピュータ 1 と同様に課題を提出する (コンピュータ 1 とは URL は異なる)。

(4) 提出状況確認や、提出物に対するコメントもコンピュータ 1 と同様に見られる。再提出する場合も、コンピュータ 1 と同様に再提出する。

という流れです。

§2　テキスト

この pdf ファイルがテキストです。各自印刷することは構いませんが、こちらから印刷したものを配布するのは、対面授業が始まってからです。印刷したテキストをもらう目的では、研究室に来ないで下さい。

§3　成績評価

提出課題 (50%) と期末試験 (50%) で成績評価します。期末試験の時期に遠隔授業が終了していない場合は、期末試験は 1 人ずつの口頭試問とし、対

面、あるいは、遠隔で行う予定です。期末試験を、レポートや追加の課題の提出で代替することはありません。遠隔授業では、対面授業と比べ、内容の理解が困難だとは思いますが、理解せずに先へ進むと口頭試問に耐えられないと思いますので、注意して下さい。

§4 Ruby の実行環境

コンピュータ 1 では、Small Basic というプログラミング言語を用いました。これは、コンピュータ室に備えてあるコンピュータにはインストール済みでしたし、テキストには、各自のコンピュータにインストールする方法も記しましたので、そうした人もいたでしょう。

コンピュータ 2 で用いる Ruby というプログラミング言語も、コンピュータ室のコンピュータにはインストール済みです。しかし、次章 「(1.2) 動作確認」のあたりに記したような、ドラッグアンドドロップでプログラムの実行をするためには、みなさんに多少の準備作業をしてもらう必要があります。この作業が、対面ではない状況で説明するには難しいと判断しました。

そこで、2020 年度前期のコンピュータ 2 では、遠隔授業を受けるみなさんは、別の方法で Ruby のプログラムを実行してもらうことにします。それは、インターネット接続が必要ですが、ウェブブラウザの上で Ruby を実行できるもので、主なものだけでも以下のようなサイトがあります。

- repl.it (この中では唯一の対話的実行が可能なサイト。ただ、人間の入力として日本語を受け付けない)
  `https://repl.it/languages/ruby`
- paiza.io (この中では唯一の日本語サイト)
  `https://paiza.io/ja/projects/new`
- Coding Ground at tutrialspoint
  `https://www.tutorialspoint.com/execute_ruby_online.php`
- Code Chef
  `https://www.codechef.com/ide`
- ideone.com
  `https://ideone.com`
- Wandbox
  `https://wandbox.org`
- jdoodle
  `https://www.jdoodle.com/execute-ruby-online/`
- Try It Online
  `https://tio.run/#ruby`
- run.rb (この中では唯一、ユーザーのコンピュータで実行するタイプだが、人間からの入力を受け付ける仕組みがない)
  `https://runrb.io`

これらのうち、最初の 2 つを解説します。1 つ目は「対話的」であることが絶対的な長所ですが、人間からの入力として日本語を受け付けません。2 つ目は日本語サイトであることが絶対的な長所ですが、対話的ではなく、さらに、実行結果が一部正しくないようです (以下では同じプログラムを実行しますが、結果が異なることが見てとれます)。非対話的なものを利用する場合も、2 つ目のサイトだけではなく、3 つ目のサイトなどを利用し、動作チェックをする必要があるかも知れません。

§5 repl.it

「repl.it」は英語のサイトではありますが、Ruby プログラムの対話的な実行が可能です。「対話的」とは何かを、コンピュータ 1 の Small Basic で学んだことを思い出して説明します。

大事なことを 1 回言ったり 2 回言ったりする実行例

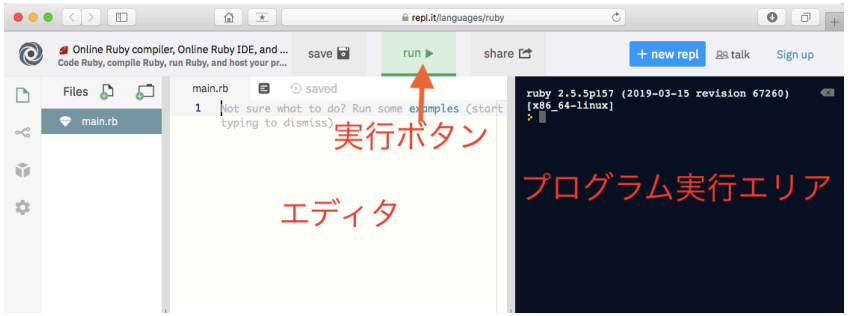
```
大事なことを入力
時は金なり
時は金なり
もっと大事なことを入力
大丈夫
大丈夫、大丈夫
```

ただし、上の実行例の 2、5 行目は、実行時に人間がキーボードから入力したものです。

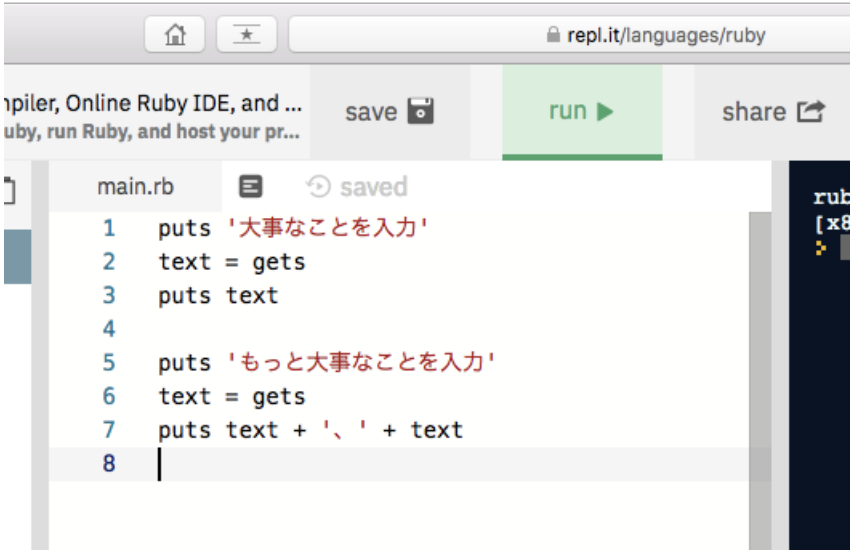
このプログラムを実行すると、1 行目の「大事なことを入力」が表示された時点で、入力待ちになります。人間が 2 行目に「時は金なり」と入力し Enter キーを押下すると、プログラムの実行が再開され、3 行目の「時は金なり」が表示されます。そのまま、続けて 4 行目の「もっと大事なことを入力」が表示された後に、再び入力待ちになり、人間が 5 行目に「大丈夫」と入力し Enter キーを押下すると、プログラムの実行が再開され、6 行目の「大丈夫、大丈夫」が表示されます。

このように、人間が入力すべき所でプログラムが一時停止する動作形態を「対話的」と呼びます。つまり、Small Basic の実行形態は対話的でした。コンピュータ室の Ruby 実行環境も対話的です。このテキストの次章以降も、対話的な実行環境を前提として書かれています。

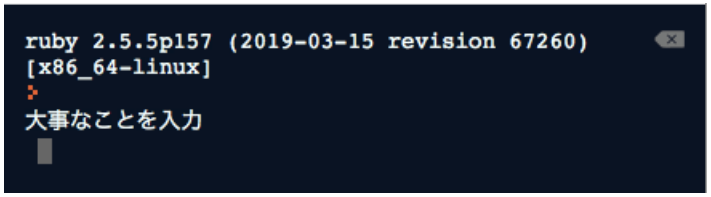
repl.it (`https://repl.it/languages/ruby`) のページへ行くと、下のような画面になります。



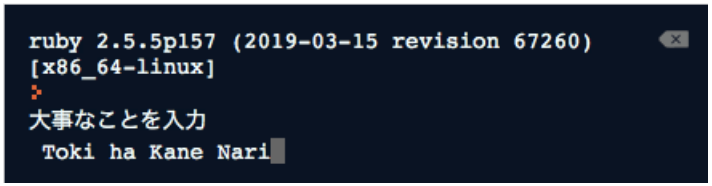
エディタに、次のようなプログラムを入力します。



実行ボタンをクリックすると、プログラム実行エリアは下の画面まで進み、入力待ちになります。



ここで、人間が「時は金なり」と入力したいのですが、repl.it では日本語で入力するとエラーになるので、「Toki ha Kane Nari」と入力し Enter キーを押下します。



すると、プログラム実行エリアは下の画面まで進み、再び入力待ちになります。

```
ruby 2.5.5p157 (2019-03-15 revision 67260)
[x86_64-linux]
:
大事なことを入力
 Toki ha Kane Nari
Toki ha Kane Nari
もっと大事なことを入力
```

ここで、人間が「Daijo-bu」と入力し Enter キーを押下します。

```
ruby 2.5.5p157 (2019-03-15 revision 67260)
[x86_64-linux]
:
大事なことを入力
 Toki ha Kane Nari
Toki ha Kane Nari
もっと大事なことを入力
 Daijo-bu
```

すると、プログラム実行エリアは下の画面まで進み、プログラムの実行が終了します。

```
ruby 2.5.5p157 (2019-03-15 revision 67260)
[x86_64-linux]
:
大事なことを入力
 Toki ha Kane Nari
Toki ha Kane Nari
もっと大事なことを入力
 Daijo-bu
Daijo-bu
、Daijo-bu
=> nil
:
```

§6 paiza.io

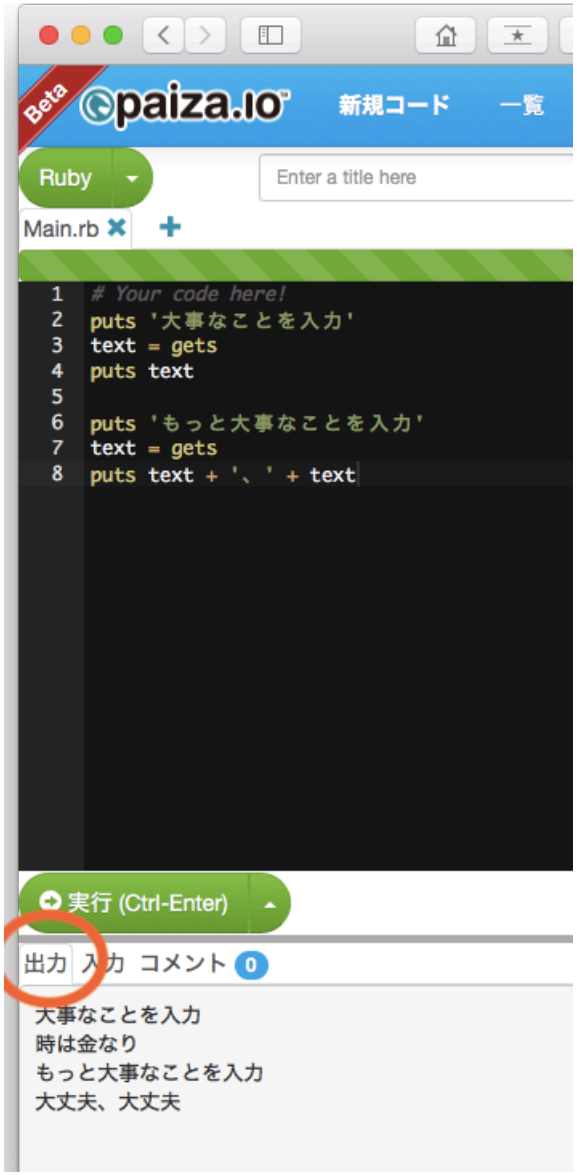
先に掲げたサイトのうち、「repl.it」以外は対話的ではありません。その場合、人間が何をどの順番で入力するかを、プログラム実行前に与えておく必要があります。対話的ではないけれど、日本語のサイトである paiza.io を例にとり、説明します。

paiza.io (`https://paiza.io/ja/projects/new`) のページへ行くと、下のような画面になります。プログラミング言語を Ruby に設定する必要があります。

エディタに、次のようなプログラムを入力します。

さらに、入力待ちになったときに人間が入力する予定の文字列を、ウィンドウ下方の「入力」タブに入力します。

実行ボタンをクリックすると、対話的な場合と異なり、人間の入力も自動的に行われ、ウィンドウ下方の「出力」タブに最終的な実行結果が表示され、プログラムの実行は終了します。



§7 ファイルの保存

オンラインの実行環境では、ファイルに保存するにはユーザー登録が必要なこともあります。もちろん登録しても良いですが、単にプログラムを全選択してコピーし、メモ帳などにペーストして保存するのが手軽です。

この時、ファイル名の特に拡張子に注意して下さい。メモ帳であれば、保存するときに、ドロップダウンメニューから「すべてのファイル」を選んでから、「0000 和地-01.rb」(0000 直後にスペースがあるように見えますが、ないです) のように、拡張子「.rb」で保存して下さい。そうしないと、「0000 和地-01.rb.txt」のような、二重に拡張子のついたファイル名になってしまいます。

反対に、再提出になった場合など、以前のファイルを再び実行したいときは、メモ帳等で当該のファイルを開いたあと、全選択してコピーし、「repl.it」などのサイトでペーストするのが手軽です。

§8 課題への取り組み

遠隔で受講する場合、このテキストを読み、まずは義務の課題を進めるのが良いでしょう。Small Basic の経験も生かせると思います。また、どのくらいの時期に、どこまでやっておけばよいかは、シラバスに記載しました。

テキストの解説は、動画の形でも提供するかも知れませんが、未定です。動画がある場合は、コンピュータ 2 の課題提出ページに、その旨掲載します。

遠隔授業という難しい形態ではありますが、積極的に課題に取り組んでくれることを楽しみにしています。

3 授業のノート (Ruby 編)

§1 Ruby のインストール

(1.1) **計算機室の ruby** この授業では ruby というプログラム言語を学習する。Windows で利用できる ruby は何種類かあるが、計算機室のコンピュータ

にインストールされているのは ActiveScriptRuby である。ActiveScriptRuby のページ (http://www.artonx.org/data/asr/) から、ウィンドウズ用のインストーラ[1]をダウンロードすれば、自宅等のコンピュータにも同じものをインストールすることができる。

また、Macintosh ならば、出荷時からインストール済みである。

(1.2) **動作確認** プログラムはメモ帳などで作成しても構わないが、専用のエディタを使うとソースコードに色付けされるなどの便利な機能がある。計算機室のコンピュータには、TeraPad エディタがインストールされていて、デスクトップにはそのアイコンがあるはずである。次のソースコードを TeraPad で作成し、例えば dosa.rb のように、拡張子を「.rb」で保存する[2]。

dosa.rb

```
1 puts "hello"
```

作成したプログラムを実行するには、dosa.rb を右クリックして現れるメニューから「プログラムから開く」を選び ruby を選択するだとか、ruby の実行ファイルに dosa.rb をドラッグアンドドロップするだとか、いろいろ方法があるが、不便であったり問題があったりもする[3]。講義ではバッチファイルを 1 つ用意して、それにドラッグアンドドロップすることにする。

[1] (Ruby-2.4.0 (i386-mswin32_100) Microsoft Installer Package (2016-12-24 revision 57164) のような名前のリンク)

[2] メモ帳でプログラムを作成するならば、保存するときに、「ファイルの種類」を「すべてのファイル」にしてから、拡張子を付加して保存することに注意。

[3] 日本語の文字化け、実行終了直後にウィンドウが閉じてしまう、カレントディレクトリが適切ではない、など

dosa.rb の実行例

```
hello
```

また、日本語 (いわゆる全角文字) を含むプログラムでは、文字コードによっては実行するとエラーが発生する。その場合、文字コードは「UTF8」にするとよい。

(1.3) **puts、コメント** puts (プットエス) は文字列を出力して改行するメソッド[4]である。また、「#」以降行末までは無視されるので、コメント (注釈) を書くなどできる。ruby では文字列は二重引用符で囲む[5]。

puts とコメントの説明

```
1 puts "hi!" # あいさつします
```

実行例

```
hi!
```

§2 繰り返しと条件分岐

(2.1) **for 文** for 文は繰り返しを実現する制御構造である。

for 文の文法

```
for <変数> in <オブジェクト>
  <処理> (複数行でもよい)
end
```

[4] ruby のメソッドは、C 言語などの「関数」、Small Basic の「サブルーチン」にあたる。

[5] 一重引用符も使えるが、二重引用符とは違いがある。§5 を参照。

〈オブジェクト〉[6] は each メソッド[7]に応答するものならば何でもよいが、典型的には Array 型や Range 型のオブジェクトが使われる。下の例では Range 型のオブジェクト (1..5) を用いている。

for 文の例

```
1 for i in (1..5)
2   puts i
3 end
```

上の例では、変数 i が 1 から 5 までの整数を変化しながら、2 行目の「puts i」を実行するので次のような出力になる。

実行例

```
1
2
3
4
5
```

下の例では Array 型のオブジェクト ["a", "bc", 4] を用いている。

for 文の例

```
1 for x in ["a", "bc", 4]
2   puts x
3 end
```

実行例

```
a
bc
4
```

[6]整数や文字列など、ruby で扱える「値」のこと。**オブジェクト**は整数 (Fixnum) や文字列 (String)、あるいは、未習だが範囲 (Range)、配列 (Array)、正規表現 (Regexp) などの**クラス**に属する。

[7]先に Small Basic のサブルーチンにあたると書いたが、ruby では「object.method(parameter)」のように、すべてのメソッド呼び出しはオブジェクトに対して行われる。each メソッドに応答できるとは、object.each(..) という呼び出しが可能な object のことを指す。

(2.2) **演算子** ruby で利用できる演算子[8]の主なものを下にあげる。表の上の方ほど優先順位が高い。つまり、1-2**3*4 は 1-((2**3)*4) と解釈される。また、商 (/) は、割る数、割られる数とも整数 (Fixnum) の場合は整数の商の意味であり、浮動小数点数 (Float) などのときは、商は小数になる可能性もある。このように、演算対象のオブジェクトの型 (クラス) によって演算子の動作が変わる場合があるので注意が必要である。

演算子	説明
[]	配列要素へのアクセス
!	論理否定 (単項演算子)
**	ベキ
-	負号 (単項演算子)
* / %	積、商、整数の剰余
+ -	和、差
> >= < <=	大小の比較演算子
== != =~ !~	等値性の比較演算子、正規表現のマッチ
&&	論理積 (かつ)
||	論理和 (または)
=	代入。自己代入も (+= -=など)

(2.3) **課題 01 – ベキの計算** 自分の学生番号の 1 乗から 10 乗までを表示するプログラムを作成し提出せよ。ただし、学生番号の先頭が 0 の場合「0012」などとはせずに、「12」としてプログラムに記述すること。

学生番号「1234」の場合の課題 01 の実行例

```
1234
1522756
1879080904
2318785835536
2861381721051424
```

[8]普段は意識しなくてよいが、a<20 のような記法は実は糖衣構文であり、ruby では演算子すらメソッドである。したがって例外を除けば演算子を再定義して動作を変更することすらできる。

```
3530945043777457216
4357186184021382204544
5376767751082385640407296
6634931404835663880262603264
8187505353567209228244052427776
```

(2.4) **if 文その 1** if 文は条件分岐を実現する制御構造である。

if 文の文法その 1

```
if <式>
  <処理1>（処理は複数行でもよい）
else      （elseの部分は省略可能）
  <処理2>
end
```

〈式〉が真[9] ならば〈処理 1〉を、偽ならば〈処理 2〉を実行する。else の部分は省略可能である。

if 文の例

```
1 year = 2012
2 if year % 4 == 0 # 本当は違う
3   puts "うるう年"
4 end
```

実行例

```
うるう年
```

「かつ」や「または」に相当する演算子も用いると、次のような if 文も書ける。

if 文の例

```
1 hour = 23
2 if hour >= 22 || hour <= 7
3   puts "睡眠中"
4 end
5 if hour > 7 && hour < 22
6   puts "起床中"
7 end
```

実行例

```
睡眠中
```

else 付き if 文の例

```
1 hour = 10
2 if hour >= 22 || hour <= 7
3   puts "睡眠中"
4 else
5   puts "起床中"
6 end
```

実行例

```
起床中
```

(2.5) **課題 02 − 3 の倍数かつ 4 の倍数ではないとき** 1 から 50 までの整数を表示するが、整数が 3 の倍数であり、かつ、4 の倍数ではないときは「さーん」と表示するプログラムを作成し、「0000 和地-02.rb」のようなファイル名 (学生番号、名前、課題番号) で提出せよ。

課題 02 の実行例

```
1
2
さーん
4
5
さーん
7
8
さーん
```

[9] ruby で式が「偽」であるとは、式が NilClass のオブジェクト nil であるか、または、FalseClass のオブジェクト false であることを言う。式が「真」であるとは「偽」ではないことを言う。

```
10
11
12
13
:
:
48
49
50
```

(2.6) **if 文その 2** if 文には elsif を記述できる。

if 文の文法その 2

```
if <式1>
  <処理1>
elsif <式2> (elsifは省略可能。複数あってもよい)
  <処理2>
elsif <式3>
  <処理3>
else          (elseの部分は省略可能)
  <処理n>
end
```

〈式 1〉が真ならば〈処理 1〉を実行し、そうではないとき、〈式 2〉が真ならば〈処理 2〉を実行し、そうではないとき、〈式 3〉が真ならば〈処理 3〉を実行し、if と elsif に書かれたすべての式が偽であったら〈処理 n〉を実行する。

elsif 付き if 文の例

```
1 point = 95
2 if point >= 90
3   puts "A"
4 elsif point >= 60
5   puts "BCD"
6 else
7   puts "F"
8 end
```

実行例

```
A
```

しかし elsif を用いずに以下のようにしてしまうと、意図に反する結果になる。それは、2 行目の if で条件が真になった後も、6 行目の if に到達してしまうからである。このような場合は先の例のように elsif を用いる方が簡潔に書ける。

意図に反する if 文の例

```
 1 point = 95
 2 if point >= 90
 3   puts "A"
 4 end
 5
 6 if point >= 60
 7   puts "BCD"
 8 end
 9
10 if point < 60
11   puts "F"
12 end
```

実行例

```
A
BCD
```

(2.7) **課題 03 − FizzBuzz** 次のような出力をするプログラムを作成し、「0000 和地-03.rb」のようなファイル名 (学生番号、名前、課題番号) で提出せよ。つまり、i 番目 ($i = 1, 2, \ldots, 100$) の行では、i が 3 の倍数ならば Fizz、5 の倍数ならば Buzz、ただし 15 の倍数ならば FizzBuzz と表示し、それ以外では i を表示するプログラムを作成せよ。

課題 03 の実行例

```
1
```

```
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
11
Fizz
13
14
FizzBuzz
16
:
:
98
Fizz
Buzz
```

§3 クラスの初歩

(3.1) **クラスとは** ruby のオブジェクトは必ずクラス[10]に属する。クラスとは同種のオブジェクトの総称のような意味である。例えば、整数「1」は Fixnum クラスのオブジェクト、文字列「"2"」は String クラスのオブジェクトである。

クラス	説明
Fixnum	(古めの環境では) 31 ビット以内の整数のクラス
Bignum	31 ビットを超える整数のクラス。Fixnum との変換は自動的
Float	浮動小数点数のクラス
Rational	有理数のクラス
Complex	複素数のクラス
String	文字列のクラス
Array	配列のクラス
Hash	ハッシュのクラス
Range	範囲オブジェクトのクラス
NilClass	nil (偽) のクラス
FalseClass	false (偽) のクラス
TrueClass	true (真) のクラス
Regexp	正規表現のクラス
IO	基本的な入出力のためのクラス
File	ファイルアクセスのためのクラス

今のところ意識しなくても問題はないが、クラスには継承関係[11]がある。Ruby では、例えば、Fixnum クラスのスーパークラスは Integer クラスであり、Fixnum クラスのオブジェクトは、Integer クラスに対して定義されているメソッドにも応答する。

オブジェクトは属するクラスのスーパークラスのメソッドを利用できるが、その他にも**モジュール**をインクルードすると、そのモジュールに定義されているメソッドも利用できるようになる。特に重要なものを以下にあげる。

[10] オブジェクト指向言語の多くは (1970 年代の Smalltalk や C++からして)、クラスシステムを備える。クラスシステムはオブジェクト指向言語に必須の要素であると誤解されたり、オブジェクト指向言語の特徴をあげてもクラスシステムの特徴をあげるに留まりがちである。しかし、ruby を ruby たらしめているのはメソッドの動的束縛である。for 文の項では、「each メソッドに応答するオブジェクト」という言い方をしたが、他の言語ならば「Array や Range クラスのオブジェクト」となるところである。つまり、ruby では、どのクラスに属するかは重要ではなく、どのメソッドに応答するかが重要である。オブジェクトの性質を知るのに、どのクラスに属するかではなく、どのメソッドに応答するかを手掛かりにする ruby のような方式を**ダックタイピング**と呼ぶ。

[11] 例えば、「正方形クラス」と「長方形クラス」があったとすると、正方形は長方形でもあるから、このようなとき、正方形クラスは長方形クラスの**サブクラス**であると言い、逆に、長方形クラスは正方形クラスの**スーパークラス**であると言う。

モジュール	説明
Kernel	すべてのクラスから参照できるメソッドを定義している
Enumerable	繰り返しを行うクラスのためのモジュール

(3.2) **class メソッド** Ruby のオブジェクトの属するクラスを知るには、class メソッドを用いる。

class の例

```
1 puts 1.class, "2".class
```

実行例

```
Fixnim
String
```

(3.3) **to_i と to_s** 「1」と「"2"」はクラスが Fixnum と String で異なるため和を計算できない。String クラスのメソッド to_i を用いると、整数に変換される。下の例で、単に、a+b としても型エラーになる。

to_i の例

```
1 a = 1
2 b = "2"
3 puts a + b.to_i
```

実行例

```
3
```

逆に、Fixnum クラスのオブジェクトは to_s メソッドで String 型に変換できる。下の例では、String オブジェクトに対する演算子+は文字列の連結をする ((5.2) 参照)。

to_s の例

```
1 a = 1
2 b = "2"
3 puts a.to_s + b
```

実行例

```
12
```

§4 数値のクラス (Fixnum, Bignum, Float)

(4.1) **Fixnum と Bignum** Ruby で扱える整数には桁数の制限がない。しかし、内部的には比較的小さい Fixnum クラスと、比較的大きい Bignum クラスの 2 つが使われている。ただ、これらの変換は自動的に行われているので普段は意識する必要はない。

Fixnum と Bignum の例

```
1 a = 2**62
2 b = a-1
3 puts a, b, a.class, b.class
```

実行例

```
4611686018427387904
4611686018427387903
Bignum
Fixnum
```

環境により結果が異なるが、上の結果は ruby 1.9.3p0 (2011-10-30) [x86_64-darwin11.3.0] のものである。

(4.2) **Float クラスと数値の演算** 2.0 など小数点を含む数値は浮動小数点数のクラス Float に属する。整数を Float に変換するときは、to_f メソッドを用いる。その逆は to_i メソッドである。和や商など Fixnum と Float に共通する演算は多いが、クラスによって結果が異なる。具体的には、整数どうしならば結果も整数、Float が混じれば結果も Float である。

Float の演算の例

```
1 puts 2.class, 2.0.class, 2.to_f.class
2 a = 5/2
3 b = 5/2.0
4 puts a, b
```

実行例

```
Fixnum
Float
Float
2
2.5
```

(4.3) **数値のクラスの主なメソッド** 数値オブジェクトが応答できる主なメソッドをあげる。

式	結果	式	結果	説明
5.6.abs	5.6	-5.6.abs	5.6	絶対値
5.6.floor	5	-5.6.floor	-6	$-\infty$ 方向への丸め
5.6.ceil	6	-5.6.ceil	-5	$+\infty$ 方向への丸め
5.6.round	6	-5.6.round	-6	四捨五入
5.6.to_i	5	-5.6.to_i	-5	0 方向への丸め

(4.4) **Float の誤差** Float クラスには、10 進法と 2 進法の変換に起因する誤差がある。下のように、ruby 内部では 0.3 ではないのに、表示では丸められて 0.3 となる場合もある。このような場合を警戒して、Float を比較するときは==や!=を用いるのではなく、差の絶対値が十分小さいかどうかを調べるべきである。

Float の誤差の例

```
1 a = 0.1 + 0.1
2 b = 0.1 + 0.2
3 puts a, b
4 puts a == 0.2
5 puts b == 0.3
```

実行例

```
0.2
0.3 # ここは環境によって異なります
true
false
```

§5 文字列のクラス (String)

(5.1) **String のリテラル**[12] "abc"のように二重引用符で囲んでもよいし、'def' のように一重引用符で囲んでもよい。両者の違いは式展開[13] と、バックスラッシュ記法が使えるかどうかである。バックスラッシュ記法を用いると、通常キーボードから入力できない文字を指定できる。主なものを以下にあげる。

記法	式	結果	説明
\n	"a\nb"	a, 0x0a, b	改行文字 (0x0a)
\"	"a\"b"	a, ", b	二重引用符
\'	"a\'b"	a, ', b	一重引用符
\\	"a\\b"	a, \, b	バックスラッシュ

一重引用符で囲んだ String のオブジェクトの場合、使えるバックスラッシュ記法は一重引用符とバックスラッシュのみである。

[12] リテラルとは、プログラム中に直接値を記述したもののことである。Fixnum のリテラル 1, Float のリテラル 2.3, Array のリテラル ["a", "bc", 4] は既に見た。"abc" が String のリテラルである。また有理数 (Rational) のようにリテラルがないものもある。

[13] 二重引用符の場合は、"ab#{1+2}cd" のように#{...} で囲んだ部分が、評価され、to_s されて埋め込まれる。したがって、この文字列は"ab3cd"となる。

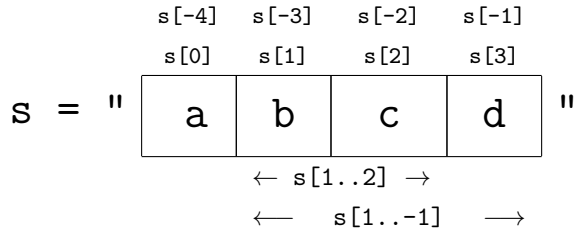
(5.2) **String の演算** String のオブジェクトに対して可能な演算のうち主なものをあげる。この他、!=, >, <=, >= も利用できる。

式	結果	説明
`"abc" + "def"`	`"abcdef"`	連結
`"abc" * 3`	`"abcabcabc"`	反復
`"abc" == "abb"`	`false`	等値
`"abc" < "abb"`	`false`	辞書順で比較
`"ab%s" % 3`	`"ab3"`	フォーマット

上記のうちフォーマットの項は、文字列中の%s を 3 で埋めるという演算である。詳細は後述する。

(5.3) **部分文字列** s を String のオブジェクトとするとき、その部分文字列を得るには次のような方法がある。先頭から何番目の文字かを数えるときは、0 開始である。また、s[0] の形式は、ruby 1.8 やそれ以前のバージョンでは結果が異なるので要注意である。

式	結果	説明
`s`	`"abcd"`	
`s[0]`	`"a"`	0 番目の文字 (ruby 1.9 以降)
`s[1..2]`	`"bc"`	1 番目から 2 番目の文字
`s[1..-1]`	`"bcd"`	$-n$ は後ろから n 番目を表す



また、部分文字列に別の文字列を代入することもできる。長さが異なる文字列を代入することもできる。

部分文字列への代入の例

```
1 s = "abcd"
2 s[3] = "x"
3 puts s
4 s[1..2] = "yzw"
5 puts s
```

実行例

```
abcx
ayzwx
```

(5.4) **String クラスの基本的なメソッド** String のオブジェクトが応答できるメソッドのうち、基本的なものをあげる。

式	結果	説明
`"abc".size`	`3`	文字数
`"abc".reverse`	`"cba"`	逆順
`"abc".index("b")`	`1`	部分文字列を検索して一致した位置

reverse メソッドは**破壊的**ではない[14] ことに注意が必要である。つまり、元の文字列はそのまま変化しない。他方、部分文字列への代入は破壊的メソッドであった。

reverse の例

```
1 s = "abc"
2 t = s.reverse
3 puts s, t
```

実行例

```
abc
cba
```

[14] 対応する破壊的メソッドが用意されていることがあり、その場合末尾に!が付加されたメソッド名であることが多い。例えば、reverse!メソッドは破壊的であり、s.revese!すると s が変化する。ただし、破壊的メソッドを積極的に利用する理由はない。

index メソッドにおいて、検索して見付からない場合は nil が返る。また、部分文字列を検索する、より強力な方法は、正規表現を用いる方法である (§13 参照)。

index の例

```
1 s = "abcac"
2 puts s.index("a"), s.index("bc"), s.index("cb")
3 t = ("123".to_i / 3).to_s.reverse.index("4")
4 puts t
```

実行例

```
0
1

1
```

nil は puts(プットエス) すると空文字列 ("") が表示されるため、上の出力の 3 行目は空行になっている。

(5.5) **課題 04 – 条件を満たす数の探索** 4 桁の整数であって、逆から読んでも自分自身と同じであるものを全て表示するプログラムを作成し提出せよ。

実行例

```
1001
1111
1221
:
(中略)
:
9779
9889
9999
```

(5.6) **文字列出力と入力 (print, p, gets)** 関数 puts(プットエス) は文字列を出力したあと改行したが、代わりに関数 print を用いると改行しない。puts(プットエス) と print は、大体、Small Basic の TextWindow.WriteLine と TextWindow.Write に対応する。また、下の例のように微妙に挙動が異なる。関数 p も文字列を出力し改行するが、オブジェクトを引数に渡すと、人間が読み易い形式で出力する[15]。

print と p の例

```
1 print [1, "2", "a"] # 改行しない
2 puts [1, "2", "a"]  # 各要素を puts する
3 p [1, "2", "a"]     # 見易い。改行する
```

実行例

```
[1, "2", "a"]1
2
a
[1, "2", "a"]
```

gets(ゲットエス) は文字列を 1 行キーボードから入力する組み込みの関数である (Small Basic の TextWindow.Read に相当)。返り値は String のオブジェクトで、末尾に改行文字 (\n) が付加されている。末尾の改行文字を除去したいときは、String クラスの chomp メソッドを用いるとよい。

gets の例

```
1 s = gets
2 puts s
3 p s
4 p s.chomp
```

p を用いると、puts(プットエス) や print では見えなくなってしまう、文字列中の改行文字 (\n) なども、下のように表示される。

abc と入力したときの実行例

```
abc
abc
```

[15] [1, 2, "a"] は少し後に学ぶ配列である。

```
"abc\n"
"abc"
```

ただし、実行例の最初の 1 行は人間がキーボードから入力したものである。

§6 真偽値のクラス (TrueClass, FalseClass, NilClass)

(6.1) **true, false, nil** Ruby で真と偽を意味する組み込みの値、true と false がある。これらは、TrueClass と FalseClass の、それぞれ唯一のオブジェクトである。

既に学んだ nil は NilClass の唯一のオブジェクトであり、これも偽として扱われる。例えば、String クラスの index メソッドで検索が不成功だったときなど、偽を表す false とはやや違う「失敗」の意味を持つ場合に、「偽」として扱われる返り値となることが多い。

Ruby では、「偽」として扱われるのは false と nil のみであり、他のすべてのオブジェクト「真」として扱われる。例えば、整数の 1 も真として扱われる。実は、if 文の条件の箇所[16]には、どんな式を置いても構わない。a < b という条件を表す式も、単に、true または false を返す式にすぎない。

真偽値の例

```
1 puts 1 < 2
2 puts 1 == 2
3 if 3 # false と nil 以外は真
4   puts "真"
5 end
```

実行例

```
true
false
真
```

[16] if 文に限らず、条件を書くべき箇所は、実はどんな式も受け付ける

(6.2) **課題 05 – 入力と真偽値** キーボードから文字列を 2 つ入力させて、両者の長さ (文字数) が一致すれば true を、一致しなければ false を出力するプログラムを作成し提出せよ。ただし、if 文は用いないこと。

実行例

```
small
basic
true
```

ただし、実行例の最初の 2 行は人間がキーボードから入力したものである。

§7 配列のクラス (Array)

(7.1) **配列のリテラル** 配列とはオブジェクトを 1 列に並べたデータ構造である。角かっこで囲み、各要素はカンマで区切り、[1, 2] とか、[1, 2, "ab", nil] のようにして[17]記述する。1 つも要素を持たない配列は [] となる。

(7.2) **Array の演算** Array のオブジェクトに対して可能な演算のうち主なものをあげる。この他、辞書順で比較する演算として、!=, >, <=, >= も利用でき、また、集合算として、| (和集合)、- (差集合) も利用できる

式	結果	説明
[1, 2] + [8, 9]	[1, 2, 8, 9]	連結
[1, 2] * 3	[1, 2, 1, 2, 1, 2]	反復
[1, 2] == [1, 3]	false	等値
[1, 2] < [1, 3]	true	辞書順で比較
[1, 2] & [1, 3]	[1]	共通部分

[17] [1,2,] のように最後の要素の後にカンマを付けてもよい。例えば、

```
a = [
  1,
  2,
]
```

と 1 行に 1 要素を記述したとき、要素の順序を交換してもカンマの付けはずしが不要になる。

(7.3) **配列の文字列への変換** to_s や join というメソッドで String のオブジェクトに変換できる。

式	結果	説明
`[1, 2, 3].to_s`	`"[1, 2, 3]"`	String に変換
`[1, 2, 3].join("--")`	`"1--2--3"`	間に文字列を挟む

配列の出力の例

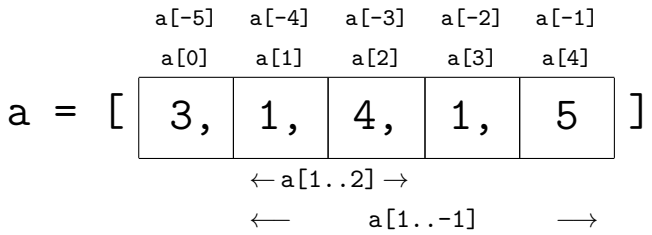
```
1 a = [1, "2"]
2 puts a      # 各要素ごとに改行
3 print a     # 1行に出力して改行しない
4 p a         # 1行に出力して改行する
5 puts a.to_s
6 puts a.join(",")
```

実行例

```
1
2
[1, "2"][1, "2"]
[1, "2"]
1,2
```

(7.4) **部分配列** a を Array のオブジェクトとするとき、その部分配列を得るには次のような方法がある。先頭から何番目の要素かを数えるときは、0 開始である。

式	結果	説明
`a`	`[3, 1, 4, 1, 5]`	
`a[0]`	`3`	0 番目の要素
`a[1..2]`	`[1, 4]`	1 番目から 2 番目の要素
`a[1..-1]`	`[1, 4, 1, 5]`	$-n$ は後ろから n 番目を表す



また、配列の一部を別の要素で置き換えることもできる。配列の一部の範囲を複数の要素で置き換えるときは、右辺の要素をカンマで区切る。こういった要素の置き換えは破壊的メソッドである。

配列の一部の置き換えの例

```
1 a = [1, 9, 9, 5, 2, 2, 4]
2 a[0] = 9
3 p a
4 a[4..6] = 1, 0
5 p a
```

実行例

```
[9, 9, 9, 5, 2, 2, 4]
[9, 9, 9, 5, 1, 0]
```

(7.5) **Array クラスの基本的なメソッド** Array のオブジェクトが応答できるメソッドのうち、基本的なものをあげる。

式	結果	説明
`[1, 9, 9, 5].size`	`4`	要素数
`[1, 9, 9, 5].reverse`	`[5, 9, 9, 1]`	逆順
`[1, 9, 9, 5].index(9)`	`1`	最初に出現する位置
`[1, 9, 9, 5].sort`	`[1, 5, 9, 9]`	整列
`[1, 9, 9, 5].max`	`9`	最大値
`[1, 9, 9, 5].uniq`	`[1,9,5]`	重複要素を省略

String のときと同様に、reverse メソッドは**破壊的**ではないことに注意が必要である。さらに、sort と uniq も破壊的ではない。つまり、元の配列はそのまま変化しない。また、最小値は min メソッドで求められる。

reverse, sort, uniq の例

```
1 a = [1, 9, 9, 5]
2 b = a.reverse
3 c = a.sort
4 d = a.uniq
5 p a, b, c, d
```

実行例

```
[1, 9, 9, 5]
[5, 9, 9, 1]
[1, 5, 9, 9]
[1, 9, 5]
```

String のときと同様に、index メソッドで要素が見付からない場合は nil が返る。

(7.6) **配列の末尾と先頭への追加と取り出し** push と pop の 2 つのメソッドは、それぞれ、配列の**末尾**への追加と取り出しを行う。どちらも破壊的、つまり、実行後に元の配列が変更される。push メソッドの返り値は (変更された) 配列自身であり、pop メソッドの返り値は取り出した要素である。

push, pop の例

```
1 a = [1, 2, 3]
2 a.push(9)
3 p a
4 x = a.pop
5 p a, x
```

実行例

```
[1, 2, 3, 9]
[1, 2, 3]
9
```

push と for 文を組み合わせると、配列の構築に使える。

push を用いた配列の構築

```
1 a = []
2 for i in (1..4)
3   str = gets.chomp
4   a.push(str)
5 end
6 p a
```

実行例

```
abc
1
2
def
["abc", "1", "2", "def"]
```

ただし、実行例の最初の 4 行は人間がキーボードから入力したものである。

また、unshift と shift の 2 つのメソッドは、それぞれ、配列の**先頭**への追加と取り出しを行う。どちらも破壊的、つまり、実行後に元の配列が変更される。unshift メソッドの返り値は (変更された) 配列自身であり、shift メソッドの返り値は取り出した要素である。

(7.7) **課題 06 – 配列の構築** キーボードから文字列を 4 つ入力させて、辞書順に出力するプログラムを作成し提出せよ。

実行例

```
Hokkaido
University
Education
Kushiro
Education
Hokkaido
Kushiro
University
```

ただし、実行例の最初の 4 行は人間がキーボードから入力したものである。

(7.8) **要素の削除** delete_at メソッドは指定した位置の要素を削除し、delete メソッドは指定した要素と等しい要素をすべて削除する。どちらも破壊的なメソッドである。また、返り値についてはどちらのメソッドも、要素を実際に削除できたらその要素を返し、そうでなければ nil を返す。

delete_at, delete の例

```
1 a = [1, 9, 9, 5, 2, 2, 4]
2 x = a.delete_at 3
3 p a, x
4 x = a.delete 2
5 p a, x
```

実行例

```
[1, 9, 9, 2, 2, 4]
5
[1, 9, 9, 4]
2
```

§8 繰り返しと配列 (Array)・範囲 (Range)

(8.1) **配列の上を繰り返す for 文** 既に学んだ for 文

for 文の文法

```
for <変数> in <オブジェクト>
  <処理>（複数行でもよい）
end
```

の〈オブジェクト〉には、(1..10) のようなオブジェクトの他に、Array のオブジェクトも利用できる。

for 文の例

```
1 for x in [2, "4", 6]
2   print x.class, " "
3   puts x
4 end
```

上の例では、変数 x が [2, "4", 6] の各要素を変化しながら、2 行目の「print x.class, " "」と 3 行目の「puts x」を実行するので次のような出力になる。

実行例

```
Fixnum 2
String 4
Fixnum 6
```

(8.2) **課題 07 – 配列** キーボードから文字列を 4 つ入力させて、辞書順の逆順に出力し、さらに、表示するときの行頭に、文字数も出力するプログラムを作成し提出せよ。

実行例

```
Hokkaido
University
Education
Kushiro
10 University
7 Kushiro
8 Hokkaido
9 Education
```

ただし、実行例の最初の 4 行は人間がキーボードから入力したものである。

(8.3) **範囲 (Range)*** for 文にも用いた (1..10) は、範囲を表す Range クラスのオブジェクトである。

Range のリテラルの文法

```
(a..b)
(a...b)
```

どちらも a から b までの範囲を表すが、(a..b) は終点を範囲に含み、(a...b) は終点を範囲に含まない。a や b は、Fixnum や String などのオブジェクト[18]を使用できる。

Range のオブジェクトは to_a メソッドで配列に変換できる。

to_a の例

```
1 r = (3...6)
2 a = r.to_a
3 p a
```

実行例

```
[3, 4, 5]
```

(8.4) **while 文** for 文は反復回数が事前にわかっている繰り返し構造だが、指定された回数で終了するのではなく、条件によって繰り返しを終了するような繰り返し構造が while 文である。

while 文の文法

```
while <式>
  <処理>（複数行でもよい）
end
```

まず繰り返しの先頭で〈式〉を評価して、偽ならば繰り返しを終了する。真ならば〈処理〉を実行し、再び 1 行目に戻って〈式〉を評価して偽ならば繰り返しを終了する。こうして、〈式〉が偽になるまで〈処理〉を繰り返し実行する。

while 文の例

```
1 a = 0
2 while a**2 < 10
3   puts a
4   a = a + 1
5 end
```

上の例では、変数 a をまず 0 に設定してから、a の 2 乗が 10 未満の間、a を表示して 1 加算することを繰り返すので、次のような出力になる。

実行例

```
0
1
2
3
```

(8.5) **break 文** break 文は、for や while の繰り返し構造から強制的に脱出します。

break 文の例

```
1 texts = []
2 while true
3   s = gets
4   if s == "\n" # Enterのみ入力した場合
5     break
6   end
7   texts.push s
8 end
9 p texts
```

上の例では、while 文の条件式が true で常に真なので、ここの判定で繰り返しを終了することはないが、gets（ゲットエス）で何もタイプせずに Enter を入力したとき[19]に、break 文で while ループを脱出する。

実行例

```
123
```

[18] 正確には、<などで互いに比較可能なオブジェクトであって、succ メソッドに応答するオブジェクトである。

[19] gets（ゲットエス）によりキーボードから文字列を入力したときは、文字列末尾が必ず改行文字 (0x0a) になるため、何もタイプせずに Enter を入力したときは、gets（ゲットエス）の返り値は改行文字 1 文字からなる String のオブジェクトである。

```
ab
xyz

["123\n", "ab\n", "xyz\n"]
```

ただし、実行例の最初の 4 行は人間がキーボードから入力したものである。

(8.6) **無限ループ** 無限ループ `while true ... end` は、`loop {...}` とも書くことができる。

`loop` 文の例

```
1 loop {
2   s = gets.chomp
3   if s == ""
4     break
5   end
6   puts s.size
7 }
8 puts "おわり"
```

実行例

```
123
3
ab
2

おわり
```

ただし、実行例の 1 行目、3 行目、5 行目は人間がキーボードから入力したものである。

(8.7) **課題 08 – 入力データを逆順に表示** 文字列を繰り返しキーボードから入力させ、Enter のみ入力されたときに終了する。入力がすべて完了した後に、文字列を入力されたのとは逆順に表示するプログラムを作成し提出せよ。

`a, Enter, b, Enter, c, Enter, Enter` と入力したときの実行例

```
a
b
c

c
b
a
```

ただし、実行例の最初の 4 行は人間がキーボードから入力したものである。

(8.8) **課題 09 – 入力データが重複を除いていくつあるか表示** 文字列を繰り返しキーボードから入力させ、Enter のみが入力されるまで続ける。入力がすべて完了した後に、入力された文字列から重複したものを除くといくつのデータがあったか表示するプログラムを作成し提出せよ。ただし最後の Enter のみの入力は、個数に含めない。

`a, Enter, b, Enter, a, Enter, Enter` と入力したときの実行例

```
a
b
a

2
```

ただし、実行例の最初の 4 行は人間がキーボードから入力したものである。

(8.9) **課題 10 – 入力データの最大値** 数値を繰り返しキーボードから入力させ、Enter のみが入力されるまで続ける。その後、入力された数値の最大値を表示するプログラムを作成し提出せよ。配列の最大値を求める `max` メソッドがあるが、用いても用いなくてもよい。

実行例

```
11
2
-3

11
```

ただし、実行例の最初の 4 行は人間がキーボードから入力したものである。

§9 関数定義

(9.1) **関数定義の利点** 何度も同じ処理をする場合や、まとまった意味のある処理をする場合は、その処理を行う**関数**を定義するとよい。まず、同じコードを何度も書くことが避けられるから、プログラムが短くなる利点がある。その処理を変更する場合も関数定義の部分だけ修正すればよく、作業量が減る利点や、修正し忘れによるミスを防ぐ利点もある。意味のある処理のまとまりで関数を定義すれば、(例えば関数名を見るだけで) 何の処理をしているかすぐにわかり、プログラムの処理内容を把握しやすくなる利点がある。**モジュール**化[20]できるという利点もある。つまり、一旦関数を定義してしまえば、どういう入力に対してどういう返り値が返るか (インターフェイス) だけ知れば関数を利用でき、インターフェイスを変更しない限りは、関数を改良してもプログラムの他の部分には影響を与えない。変数名の衝突も気にしなくてよいので、他のプログラムで関数を再利用することができる。

(9.2) **関数定義の方法** ここで定義する関数は、ruby では「関数形式で呼び出せるメソッド」である。つまり、ruby では関数もメソッドである。

関数定義の文法

```
def <関数名>(<仮引数1>, <仮引数2>, ...)
  <処理> (複数行でもよい)
end
```

〈仮引数(かりひきすう)〉[21]はひとつもなくてもよく、その場合は丸かっこの対も不要である。

[20] (3.1) で述べた、Enumerable モジュールなどの ruby の組み込みクラスの Module の意味ではなく、まとまった機能を持つ単位という一般名詞的な意味である。

[21] 引数(ひきすう)とは関数に渡されるパラメータのことである。関数定義中での引数は、まだ実行時の値が決まっていないというような意味で、仮引数と呼ばれ、下の例の puts1(3) の 3 のような、実行時関数に渡されるものを実引数と呼ぶ。ただし、両方とも単に引数と呼んでも実害はあまりない。

関数定義と呼び出しの例

```
1 def puts1(n)
2   puts n+1
3 end
4
5 puts1 2
6 puts1(3)
```

上の例では、引数 n に 1 を加えたものを表示する関数 puts1 を定義している。それを利用して、引数に 2 を渡して呼び出し、次に引数に 3 を渡して呼び出している[22]。

実行例

```
3
4
```

(9.3) **関数の返り値 (戻り値)** 擬似乱数を返す ruby の組み込みの関数 rand がある。m が正整数のとき、rand(m) は 0 以上 m 未満の整数をランダムに返す。このような、関数が返す値を**返り値** (**戻り値**) と呼ぶ。ユーザーが定義した関数の返り値は、最後に実行した命令の値である。

関数の返り値の例

```
1 def rev(x) # xを逆から読んだ整数を返す
2   y = x.to_s
3   y.reverse.to_i
4 end
5
6 puts rev(13), rev(25), rev(13)+rev(25)
```

上の関数 rev は、引数の整数を逆から読んだ数を返り値として返す。

実行例

```
31
52
83
```

[22] 関数呼び出しのとき、引数を囲む丸かっこは、誤解が生じないときは省略できる。

(9.4) **例** 引数を 2 つ受け取り、比較して大きい方 (正確には小さくない方) を返す関数 big を定義して使用してみる。

関数定義と利用の例

```
1 def big(a, b) # aとbの最大値を返す
2   if a > b
3     a
4   else
5     b
6   end
7 end
8
9 puts big(2, 3), big("ab", "ac")
```

実行例

```
3
ac
```

(9.5) **例** 数値の引数を 1 つ受け取り、1 からその数までの総和を返す関数 sum を定義して使用してみる。

関数定義と利用の例

```
1 def sum(n) # 1からnまでの総和を返す
2   total = 0
3   for i in (1..n)
4     total = total + i
5   end
6   total
7 end
8
9 puts sum(10), sum(100)
```

実行例

```
55
5050
```

(9.6) **関数定義中の変数のスコープ** 上の総和を求める関数 sum を次のように書いても、エラーが出て動作しない。

関数定義の失敗例

```
1  def sum() # 1からnまでの総和を返す失敗例
2    total = 0
3    for i in (1..n)
4      total = total + i
5    end
6  end
7
8  n = 10
9  sum()
10 puts total
```

これは関数定義中、そこで現れた引数も含めた変数 (1 行目から 6 行目までの total や n) は、その関数定義中でのみ有効であり、呼び出し元に同名の変数があったとしても別のものとして扱われるからである。このような、変数が有効な範囲を**スコープ**と言う[23]。上の例だと、3 行目の n は、8 行目の n と別物なので、値がないというエラーが 3 行目で発生する。もし、そのエラーがなかったとしても、10 行目の total は、2, 4 行目の total と別物なので、10 行目でもエラーが発生する。

逆に、スコープのおかげで、関数定義の内外での変数名の衝突が避けられる。

(9.7) **課題 11 − 3 要素の最小値** 引数を 3 つとり、その最小値を返す関数 small(a, b, c) を定義し、それを用いて、数を 3 つキーボードから入力させ、その最小値を表示するプログラムを作成し提出せよ。ただし、配列の最小値を求めるメソッド min は用いても用いなくてもよい。

1, Enter, 2, Enter, -1, Enter と入力したときの実行例

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

[23] Small Basic にはスコープがなかったので、このようなコードでも期待通りに動作した

最初の 3 行分は人間の入力であり、最後の 1 行が ruby の出力である。

(9.8) **課題 12 – 階乗** 階乗を計算して返す関数 fact(n) を定義し、それを用いて、数を 1 つキーボードから入力させ、その階乗を表示するプログラムを作成し提出せよ。ただし、(11.1) のような再帰は用いないこと。

36, Enter と入力したときの実行例

```
36
371993326789901217467999448150835200000000
```

最初の行は人間の入力であり、次の行が ruby の出力である。

§10 関数定義の例

(10.1) **return 文** 関数 (メソッド) の実行を中断し呼び出し元へ戻る。

return 文の文法

```
return
return <式>
```

その関数の返り値は〈式〉で指定するが、〈式〉を省略すると返り値は nil になる。

(10.2) **素数判定** 引数で与えられた整数が素数かどうか判定する関数 prime?を作る。そして、これを用いて、入力された整数が素数かどうか表示するプログラムを作ってみる。

素数判定の例

```
1  def prime?(n) # nが素数ならtrue、他はfalseを返す
2    if n == 1
3      return false
4    end
5    for a in (2..n-1)
6      if n % a == 0
7        return false
8      end
9    end
10   true
11 end
12
13 puts prime?(gets.to_i)
```

ここでは、1 は素数ではないから直ちに false を返し、2 以上 n 未満の整数で割り切れたら直ちに false を返し、一度も割り切れなかったら true を返すというアルゴリズムを採用した[24]。

17 を入力したときの実行例

```
17
true
```

ただし、実行例の 1 行目は人間がキーボードから入力したものである。

(10.3) **課題 13 – 素数列挙** 1 から 100 までの素数からなる配列を作成し、出力するプログラムを作成し提出せよ。ただし、(10.2) の関数 prime?を変更なしに再利用せよ。

実行例（長いので途中で折り返して例示している）

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41,
43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]
```

(10.4) **約数のリスト** 引数で与えられた整数に対して、その整数の約数からなる配列を返り値として返す関数 divisors を定義する。そして、これを用いて、入力された整数の約数の配列を表示するプログラムを作ってみる。

約数列挙の例

```
1  def divisors(n) # nの約数の配列を返す
2    divs = []
3    for a in (1..n)
```

[24] n-1 まで割り切れるか試す必要はなく、$\sqrt{n}$ までで十分である。

```
4       if n % a == 0
5         divs.push a
6       end
7     end
8     divs
9   end
10
11  p divisors(gets.to_i)
```

関数 divisors では、1 以上 n 以下の整数が、n を割り切ったならば、配列 divs に追加する。そして、最後に、配列 divs を返り値として返している。

72 を入力したときの実行例

```
72
[1, 2, 3, 4, 6, 8, 9, 12, 18, 24, 36, 72]
```

ただし、実行例の 1 行目は人間がキーボードから入力したものである。

(10.5) **課題 14 – 素数である約数** 引数で与えられた整数に対して、その整数の約数のうち素数であるものからなる配列を返り値として返す関数 primedivisors を定義せよ。そして、それを用いて、キーボードから入力された整数の約数のうち、素数であるものを表示するプログラムを作成し提出せよ。

72 を入力したときの実行例

```
72
[2, 3]
```

ただし、実行例の 1 行目は人間がキーボードから入力したものである。

解くにあたり、(10.4) の例を参考にし、かつ、素数判定関数として、(10.2) の prime?を変更なしに再利用すること。

(10.6) **課題 15$^{\#}$ – 公約数** 2 つの整数を引数にとり、それらの公約数からなる配列を返す関数 commondivisors(m, n) を定義せよ。そして、それを用いて、2 数をキーボードから入力し、それらの公約数の配列を表示するプログラムを作成し提出せよ。解くにあたり、(10.4) の関数 divisors を再利用すること。また、Array クラスの演算子& (集合としての共通部分) を用いよ。

72 と 60 を入力したときの実行例

```
72
60
[1, 2, 3, 4, 6, 12]
```

ただし、実行例の最初の 2 行は人間がキーボードから入力したものである。

(10.7) **課題 16$^{\#}$ – 最大公約数** 2 つの整数を引数にとり、それらの最大公約数を返す関数 gcd(m, n) を定義せよ。そして、それを用いて、2 数をキーボードから入力し、それらの最大公約数を表示するプログラムを作成し提出せよ。

72 と 60 を入力したときの実行例

```
72
60
12
```

ただし、実行例の最初の 2 行は人間がキーボードから入力したものである。

解くにあたり、(10.4) の関数 divisors を変更なしに再利用すること。また、(10.6) を参考にしてもよい。

§11 再帰

(11.1) **再帰呼び出し** 関数が自分自身を呼び出すことを**再帰呼び出し**と呼んだり、その関数が**再帰的に**定義されていると言う。数学でも再帰的な定義はよく見られる[25]。

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

[25] n が負の場合は $n!$ を定義していないことに注意。

この階乗の定義をそのまま ruby の関数にすると以下のようになる[26]。

階乗関数の再帰による定義

```
1 def fact(n) # nの階乗を返す
2   if n == 0
3     1
4   else
5     n * fact(n-1)
6   end
7 end
```

実際に fact(4) を呼び出したときの動作は以下のようになるが、ここでは省略。

(11.2) **配列の総和** 数値を要素に持つ配列の総和を求める関数 sum を、繰り返しを用いて定義すると次のようになる。

配列の総和 sum の繰り返しによる定義 その 1

```
1 def sum(ary) # 配列 aryの要素の総和を返す
2   total = 0
3   for i in (0...ary.size)
4     total = total + ary[i]
5   end
6   total
7 end
8
9 puts sum [2, 3, 5, 7]
```

あるいは、配列の上を繰り返すこともできるので、次のようにも定義できる[27]。

配列の総和 sum の繰り返しによる定義 その 2

```
1 def sum(ary) # 配列 aryの要素の総和を返す
2   total = 0
3   for x in ary
4     total = total + x
5   end
6   total
7 end
8
9 puts sum [2, 3, 5, 7]
```

ところで、総和は、

$$\texttt{ary}\text{の総和} = \begin{cases} 0 & (\texttt{ary}\text{が空}) \\ \texttt{ary[0]} + \text{「}\texttt{ary[1..-1]}\text{の総和」} & (\texttt{ary}\text{が空ではない}) \end{cases}$$

と再帰的にも定義できるので、これを ruby の関数にすると以下のようになる[28]。

配列の総和 sum の再帰による定義の例

```
1 def sum(ary) # 配列 aryの要素の総和を返す
2   if ary.size == 0
3     0
4   else
5     ary[0] + sum(ary[1..-1])
6   end
7 end
8
9 puts sum [2, 3, 5, 7]
```

実行例

```
17
```

(11.3) **課題 17 – 配列の要素の積** 数値を要素に持つ配列 ary を引数にとり、その要素の積を求める関数 mul(ary) を、再帰を用いて定義せよ。ただし、空配列に対しては 1 を返すこと。そして、関数 mul(ary) を用いて、puts

[26] n が非負整数以外で呼び出されることは想定していないが、仮に n に負の数や、整数以外を与えて呼び出すとどうなるだろうか。

[27] こちらの方が、配列の添字 i や配列のサイズという、総和を求めるのに不要なものをプログラムから排除できるので、結果として読み易いプログラムになる。

[28] if ary.size == 0 は、if ary.empty? と書き換えることができる。この empty?は、配列が空なら真、そうでなければ偽を返すメソッドである。配列が空かどうか知りたいだけなので、empty?メソッドを用いると、配列のサイズの計算がプログラムから消えて読み易いプログラムになる。

mul([2, 3, 5, 7]) (数値は任意) などとして適当な配列の要素の積を表示するプログラムを作成せよ。

puts mul([2, 3, 5, 7]) としたときの実行例

```
210
```

(11.4) **二項係数** 二項係数は、整数 $0 \leq r \leq n$ に対して、

$$\binom{n}{r} = \frac{n(n-1)\cdots(n-r+1)}{r!} = \frac{n!}{r!(n-r)!}$$

で定義された。これをそのまま ruby の関数にすると以下のようになる。ただし、(11.1) で定義した関数 fact をそのまま用いる。

二項係数の定義の例

```
1  def fact(n) # nの階乗を返す
2    if n == 0
3      1
4    else
5      n * fact(n-1)
6    end
7  end
8
9  def binom(n, r) # 二項係数nCrを返す
10   fact(n) / fact(r) / fact(n-r)
11 end
12
13 puts binom(10, 3)
```

他方、二項係数は、整数 $0 \leq r \leq n$ に対して、

$$\binom{n}{r} = \begin{cases} 1 & (r = 0 \text{ または } r = n) \\ \binom{n-1}{r-1} + \binom{n-1}{r} & (0 < r < n) \end{cases}$$

と再帰的にも定義できた[29]。これを ruby の関数にすると以下のようになる。

二項係数の再帰による定義の例

```
1  def binom(n, r) # 二項係数nCrを返す
2    if r == 0 || r == n
3      1
4    else
5      binom(n-1, r-1) + binom(n-1, r)
6    end
7  end
8
9  puts binom(10, 3)
```

実行例

```
120
```

(11.5) **課題 18 – フィボナッチ数列** フィボナッチ数列とは、

$$F_1 = 1,$$
$$F_2 = 1,$$
$$F_{n+2} = F_{n+1} + F_n$$

で再帰的に[30]定義される数列である。1 つの整数 n を引数にとり、フィボナッチ数列の第 n 項を返す関数 fib(n) を定義し、それを用いて、数を 1 つキーボードから入力させ (n とする)、フィボナッチ数列の第 n 項を表示するプログラムを作成し提出せよ。

30 を入力したときの実行例

```
30
832040
```

ただし、実行例の 1 行目は人間がキーボードから入力したものである。計算には意外に時間がかかるので注意。

[29] パスカルの三角形を作る関係式である

[30] 数学では帰納的にと言う。同じことである。

(11.6) **課題 19# – ユークリッドの互除法** 2 つの整数を引数にとり、それらの最大公約数を返す関数 gcd(m, n) を (10.7) で定義したが、ここでは、ユークリッドの互除法を用いて再帰的な関数を定義する。a と b の最大公約数を求めるユークリッドの互除法は以下のような再帰的アルゴリズムである。

(1) $b = 0$ ならば、a を最大公約数として返して終了する。
(2) そうでないなら、a を b で割った余りを r として、b と r ($b > r$ に注意) の最大公約数を求める。

2 つの整数を引数にとり、それらの最大公約数を返す関数 gcd(m, n) を、ユークリッドの互除法を用いて定義し、それを用いて、数を 2 つキーボードから入力させ、それらの最大公約数を表示するプログラムを作成し提出せよ。

72 と 60 を入力したときの実行例

```
72
60
12
```

ただし、実行例の最初の 2 行は人間がキーボードから入力したものである。

(11.7) **課題 20# – ハノイの塔** ハノイの塔というゲームがある。互いに大きさの異なる円板が n 枚あり、最初は場所 A に下ほど大きくなるように積み上げられている。他に場所 B, C がありそこには円板は置かれていない。ゲームの目的は n 枚の円板を場所 C に移動することである。

円板の移動では、最も上に積んである 1 枚だけが移動でき、移動した先でも下ほど大きくなるように積み上げられていなくてはならない。

例えば、

円板が 2 枚のとき: A→B, A→C, B→C

円板が 3 枚のとき: A→C, A→B, C→B, A→C, B→A, B→C, A→C

という手順となる。つまり、

(1) A から B へ、上から $n - 1$ 枚を移動する
(2) A から C へ円板を 1 枚移動する (大きさ最大のものである)
(3) B から C へ、$n - 1$ 枚を移動する

$n - 1$ 枚を移動するには再帰の考えを用いる。

ハノイの塔の解を表示する関数 hanoi(start, goal, other, n) を再帰を用いて作成せよ。ここで、n は移動する円板の枚数、start, goal, other は、それぞれ円板の移動元、移動先、残りの場所の名前である。それを用いて、キーボードから正整数 n 入力させて、円板 n 枚のハノイの塔の解を表示するプログラムを作成し提出せよ。

4 ををを入力したときの実行例

```
4
A->B
A->C
B->C
A->B
C->A
C->B
A->B
A->C
B->C
B->A
C->A
B->C
A->B
A->C
B->C
```

ただし、実行例の最初の行は人間がキーボードから入力したものである。

§12 コンテナの要素の反復操作 (Enumerable)

(12.1) **Enumerable モジュール** Ruby には配列 (Array) やハッシュ (Hash) のようなコンテナクラス[31]がいくつかある。これらは、他のオブジェクトを

[31] コンテナとは、配列 (Array) オブジェクトのように、他のいくつかのオブジェクトを「格納」しているオブジェクトのことを指す。

「格納」しているという共通点があるため、行いたい操作も共通したものが多いといえる。例えば、配列やハッシュの各要素を表示したいとか、配列やハッシュの要素を小さい順に整列したいだとかである。Array クラスや Hash クラスにそういった機能を追加するのが **Enumerable モジュール**であり、Array クラスには Enumerable モジュールが **Mix-in** されている、などと言う。多くのコンテナに共通に Enumerable モジュールが Mix-in されているため、それらのコンテナに対する整列などの操作は共通の記述で行える。

(12.2) **each メソッド** コンテナの各要素を繰り返し訪れるのが **each** メソッドである。配列などのコンテナオブジェクト container に対する each メソッドの文法は、

each メソッドの文法

```
container.each {|x|
  <x を用いた処理>
}
```

となる。下の例では、配列の各要素を表示する。

each メソッドの例

```
1 [2, 4, 5, 7].each {|x|
2   puts x
3 }
```

これは、次の for 文を用いた例とほぼ等価である[32]。

for 文を用いた等価な例

```
1 for x in [2, 4, 5, 7]
2   puts x
3 end
```

実行例

```
2
4
5
7
```

(12.3) **ブロック** each メソッドの文法にあるブレース ({}) で囲まれた部分を**ブロック**と呼ぶ。

ブロックの文法

```
{|x1, x2, .., xn|  <処理> }
```

ブロックは、x1 から xn を引数とする関数の役割をし、最後に実行した式がブロックの返り値となる。しかし、通常の関数定義のように関数名を付けるわけではないので、**無名関数**と呼ばれることもある。

Ruby では for 文のような形式よりも、each メソッドのようなブロックを用いた形式で、さまざまな操作を記述することが多い。

(12.4) **Enumerable#map** Enumerable モジュールには、**map** メソッド[33] が定義されているので、Enumerable モジュールを Mix-in しているどのクラスでも、map メソッドを利用できる。map メソッドは、コンテナの各要素を写像して、新しい配列を返すメソッドである。

Enumerable#map の文法

```
container.map {|x|
 <x を用いた式>
}
```

次の例では、配列の各要素を平方して新しい配列を作成している。

[32] 唯一の違いは、each メソッドでは変数のスコープが導入されるが、for 文だと導入されないことである。つまり、each メソッドの外側にあるかも知れない変数 x は変更されないが、for 文だと変更される。

[33] あるクラスやモジュールで、あるメソッドが定義されているとき、そのメソッドを「クラス名#メソッド名」や「モジュール名#メソッド名」のように表す。同名メソッドでもクラスにより動作が異なる場合があるので、クラスやモジュール名を明示したい時や、どのクラスやモジュールで利用できるのかを明示したい時に便利である。

Enumerable#map **の例**

```
1 a = [2, 4, 5, 7]
2 b = a.map {|x|
3   x**2
4 }
5 p b
```

実行例

```
[4, 16, 25, 49]
```

また、Enuberable#map は破壊的ではない。つまり、上の例では、a は map メソッド実行後も、[2, 4, 5, 7] のままである。

(12.5) **Enumerable#find** Enumerable#find は、ブロックの返り値が真になった最初の要素を返す。

Enumerable#find **の例**

```
1 a = [2, 4, 5, 7]
2 t = a.find {|x|
3   x % 3 == 1
4 }
5 puts t
```

実行例

```
4
```

(12.6) **Enumerable#select** Enumerable#select は、ブロックの返り値が真になる要素を集めた配列を返す。

Enumerable#select **の例**

```
1 a = [2, 4, 5, 7]
2 b = a.select {|x|
3   x % 3 == 1
4 }
5 p b
```

実行例

```
[4, 7]
```

また、Enuberable#select は破壊的ではない。つまり、上の例では、a は select メソッド実行後も、[2, 4, 5, 7] のままである。

(12.7) **Enumerable#sort_by** Enumerable#sort_by メソッドは、コンテナの要素どうしを直接比較するのではなく、コンテナの各要素を一旦写像したもの (**ソートキー**と呼ぶ) を比較してソートする。次の例では、数値をその絶対値の小さい順にソートしている。

Enumerable#sort_by **の例**

```
1 a = [4, -3, 2, -5, 1]
2 b = a.sort_by {|x| x.abs }
3 p b
```

実行例

```
[1, 2, -3, 4, -5]
```

絶対値はあくまでソートキーであって、整列されるのは元の要素である。sort メソッドと同様に、sort_by メソッドも破壊的ではない。

(12.8) **例 (偶数からなる配列)** 正整数 n が与えられたときに、[2, 4, 6, .., 2n] という配列をいくつかの方法で作成してみる。

for **文を用いた例**

```
1 n = 4
2 a = []
3 for i in (1..n)
4   a.push(i*2)
5 end
6 p a
```

each メソッドを用いた例

```
n = 4
a = []
(1..n).each {|x|
  a.push(x*2)
}
p a
```

map メソッドを用いた例

```
n = 4
a = (1..n).map {|x|
  x*2
}
p a
```

select メソッドを用いた例

```
n = 4
a = (1..n*2).select {|x|
  x % 2 == 0
}
p a
```

実行例

```
[2, 4, 6, 8]
```

(12.9) **例 (異なる方法での整列)** 3 人に対して、算数と国語の試験をした。

名前	算数	国語
A	78	74
B	69	85
C	77	78

1 人の名前と算数と国語の試験の点数を、配列で `["A", 78, 74]` のように表すことにし、それらを 3 人分集めて配列を作ると、`[ ["A", 78, 74], ["B", 69, 85], ["C", 77, 78] ]` となる。この配列を名前、算数の得点、国語の得点、合計得点でそれぞれ整列すると次のようになる。

異なる方法での整列の例

```
a = [["A",78,74], ["B",69,85], ["C",77,78]]
a0 = a.sort_by {|name, math, jpn| name }
a1 = a.sort_by {|name, math, jpn| math }
a2 = a.sort_by {|name, math, jpn| jpn }
a3 = a.sort_by {|name, math, jpn| math + jpn }
p a0, a1, a2, a3
```

実行例

```
[["A", 78, 74], ["B", 69, 85], ["C", 77, 78]]
[["B", 69, 85], ["C", 77, 78], ["A", 78, 74]]
[["A", 78, 74], ["C", 77, 78], ["B", 69, 85]]
[["A", 78, 74], ["B", 69, 85], ["C", 77, 78]]
```

(12.10) **課題 21 – Enumerable#map** 数を 4 つキーボードから入力させて、その 4 数を要素に持つ配列をまず作り、次に、4 数をそれぞれ平方した要素を持つ配列を作り、表示するプログラムを作成し提出せよ。

1, 3, 5, 7 の順に入力したときの実行例

```
1
3
5
7
[1, 9, 25, 49]
```

ただし、実行例の 1 行目から 4 行目は人間がキーボードから入力したものである。

(12.11) **課題 22 – Enumerable#select** 正整数を 1 つキーボードから入力させて、それを n とするとき、1 以上 n 以下の整数のうち、3 の倍数でも 4

の倍数でもない整数からなる配列をまず作り、表示するプログラムを作成し提出せよ。ただし、for 文は用いないこと。

14 を入力したときの実行例

```
14
[1, 2, 5, 7, 10, 11, 13, 14]
```

ただし、実行例の最初の行は人間がキーボードから入力したものである。

(12.12) **課題 23 – 文字数によるソート** 文字列を 4 つキーボードから入力させて、それらを文字数の短い順に整列して表示するプログラムを作成し提出せよ。

実行例

```
Hokkaido
University
Education
Kushiro
Kushiro
Hokkaido
Education
University
```

ただし、実行例のはじめの 4 行は人間がキーボードから入力したものである。

(12.13) **課題 24 – 素数の配列** 1 から 100 までの素数からなる配列を (10.2) の prime? と select メソッドを利用して作成し、その配列を表示するプログラムを作成し提出せよ。

実行例

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41,
43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]
```

上の出力では、行が長いので途中で折り返して例示している。

§13 正規表現 (Regexp)

(13.1) **正規表現** **正規表現**とは、文字列の比較や照合に用いる「パターン」を表す非常に強力な方法である。Ruby に限らず多くの言語[34]で利用できる。正規表現といっても言語ごとに方言があり、すべをここで網羅するわけにはいかないが、どの言語でもほぼ共通に使える正規表現記号を以下に記す。また、正規表現記号ではない文字、例えば「a」とか「"」などは、単にその文字そのものにマッチする正規表現になる。

主な正規表現記号 (メタ文字)

記号	意味
.	改行 (\n) を除く任意の 1 文字にマッチする
^	行頭 (文字列先頭や改行直後) にマッチする。幅は 0
$	行末 (文字列末尾や改行直前) にマッチする。幅は 0
[]	角かっこ内のどれか 1 文字にマッチする
[^]	角かっこ内に含まれない 1 文字にマッチする
+	直前の表現の 1 回以上の繰り返し (最長一致)
*	直前の表現の 0 回以上の繰り返し (最長一致)
?	直前の表現の 0 回または 1 回の繰り返し (最長一致)
|	|の前か後ろどちらかの表現にマッチする
()	後方参照などのためのグループ化
\1, \2,..	グループ化した文字列の後方参照

(13.2) **Regexp クラス** Ruby では正規表現は Regexp クラスのオブジェクトである[35]。Regexp オブジェクトは正規表現をスラッシュで囲んで作成できる。

文字列とのマッチには、演算子=~を用いる。この演算子の否定は!~である。演算の結果、照合する文字列の一部とマッチしたなら、文字列の何文字目か

[34] 例をあげると、Perl, Python, PHP, Java, JavaScript, いわゆる.NET, awk, sed などがあり、文字列検索に正規表現が使えるエディタには、Microsoft Word, Emacs (Meadow), vi, サクラエディタ、秀丸などがある

[35] 正規表現を扱える言語は多いが、正規表現がオブジェクトである言語はほとんどない。

らマッチしたかが返り、マッチしなかったならば nil が返る。複数の位置でマッチする場合もありうるが、位置は最も左のものが返る。また、マッチした文字列は、ruby の組み込み変数$&に自動的に設定される。

Regexp オブジェクトの作成の例

```
1 re = /bc/
2 p re =~ "abcd abcd"
3 p re =~ "b cb c"
```

実行例

```
1
nil
```

(13.3) **メタ文字「.」 (任意の 1 文字)** メタ文字「.」は改行以外の任意の 1 文字にマッチする。

メタ文字「.」の例

```
1 re = /b.d/
2 p re =~ "abcd abcd"
3 p $&
4 p re =~ "b cb d"
5 p $&
```

実行例

```
1
"bcd"
3
"b d"
```

文字「.」 (ピリオド) 自身にマッチさせたいときは、バックスラッシュを前置して、\. と記述する。他のメタ文字でも同様にバックスラッシュを前置すれば、その文字自身にマッチする。

単なる文字「.」の例

```
1 re = /1\.3/
2 p re =~ "123 1.3"
3 p $&
```

実行例

```
4
"1.3"
```

(13.4) **メタ文字「^」と「$」 (行頭、行末)** メタ文字「^」と「$」は、それぞれ行頭と行末にマッチする。文字列が改行 (\n) を含めば、その直後は行頭、直前は行末と見なされる。

メタ文字「^」と「$」の例

```
1  re1 = /^123/
2  p re1 =~ "0123"
3  p re1 =~ "1234"
4
5  re2 = /123$/
6  p re2 =~ "0123"
7  p re2 =~ "1234"
8
9  re3 = /^123$/
10 p re3 =~ "0123"
11 p re3 =~ "1234"
12 p re3 =~ "123"
```

実行例

```
nil  # re1のマッチ
0
1    # re2のマッチ
nil
nil  # re3のマッチ
nil
0
```

(13.5) **メタ文字「[]」(文字クラス指定)** メタ文字「[]」は、角かっこ内の 1 文字にマッチするが、[a-z] や [0-9] のように文字の範囲を指定することもできる。また、角かっこ内の先頭に「^」を書くと、角かっこ内以外の 1 文字にマッチする[36]。

メタ文字「[]」の例

```
1 re1 = /201[0-3]/
2 p re1 =~ "2013-2-24"
3 p re1 =~ "2014-2-24"
4 re2 = /201[^0123]/
5 p re2 =~ "2013-2-24"
6 p re2 =~ "1999:201b"
```

実行例

```
0   # re1のマッチ
nil
nil # re2のマッチ
5
```

(13.6) **例 (3 桁の整数にマッチする正規表現)** 3 桁の整数にマッチする正規表現を作り、入力された文字列が整数を含んでいれば、それを表示するプログラムを作成する。

3 桁の整数にマッチする正規表現の例

```
1 re = /[1-9][0-9][0-9]/
2 if re =~ gets
3   puts $&
4 end
```

「合計 103 万円」と入力した場合の入力と実行例

```
合 計103 万 円
103
```

[36] 文字クラス指定の [] 内の文字「^」は、[] 内の先頭にあるかどうかに関わらず、行頭にマッチするメタ文字「^」の働きはない。

上の実行例では、1 行目はキーボードから入力した文字列であり、2 行目がプログラムの出力である[37]。

(13.7) **課題 25 – 学籍番号にマッチする正規表現** 2013 年度入学生の学籍番号にマッチする正規表現を作り、キーボードから入力された文字列が 2013 年度入学生の学籍番号を含むならば OK、そうでないならば NG と出力するプログラムを作成せよ。ただし、2013 年度入学生の学籍番号は、

b4⟨3 から始まる 4 桁の整数⟩⟨ アルファベット小文字が 2 文字⟩

の形をしているとし、日本人の名前としてはあり得ないアルファベットなども許すこととする。

「I am b43291bx」と入力した場合の入力と実行例

```
I am b43291bx
OK
```

「b49299aw has a pen.」と入力した場合の入力と実行例

```
b49299aw has a pen.
NG
```

(13.8) **メタ文字「+」、「*」、「?」(直前の表現の繰り返し)** メタ文字「+」、「*」、「?」は、直前の表現の、それぞれ、1 回以上、0 回以上、0 または 1 回の繰り返しにマッチする。

メタ文字「+」、「*」、「?」の例

```
1 re1 = /ab+/
2 puts re1 =~ "abbac"
3 puts re1 =~ "accab"
4 re2 = /ab*/
5 puts re2 =~ "abbac"
```

[37] この方法だと、桁数が多くなると正規表現が長くなってしまうが、メタ文字「{}」(直前の表現の繰り返し回数の指定) を用いると、例えば 8 桁の整数にマッチする正規表現は/[1-9][0-9]{7}/と書ける。

```
6 puts re2 =~ "accab"
7 re3 = /ab?/
8 puts re3 =~ "abbac"
9 puts re3 =~ "accab"
```

実行例

```
0   # re1のマッチ (abb にマッチ)
3   #            (ab にマッチ)
0   # re2のマッチ (abb にマッチ)
0   #            (a にマッチ)
0   # re3のマッチ (ab にマッチ)
0   #            (a にマッチ)
```

1 回以上という説明では、1 回なのか 2 回なのかあいまいであるが、できる限り回数は多くするようにマッチする (最長一致)。例えば、上の出力の 1 番目は 2 文字 (ab) ではなく、3 文字 (abb) にマッチする。しかし、マッチ位置は繰り返し回数とは無関係に、最も左の位置になる。例えば、上の出力の 4 番目では、位置 0 で 1 文字 (a) にマッチしているが、より長いマッチとなる位置 3 で 2 文字 (ab) にマッチするわけではない。

これらの繰り返しを表すメタ文字は、他のメタ文字と組み合わせて使うことが多い。

メタ文字「+」、「*」、「?」の他のメタ文字との組み合せの例

```
1 re1 = /[a-z]+/
2 puts re1 =~ "aw9981tz"
3 re2 = /[a-z]+[0-9]*:/
4 puts re2 =~ "aw9981:"
5 puts re2 =~ "9981aw:"
```

実行例

```
0   # re1のマッチ (aw にマッチ)
0   # re2のマッチ (aw9981: にマッチ)
4   #            (aw: にマッチ)
```

(13.9) **メタ文字「|」 (選択)*** メタ文字「|」は、「表現 1|表現 2」と用いると、表現 1 か表現 2 のいずれかにマッチする。しかし優先順位が低いため、グループ化 (()) とともに用いることが多い。

メタ文字「|」の例

```
1 re1 = /ab|xy/
2 puts re1 =~ "abxy"
3 re2 = /a(b|xy)/
4 puts re2 =~ "abxy"
5 re3 = /(ab|x)y/
6 puts re3 =~ "abxy"
```

実行例

```
0   # ab にマッチ
0   # ab にマッチ
2   # xy にマッチ
```

(13.10) **後方参照*** メタ文字「()」でグループ化された表現にマッチした文字列は、1 番目のグループが「\1」、2 番目のグループが「\2」... で参照できる。表現ではなく、マッチした文字列を参照することに注意が必要である。

後方参照の例

```
1 re1 = /(a)\1/
2 puts re1 =~ "aaa"
3 re2 = /([a-z])\1/
4 puts re2 =~ "abba"
```

実行例

```
0   # aa にマッチ
1   # bb にマッチ
```

(13.11) **課題 26# – 小数にマッチする正規表現** 整数部分は 0 のみ、または先頭が 0 ではない整数であり、次に小数点がきて、その次に 1 桁以上の数字が続くような小数にマッチする正規表現を作れ。つまり、例えば下のような小数にマッチするような正規表現である。

```
12.3        10.01        0.12        0.00
```

それを用いて、キーボードから入力された文字列が該当する小数を含むならば、その小数を表示するプログラムを作成し提出せよ。

「9m012.340w」と入力した場合の実行例

```
9m012.340w
12.340
```

「9m012.w34」と入力した場合の実行例

```
9m012.w34
（この行は空行）
```

ただし、ともに、1 行目は人間がキーボードから入力したものである。

(13.12) **課題 27# – 数にマッチする正規表現** 整数または小数にマッチするような正規表現を作れ。ただし、整数とは 0 かまたは先頭が 0 ではない 1 桁以上の整数であり、小数とは、(13.11) と同じ意味である。つまり、例えば下のような数にマッチするような正規表現である。

```
0    1    123    12.3    10.01    0.12    0.00
```

それを用いて、キーボードから入力された文字列が該当する数を含むならばそれを表示するプログラムを作成し提出せよ。

「x9m012.w34」と入力した場合の実行例

```
x9m012.w34
9
```

「m012.34」と入力した場合の実行例

```
m012.34
0
```

「m12.y34w」と入力した場合の実行例

```
m12.y34w
12
```

「m12.340w」と入力した場合の実行例

```
m12.340w
12.340
```

ただし、いずれも、1 行目は人間がキーボードから入力したものである。

§14 文字列操作

(14.1) **文字列末尾の削除** gets（ゲットエス）で文字列を入力すると、末尾には改行文字が付加されている。これを削除するには、末尾の改行文字を取り除いた新しい文字列を返す、String#chomp メソッドを使えばよい[38]。

String#chomp の例

```
1 s = gets
2 t = s.chomp
3 p s, t
```

bcg と入力したときの実行例

```
"bcg\n"
"bcg"
```

[38] 末尾に改行のない文字列に対して chomp を実行すると、その文字列の単なるコピーが返る。

(14.2) **進法変換** 文字列 (String のオブジェクト) を数値 (Fixnum などのオブジェクト) に変換するには、to_i メソッドを用いた。実は、to_i は省略可能な引数を 1 つとり、変換するときの進法を指定できる。この引数は、省略すれば 10 となる。

String#to_i(base) の例

```
1 s = "12"
2 puts s.to_i
3 puts s.to_i(7)
```

実行例

```
12
9
```

変換するときの進法を指定するのであって、得られた数値を puts(プットエス) で表示するときは 10 進法となる。

反対に、数値 (Fixnum などのオブジェクト) を、文字列 (String のオブジェクト) に変換するには、to_s メソッドを用いた。実は、to_s も省略可能な引数を 1 つとり、変換するときの進法を指定できる。

Fixnum#to_s(base) の例

```
1 a = 12
2 puts a.to_s
3 puts a.to_s(7)
```

実行例

```
12
15
```

(14.3) **フォーマット** Float クラスの数値 (浮動小数点数) の小数点以下が必要以上に長く表示されてしまう、だとか、数値が 2 桁か 3 桁かわからないけれど、3 桁右揃えで表示したいといったときは、String クラスのオブジェクトが利用できる演算子%を用いる。

String#%の文法

```
<文字列> % <引数>                  # 引数が1つの場合
<文字列> % [<引数1>, <引数2>, ..]  # 複数の場合
```

この演算をすると、小数点以下の桁数などの書式を指定した上で、引数を文字列の中に埋め込み、その結果を演算の結果とする。下の例では、str の中の%s という**フォーマット文字列**に、数値 a を文字列化したものが埋め込まれている[39]。

%s の例

```
1 a = 1 / 3.0
2 str = "埋め%s込めます" % a
3 puts str
```

実行例

```
埋め0.3333333333333333込めます
```

(14.4) **フォーマット文字列** %s の他にもさまざまなフォーマット文字列がある。

フォーマット文字列の文法

```
%[引数指定$][フラグ][幅][.精度]指示子
```

引数指定、フラグ、幅、精度、指示子のうち、指示子 (と先頭の%) が省略不可能で、その他は省略可能である (つまり、%s というフォーマット文字列は、必要最小限のもののみで構成されている)。これらを順に、主なもののみ解説するが、引数指定は使用頻度が少ないので解説を省略する。

[39] この例では 1 つの引数が数値だから問題はないが、もし 1 つの配列を埋め込みたいときは、単に配列を引数にしてしまうとうまくいかない。複数の引数を渡しているものと解釈されてしまうからである。この場合、その配列を [] でくるんで引数とするのが最も容易な解決策である。

主な「フラグ」

記号	意味
+	埋め込むものが正の数値の場合先頭に + を付加する
-	幅指定がある場合、左詰めにする
0	幅指定があり右詰めの場合、余った部分を 0 で埋める

以下の幅と精度において、記号の欄に ⟨ 数値 ⟩ と記してあるものは、先頭が 0 であってはならない。

「幅」

記号	意味
<数値>	引数を埋め込むときの幅の指定

次の精度の指定は省略すると 6 とみなされる。

主な「精度」

記号	意味
.<数値>	浮動小数点数の小数点以下の桁数

主な「指示子」

記号	意味
s	文字列。String でないものは to_s される
d	整数
f	浮動小数点数

(14.5) **フォーマットの例** いくつかの実例をあげる。

幅と詰めの指定の例

```
1 str = "abc"
2 puts "(%s)" % str              出力は (abc)
3 puts "|%6s|" % str             出力は |   abc|
4 puts "<%2s>" % str             出力は <abc>
5 puts "!%-6s!" % str            出力は !abc   !
```

上のように幅が不足するとき、幅指定は無視される。また、上の例ではすべて括弧等でくるんでいるが、右寄せや左寄せがわかるようにしているだけで、常に必要なわけではない。

整数の例

```
1 puts "あ%2sい%dう%+-3dえ" % [111, -2, 3]
```

実行例

```
あ111い-2う+3  え
```

上のように、フラグは複数指定可能である。

浮動小数点数の例

```
1 a = 100 / 3.0
2 puts "[%f]{%.2f}<%4f>(%6.1f)" % [a, a, a, a]
```

実行例

```
[33.333333]{33.33}<33.333333>(  33.3)
```

上の 3 つ目では、幅 4 を指定しても、精度の指定がないため 6 とみなされ、結果として幅が不足するので幅指定が無視されている。4 つ目では幅が不足しないので、幅指定が有効になっている。

(14.6) **課題 28 – 進法変換** キーボードから数字の列を入力すると、それを 7 進法で表された数値とみなして 10 進法に変換し、下の例のように表示するプログラムを作成し提出せよ。ただし、7 進法で許されていない文字が入力されたとき、つまり、0 から 6 以外の文字が存在したときは、「エラー」とだけ表示すること。

164 と入力した時の実行例

```
164
164 （7 進 法） = 95 （10 進 法）
```

174 と入力した時の実行例

```
174
エ ラ ー
```

ただし、ともに、1 行目は人間がキーボードから入力したものである。

(14.7) **置換** String#sub メソッド、String#gsub メソッドを使うと、正規表現にマッチする部分を置換できる。sub は、文字列中の最初にマッチした部分のみ置換し、gsub は、マッチした部分すべてを置換する[40]。

String#sub と String#gsub の文法

```
<文 字 列>.sub(<正 規 表 現>, <置 換 文 字 列>)
<文 字 列>.gsub(<正 規 表 現>, <置 換 文 字 列>)
```

String#sub と String#gsub の例

```
1 str = "Sep. 29, 1970"
2 puts str.sub(/[789]/, "X")
3 puts str.gsub(/[789]/, "X")
4 puts str.sub(/[a-z]+/, "-")
```

実行例

```
Sep. 2X, 1970
Sep. 2X, 1XX0
S-. 29, 1970
```

[40] 例えば、7 とマッチしたら X、8 とマッチしたら Y、9 とマッチしたら Z に置換したいというように、マッチした結果を置換に反映させたい場合は、<文字列>.sub(<正規表現>) { ブロック } という構文が利用できる。ブロックの引数にマッチした文字列が設定されるので、それを利用して式を計算する。ブロックの返り値が置換文字列として扱われる。また、単に、String#tr メソッドで処理できる場合もある。

(14.8) **分解** String#split メソッドを使うと、正規表現にマッチする部分で文字列を分解し、それらを配列が返る。

String#split の文法

```
<文 字 列>.split(<正 規 表 現>)
```

String#split の例

```
1 str = "Sep. 29, 1970"
2 a = str.split(/ /)
3 p a
4 b = str.split(/[^a-zA-Z0-9]/)
5 p b
```

実行例

```
["Sep.", "29,", "1970"]
["Sep", "", "29", "", "1970"]
```

正規表現を与える代わりに文字列を与えると、その文字列に一致する箇所で分解される。正規表現を与える代わりに空白文字 1 文字" "を与えると、文字列の先頭と末尾の空白を除去した上で、空白文字 (スペースやタブなど) の連続で分解する。正規表現を与える代わりに空文字列""を与えると、文字列を 1 文字ずつに分解する。

String#split の例

```
1 str = "a b  c"
2 a = str.split("b")
3 p a
4 b = str.split(" ")
5 p b
```

実行例

```
["a ", "  c"]
["a", "b", "c"]
```

(14.9) **課題 29 – 置換と分解** キーボードから英文 (単語、スペース、 カンマ、ピリオドからなる列) を入力すると、すべてのピリオドを感嘆符「!」に置換した英文を表示し、さらに、単語数を下の例のように表示するプログラムを作成し提出せよ。

We see... という入力に対する実行例

```
We see a full moon the night of a lunar eclipse.
We see a full moon the night of a lunar eclipse!
単語数 11
```

ただし、1 行目は人間がキーボードから入力したものである。

(14.10) **課題 30 – フォーマット** キーボードから、いくつかの数値をスペース区切りで入力すると、その平均を表示するプログラムを作成し提出せよ。ただし、平均はフォーマットの機能を利用して小数点以下 3 桁までを表示すること。

実行例

```
1 23 45 67 8 89
平均は38.833
```

ただし、1 行目は人間がキーボードから入力したものである。

§15 クラス

(15.1) **クラス** コンピュータ言語における**クラス**とは、同じ種類のオブジェクトの集まりに名前を付けたものというような意味である[41]。あるクラスに属するオブジェクトを、そのクラスの**インスタンス**と呼ぶ[42]。

[41] class という英単語の和訳で言えば「類」が相当すると思われる。

[42] そのクラスのオブジェクトと呼んでも意味は通じる。オブジェクトという語はどのクラスかあまり気にしておらず、インスタンスという語はどのクラスかを強調したいような印象を与える。

主なクラスとそのインスタンスの例

クラス	意味	インスタンス
Fixnum	整数	`1`, `-3`, `1073741823`
String	文字列	`"a"`, `"あいう"`
Float	浮動小数点数	`1.1`, `-3e+2`
Array	配列	`[1,2]`, `["a", 1.1, [1,2]]`
Range	範囲	`(1..10)`, `("a".."z")`
Regexp	正規表現	`/abc/`, `/[1-9]+[a-z]*/`

クラスシステムの利点には、モジュール化 (再利用の容易さ)、カプセル化 (実装の隠蔽による保守の容易さ)、多態性 (同一の機能を同一の名前で実装できる) などがあり、Ruby のプログラムを作成する場合、意識せずともこれらのクラスシステムの利点の恩恵を受けることになる[43]。例えば、どのクラスのオブジェクトでも `to_s` で文字列に変換できるだとか、`puts` (プットエス) で表示できるだとかは、多態性の利点である。

(15.2) **クラスの作成** Ruby では多くのオブジェクト指向言語と同様に、自分でクラスを作成できる[44]。

簡単な例として、名前と生年月日をデータに持つような Person クラスを作成してみる。クラス名はアルファベット大文字で開始しなくてはならない。

クラス定義の文法

```
class <クラス名>
  <処理>
end
```

クラスがあるとき、そのインスタンスは `new` メソッドで作成する[45]。

[43] Ruby で「気軽に」プログラムを組む場合は、組み込みクラスを利用するだけでも、十分クラスシステムの利点を享受できる。クラスの作成 (後述) まで必要になるのは、ある程度の大きさを持つプログラムである

[44] クラスシステムを備えた言語においては、大規模なプログラムの作成は、クラスを設計することと言ってもよい。

[45] これは組み込みも含めてほとんどのクラスに共通である。例えば、`a = Array.new` は `a = []` と等価であり、`re = Regexp.new("b")` は `re = /b/` と等価である。

クラスのインスタンス作成の文法

```
<クラス名>.new(<引数>, ..)
```

クラス定義の文法の<処理>の部分に関数定義のように書くと、そのクラスのメソッドが定義される。initialize メソッドは特別であり、new メソッドを呼んだときに実行される。

Person クラスの定義の例

```
1  class Person
2    def initialize(person_name, person_birth)
3      @name = person_name
4      @birth = person_birth
5    end
6
7    def to_s
8      "%s さんは %s 生まれ" % [@name, @birth]
9    end
10 end
```

@が先頭についた変数は**インスタンス変数**と呼ばれ、インスタンスごとに (名前や生年月日のように) 異なるが、1 つのインスタンスの各メソッドの間では共通な変数である。また、to_s メソッドを定義しておくと、puts (プットエス) で表示するときに利用される。

Person クラスの利用の例

```
1  man = Person.new("藤井猛", "1970/9/29")
2  woman = Person.new("クルム伊達", "1970/9/28")
3  puts man, woman
```

実行例

```
藤井猛さんは1970/9/29生まれ
クルム伊達さんは1970/9/28生まれ
```

(15.3) **オープンクラス** Ruby では、1 度定義し終わったクラスに定義を追加することができる。この概念を**オープンクラス**と呼ぶ。これは組み込みクラスにもあてはまる[46]。

例えば、String クラスに、あいさつを表示するメソッドを追加してみる。

String#hi の例

```
1  class String
2    def hi
3      puts "%s さんこんにちは" % self
4    end
5  end
6
7  a = "藤井猛"
8  a.hi
```

実行例

```
藤井猛さんこんにちは
```

self は、自分自身 (上の例では、文字列そのもの) を表す組み込みの値である。

(15.4) **課題 31# – クラス** 縦と横の長さをインスタンス変数として持ち、to_s メソッドと面積を返す area メソッドを持つ長方形のクラスを定義せよ。具体的には、次のように利用できるクラス Rect を定義せよ。

クラス Rect の利用例

```
1  r = Rect.new(10,20)
2  puts r
3  puts r.area
```

実行例

```
縦10横20の長方形
200
```

Rect クラスの定義と上の利用例 (数値 10 と 20 は任意でよい) を、1 つのファイルに順に記述して提出せよ。

[46] これにより、組み込みのクラスを破壊することも可能であり、Ruby の柔軟性を示している

§16 連想配列 (Hash)#

(16.1) **連想配列のリテラルと値へのアクセス** 配列では、要素のインデックスが 0 から始まる連続した整数であった。連想配列は任意のオブジェクトをインデックスに用いることのできるコンテナである。連想配列のインデックスを**キー**と呼び、キーに対応する要素を**値**と呼ぶ[47]。

連想配列のリテラルの文法

```
{<キー1> => <値1>, <キー2> => <値2>, ..}
```

連想配列の値を得るには、下のように角かっこで囲んでキーを渡す。

連想配列の利用例

```
1 h = {1 => "藤井", "森下" => "システム"}
2 puts h[1], h["森下"]
```

実行例

```
藤 井
シ ス テ ム
```

(16.2) **キーの存在の判定** キーが連想配列に存在しないとき、その値を求めると nil になる。これを利用してキーの存在を判定することもできる。値が nil である可能性があれば、この方法では判定できないが、その場合は Hash#has_key? メソッドを用いるとよい。

キーの存在判定の例

```
1 h = {"藤井" => "猛", "羽生" => "善治"}
2 if h["伊達"] == nil
3   puts "なし"
4 end
5 if h.has_key?("伊達")
6   puts "あり"
7 end
```

[47] 連想配列からキーに対応する値を検索するときに、ハッシュ関数と呼ぶ関数を利用することが、クラス名の由来である。

実行例

```
な し
```

(16.3) **要素の上書きと追加** 連想配列 hash があるとき、hash[<キー>] に <値>を代入すると、要素が上書きされる。また、そのキーが存在しなかったときは、新たな要素として追加される。

連想配列の要素の上書きと追加の例

```
1 h = {"藤井" => "猛", "羽生" => "善治"}
2 h["藤井"] = "システム"
3 h["クルム伊達"] = "公子"
4 puts h["藤井"], h["クルム伊達"]
```

実行例

```
猛
公 子
```

(16.4) **連想配列の要素の反復操作** 連想配列に each メソッドを使用すると、キーと値の 2 引数がブロックに渡る。

Hash#each の例

```
1 h = {"藤井" => "猛", "羽生" => "善治"}
2 h.each {|k, v|
3   puts "%s %s" % [k, v]
4 }
```

実行例

```
藤 井 猛
羽 生 善 治
```

(16.5) **課題 32# – 単語数のカウント** キーボードから入力した文字列を英文とみて、単語ごとの出現回数を表示するプログラムを作成し提出せよ。

実行例

```
We see a full moon the night of a lunar eclipse.
lunar       1
of          1
moon        1
a           2
We          1
night       1
the         1
eclipse     1
full        1
see         1
```

ただし、1 行目は人間がキーボードから入力したものである。

§17 数値計算

(17.1) **自然対数の底 e の計算** 自然対数の底 $e = 2.718281828\cdots$ を、

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \cdots$$

を用いて Float (組み込みの浮動小数点数) の精度で計算してみる。

e の計算の例

```
1  def fact(n) # nの階乗を返す
2    if n == 0
3      1
4    else
5      n * fact(n-1)
6    end
7  end
8
9  e = 0.0
10 (0..20).each {|n|
11   e = e + 1.0 / fact(n)
12 }
13 puts e
```

実行例

```
2.7182818284590455
```

上の例では 1/20! までを加えているが、級数をどこで打ち切るかで精度が違ってくる。

上の例では、わかり易さを優先したため効率は悪い[48]。

(17.2) **BigDecimal** Float よりも精度が必要な場合は、いくつか方法が考えられるが、Ruby の添付ライブラリの BigDecimal クラスを用いるのが簡便である。BigDecimal は、任意の精度を持つ 10 進表現された浮動小数点数のクラスである。ただ、初期状態では Ruby に読み込まれていないため、利用するにはプログラムの冒頭で `require "bigdecimal"` と記述して BigDecimal クラスを読み込む必要がある。BigDecimal クラスの数を作成するには、専用のリテラルがないので、

BigDecimal の数の作成の文法

```
a = BigDecimal("1.2")
```

のようにする。BigDecimal の精度を保って演算するには、

BigDecimal の演算の文法

```
b = a.mult(3, 100)
c = a.div(7, 100)
```

のように、メソッド BigDecimal#mult や BigDecimal#div などを用いて、精度も指定すればよい。BigDecimal クラスを用いて自然対数の底 e を計算してみる。

[48] まず、階乗を求めるなら再帰よりは繰り返しの方が効率が良い。また、階乗を毎回 1 から n まで掛けているが、n の階乗を計算した次の繰り返しでは $n+1$ の階乗を計算するわけだから、$n!$ を変数に保存しておいて、それに $n+1$ を掛けるだけで $(n+1)!$ を求めた方が効率が良い。

BigDecimal を利用した e の計算例

```
require "bigdecimal"
zero = BigDecimal("0.0")
one = BigDecimal("1.0")

def fact(n) # nの階乗を返す （階乗はFixnumで十分）
  if n == 0
    1
  else
    n * fact(n-1)
  end
end

e = zero
(0..20).each {|n|
  e = e + one.div(fact(n), 100)
}
puts e
```

実行例

```
0.27182818284590452353397844906664158861464034345402617207765517901788134377596413222879533900368485014923465960754714514E1 # 長いので改行した
```

出力の最後の「E1」は、「$\times 10^1$」の意味である。また、ruby のバージョンなどの環境によっては、表示結果が異なる。

(17.3) **課題 33 円周率の計算** 円周率 π を、

$$\pi = 4\left(\frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \cdots\right)$$

を用いて計算し、表示するプログラムを作成し提出せよ。利用するのは Float でも BigDecimal でも構わないが、小数第 4 位までは正確な値が求まるようにせよ。

(17.4) **課題 34# 円周率の計算その 2** 円周率 π の、より効率のよい公式

$$\pi = 16\tan^{-1}\frac{1}{5} - 4\tan^{-1}\frac{1}{239}$$

$$\text{ただし、} \tan^{-1} x = \frac{1}{1}x - \frac{1}{3}x^3 + \frac{1}{5}x^5 - \frac{1}{7}x^7 + \cdots$$

を用いて、計算し表示するプログラムを作成し提出せよ。少くとも小数第 20 位までは正確な値になるようにせよ。

(17.5) **ニュートン法** 数値計算の代表的な方法として、関数 $f(x)$ の零点、つまり、方程式 $f(x) = 0$ の解を求めるための**ニュートン法**がある。

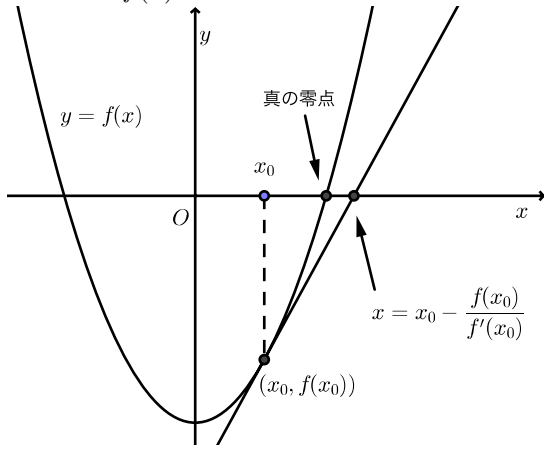
関数 $y = f(x)$ が零点を持ち、その零点の近くでは $f(x)$ は微分可能で、導関数 $f'(x)$ は 0 にならないとする。零点に十分近いと思われる $x = x_0$ をとると、$y = f(x)$ の $x = x_0$ における接線の方程式は、

$$y - f(x_0) = f'(x_0)(x - x_0)$$

となり、この直線の x 切片を求めると、$y = 0$ とおくことより、

$$x = x_0 - \frac{f(x_0)}{f'(x_0)} \qquad (\#)$$

となる。この x 切片は、$f(x)$ の零点に近いと考えられる[49]。



[49]一般的な関数ならば x_0 よりも真の零点に近くなるが、そうならないこともある。

つまり、上の (#) 式を用いて、x_0 を x に取り替えることを反復すると、徐々に真の零点に近付いていく。取り替えても変化しないならば、真の零点に到達したと考えてよい。実際には誤差があるので、$f(x_0)/f'(x_0)$ が十分小さくなったら、真の零点の近似値が得られたと考えてよい。

$\sqrt{2}$ は x^2-2 の正の零点だから、ニュートン法で計算するプログラムは以下のようになる。小数点以下 6 桁は正しく求まるように計算している。

ニュートン法による $\sqrt{2}$ の計算例

```
1  def f(x)
2    x**2 - 2
3  end
4
5  def df(x)
6    2*x
7  end
8
9  x = 2.0
10 while true
11   dx = f(x) / df(x)
12   if dx.abs < 0.00000001
13     break
14   end
15   x = x - dx
16 end
17 puts x
```

実行例

```
1.4142135623746899
```

環境により、多少結果が異なる可能性がある。

(17.6) **課題 35 ニュートン法その 1** ニュートン法を用いて、小数第 6 位まで正確に、$\sqrt[3]{3}$ を求めよ。

実行例

```
1.4422495703074079
```

(17.7) **課題 36# ニュートン法その 2** 整数 n をキーボードからすると、ニュートン法を用いて、$\sqrt{n}$ を求め、表示するプログラムを作成し提出せよ。

実行例

```
3
1.7320508075688772
```

ただし、1 行目は人間がキーボードから入力したものである。

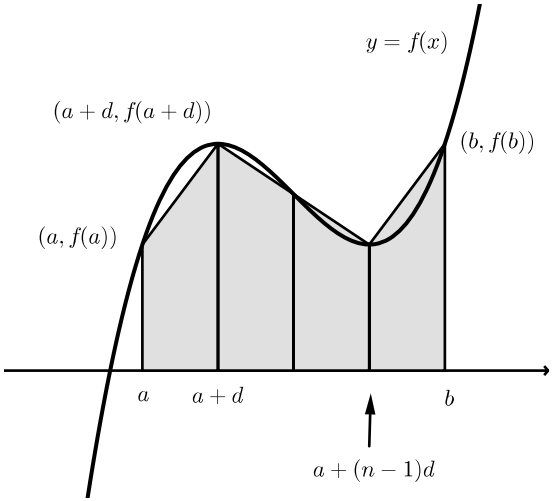
(17.8) **課題 37# ニュートン法その 3** ニュートン法を用いて、3 次方程式 $x^3-3x+1=0$ の実数解をすべて求めよ。ニュートン法を 1 度実行するだけでは、零点は 1 つしか求まらないので、x の初期値を変更して 3 度ニュートン法を実行すること。

実行例

```
0.894427190999916
3.60555127546399
-1.18321595661992
```

ただし、上の例の値は適当です。

(17.9) **台形公式** # **台形公式**とは、以下のように、定積分をいくつかの台形の面積で近似する方法である。関数 $y=f(x)$ が、閉区間 $a \leq x \leq b$ で定積分可能であるとする。この閉区間を n 等分し、グラフと x 軸の間の領域を n 個の台形で近似してその面積を求めることで、定積分 $\int_a^b f(x)\,dx$ の近似値とすることができる。



台形の底辺の長さを d とすると、$d = (b-a)/n$ であるから、定積分を図の n 個の台形の面積の和で近似すると以下のようになる。

$$\begin{aligned}
&\int_a^b f(x)\,dx \\
&\fallingdotseq \frac{d(f(a)+f(a+d))}{2} + \frac{d(f(a+d)+f(a+2d))}{2} + \cdots \\
&\qquad + \frac{d(f(a+(n-1)d)+f(a+nd))}{2} \\
&= \frac{d}{2}\Big(f(a) + 2f(a+d) + 2f(a+2d) + \cdots + 2f(a+(n-1)d) + f(b)\Big)
\end{aligned}$$

台形公式を用いて、単位円の面積のうち、第 1 象限にある部分の面積を求めて、その 4 倍を表示してみる。

台形公式による π の計算例

```
1  def f(x)
2    (1-x**2)**0.5
3  end
4
5  a = 0.0
6  b = 1.0
7  n = 1000
8  d = (b-a) / n
9
10 area = f(a) + f(b)
11 for i in 1..(n-1)
12   area = area + 2 * f(a+i*d)
13 end
14 area = area * (d/2)
15
16 puts area * 4
```

実行例

```
3.141555466911023
```

(17.10) **課題 38# 台形公式** 台形公式を用いて、短軸半径 1、長軸半径 2 である楕円

$$\begin{cases} x = \cos t \\ y = 2\sin t \end{cases}$$

の周の長さを求め、表示せよ。ただし、この楕円の第 1 象限にある部分の周の長さは、

$$\int_0^{\pi/2} \sqrt{\sin^2 t + 4\cos^2 t}\,dt$$

で計算できる。三角関数は、`Math.sin(x)`, `Math.cos(x)` を、円周率は、`Math::PI` を用いよ。

実行例

```
9.688448220547674
```

ただし、積分区間をいくつに分割するかで、出力は異なる。

§18 アルゴリズム

(18.1) **アルゴリズム** 計算機に何かの処理をさせるときの、処理の手順のことを**アルゴリズム**と呼ぶ。

2 つの正整数の最大公約数を求めるには、(10.6) や (10.7) のときのように、約数を列挙して、共通なもののうち最大のものを求めるアルゴリズムもあれば、(11.6) のユークリッドの互除法もある。紀元前からあるユークリッドの互除法は、現在でも最速のアルゴリズムである。

また、方程式の解の近似値を求めるニュートン法も、効率の良いアルゴリズムである。

(18.2) **単純な探索** 数値の配列に、特定の数 x が属するかどうかを調べるとき、基本的には先頭の要素から順に x と一致するかどうか調べるしか手段はない。ただし、Ruby の配列では、include? というメソッドで、配列に属するかどうかを調べられるから、このコードはアルゴリズムの練習のためである。

単純な探索

```
1  def search(x, ary) # xがaryに属するならtrueを返す
2    for a in ary
3      if x == a
4        return true
5      end
6    end
7    false
8  end
9
10 ary = [3, 1, 4, 1, 5, 9, 2, 6, 5]
11 x = gets.to_i
12 puts search(x, ary)
```

実行例その 1

```
6
true
```

実行例その 2

```
7
false
```

ただし、ともに 1 行目は人間がキーボードから入力したものである。

(18.3) **二分探索** 配列が整列済みならば、探索範囲を半分ずつに絞っていく、**二分探索**と呼ばれるアルゴリズムがある。配列に属するかどうか調べたい x を先頭ではなく、まず中央の要素 y と比較して、x の方が小さければ、配列の前半半分を調べればよいし、大きければ後半半分を調べればよい、という原理のアルゴリズムである。

二分探索による探索

```
1  def bsearch(x, ary) # xがaryに属するならtrueを返す
2    if ary.size == 0
3      return false
4    end
5    center = ary.size / 2
6    y = ary[center]
7    if x == y
8      true
9    elsif x < y
10     bsearch(x, ary[0...center])
11   else
12     bsearch(x, ary[center+1..-1])
13   end
14 end
15
16 ary = [3, 1, 4, 1, 5, 9, 2, 6, 5].sort
17 x = gets.to_i
18 puts bsearch(x, ary)
```

実行例は単純な探索と同じである。また、0...center は 0...center-1 と同じ意味である。配列の要素数が小さいうちは、単純な探索と二分探索の実行速度に大差はないが、要素数が巨大になれば決定的な差が開く[50]。

[50] 実行速度のことを考えるならば、掲げたコード例はまだ効率が悪い。特に半分の長さの配列を作成している所と、再帰をしている所が効率が悪い (再帰は、繰り返しと比べて一般にはメ

(18.4) **計算量** 単純な探索と二分探索を比べると、二分探索の方が高速なアルゴリズムと言えるが、その速さを定式化したものが**計算量**である。計算量といっても、実行時間、実行される命令数、消費されるメモリ量など、何をカウントするかで種類があるが、ここでは、大小比較の回数に基いた計算量を考える。

単純な探索では、配列の長さが n ならば、探索が成功する場合は最悪 n 回の大小比較がなされる。二分探索では、配列の長さが $n = 2^m$ ならば、長さが m 回半分になれば要素が 1 つになるので、最悪 $m+1 = \log_2 n + 1$ 回の大小比較がなされる。n が大きくなったときに無視できる項や、比例定数を無視して、それぞれ、(最悪) 計算量が $O(n)$, $O(\log n)$ であると言う。

アルゴリズム	単純な探索	二分探索
最悪計算量	$O(n)$	$O(\log n)$

(18.5) **整列アルゴリズム** Ruby の配列の sort メソッドに相当する、配列の整列のアルゴリズムを考える。主なアルゴリズムとしては、**バブルソート**、**選択ソート**、**挿入ソート**、**シェルソート**、**マージソート**、**ヒープソート**、**クイックソート**などがある。これらの (最悪) 計算量は以下の通りである。

アルゴリズム	バブル	選択	挿入	シェル
最悪計算量	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n(\log n)^2)$

	マージ	ヒープ	クイック
	$O(n \log n)$	$O(n \log n)$	$O(n^2)$

以下では、これらの一部を扱う。

モリ効率が悪い)。配列を作成する代わりにどの範囲を探索中かを変数で管理すれば、配列を作成する必要はなくなる。また、再帰はこの場合は繰り返しに書き換えることができる。Ruby ではサポートされていないが、このコードのような再帰 (末尾再帰) を、内部で自動的に繰り返しに変換して実行する言語もある。

(18.6) **バブルソート** 長さ n の配列 a があるとする。バブルソートのアルゴリズムは以下の通りである。バブルソートは効率の悪いソートの代名詞である。

第 1 段 (1) a[0] と a[1] を比較して、前者が大きければ両者を交換する。
(2) a[1] と a[2] を比較して、前者が大きければ両者を交換する。
(3) これを、a[n-2] と a[n-1] の比較・交換まで行う。

第 2 段 再び、a[0] と a[1] の比較・交換をし、今度は、a[n-3] と a[n-2] の比較・交換まで行う。

第 3 段以降 これを、比較・交換の回数を 1 ずつ減じつつ、$n-1$ 段まで繰り返す。

第 1 段が終了すると、最後の要素が最大値になっている。同じ理屈で、第 2 段が終了すると、最後から 2 番目の要素が、大きい方から 2 番目の要素になる。これを $n-1$ 段まで実行すると整列が完了する。下の図ははバブルソートの過程の例である。

⑤ ③ 2 8 9 1 7 隣接要素で大小が逆転しているものを交換していく
3 ⑤ ② 8 9 1 7
3 2 5 8 ⑨ ① 7
3 2 5 8 1 ⑨ ⑦
③ ② 5 8 1 7 9 最大値が末尾に来た。再び先頭から比較・交換する
2 3 5 ⑧ ① 7 9
2 3 5 1 ⑧ ⑦ 9
2 3 ⑤ ① 7 8 9 2 番目に大きい値が末尾から 2 番目に来た。以下同様
2 ③ ① 5 7 8 9
② ① 3 5 7 8 9
1 2 3 5 7 8 9

バブルソート

```
1 def bubblesort(ary)
2   (ary.size-2).downto(1) {|tail|
3     for i in (0..tail)
4       if ary[i] > ary[i+1]
```

```
5           ary[i], ary[i+1] = ary[i+1], ary[i]
6         end
7       end
8     }
9     ary
10  end
11
12  ary = [5, 3, 2, 8, 9, 1, 7]
13  p bubblesort(ary)
```

実行例

```
[1, 2, 3, 5, 7, 8, 9]
```

上のコードにおいて、

downto

```
x.downto(y) {|i| ブロック }
```

は、カウンタ変数 i が x から y まで 1 ずつ減りながら、ブロックを繰り返す命令である。つまり、逆順の for や each に相当する。

また、

多重代入

```
x, y = a, b
```

は、x と y に、それぞれ、a と b を代入する命令である。

(18.7) **選択ソート** 長さ n の配列 a があるとする。選択ソートのアルゴリズムは以下の通りである。

(1) 最小値を探し、a[0] と交換する。
(2) a[1] 以降の要素の最小値を探し、a[1] と交換する。
(3) これを最後まで反復する。

下の図は選択ソートの過程の例である。

5	3	2	8	9	①	7	最小値①を先頭と交換
1	3	②	8	9	5	7	a[1] 以降の最小値②を a[1] と交換
1	2	③	8	9	5	7	a[2] 以降の最小値③を a[2] と交換
1	2	3	8	9	⑤	7	以下同様
1	2	3	5	9	8	⑦	
1	2	3	5	7	⑧	9	
1	2	3	5	7	8	9	

選択ソート

```
1   def selectionsort(ary)
2     for i in (0..ary.size-2)
3       min = ary[i]
4       min_pos = i
5       for j in (i+1..ary.size-1)
6         if min > ary[j]
7           min = ary[j]
8           min_pos = j
9         end
10      end
11      ary[i], ary[min_pos] = ary[min_pos], ary[i]
12    end
13    ary
14  end
15
16  ary = [5, 3, 2, 8, 9, 1, 7]
17  p selectionsort(ary)
```

(18.8) **課題 39# 選択ソート** 上のプログラムでは、最小値を求めるのに Array#min メソッドを用いていない。Array#min メソッドを用いて書き直したプログラムを作成し提出せよ。また、最小値の位置も知る必要があるが、それには Array#index メソッドを用いよ。

実行例

```
[1, 2, 3, 5, 7, 8, 9]
```

(18.9) **挿入ソート** 長さ n の配列 a があるとする。挿入ソートのアルゴリズムは以下の通りである。

(1) a[0] と a[1] を比較し整列する。
(2) a[0..i] が整列されたとして、a[i+1] を、a[0..i] の適切な位置に挿入し、a[0..i+1] を整列済みにする。
(3) これを最後まで反復する。

下の図は挿入ソートの過程の例である。丸数字を、それより左にある整列済みの列に挿入することを反復する。

5	③	2	8	9	1	7
3	5	②	8	9	1	7
2	3	5	⑧	9	1	7
2	3	5	8	⑨	1	7
2	3	5	8	9	①	7
1	2	3	5	8	9	⑦
1	2	3	5	7	8	9

(18.10) **課題 40# 挿入ソート** 挿入ソートをする関数 insertionsort を作成し、それを用いて適当な配列 ([5, 3, 2, 8, 9, 1, 7] など) を整列して表示するプログラムを作成し提出せよ。

実行例

```
[1, 2, 3, 5, 7, 8, 9]
```

(18.11) **マージソート** 長さ n の配列 a があるとする。マージソートのアルゴリズムは再帰的であり、以下の通りである。

(1) 配列を前半と後半に分割する。
(2) それぞれをマージソートする。
(3) それらを、合併する。

下の図はマージソートの過程の例である。

まず分割をする

5	3	2	8	9	1	7
5	3	2 |	8	9	1	7
5 |	3	2 |	8	9 |	1	7
5 |	3 |	2 |	8 |	9 |	1 |	7

マージを始める

5 |	2	3 |	8	9 |	1	7
2	3	5 |	1	7	8	9
1	2	3	5	7	8	9

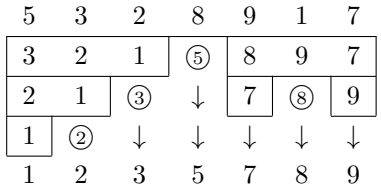
マージソート

```
1  def mergesort(ary)
2    len = ary.size
3    if len <= 1
4      return ary
5    end
6    ary1 = mergesort(ary[0...len/2])
7    ary2 = mergesort(ary[len/2..-1])
8    ary3 = []
9    loop {
10     if ary1.size == 0
11       return(ary3 + ary2)
12     elsif ary2.size == 0
13       return(ary3 + ary1)
14     elsif ary1[0] <= ary2[0]
15       ary3.push(ary1.shift)
16     else
17       ary3.push(ary2.shift)
18     end
19   }
20 end
21
22 ary = [5, 3, 2, 8, 9, 1, 7]
23 p mergesort(ary)
```

(18.12) **クイックソート** クイックソートは最悪計算量は他のソートアルゴリズムに劣るが、平均的には最も良い性能を持つとされる。長さ n の配列 a があるとする。クイックソートのアルゴリズムは再帰的であり、以下の通りである。

(1) 配列を、先頭の要素 a[0] と、a[0] 未満の要素たちと、a[0] 以上の要素たちに 3 分割する。
(2) 「未満」と「以上」をクイックソートする。
(3) 整列した「未満」、a[0]、整列した「以上」の順に並べる。

下の図はクイックソートの過程の例である。丸数字が上の手順の a[0] に相当し、その左右に四角く囲んだ部分が、「未満」と「以上」に相当する。また、以降の再帰における整列に影響のない箇所は、下矢印で示した。



クイックソート

```
def quicksort(ary)
  if ary.size <= 1
    return ary
  end
  x = ary[0]
  ary1 = ary[1..-1].select {|a| a < x }
  ary2 = ary[1..-1].select {|a| a >= x }
  quicksort(ary1) + [x] + quicksort(ary2)
end

ary = [5, 3, 2, 8, 9, 1, 7]
p quicksort(ary)
```

(18.13) **課題 41$^{\#}$ クイックソート** 上のプログラムでは、a[0] 未満・以上の要素からなる配列 ary1, ary2 を作成するのに、Array#select メソッドを用いた。これを用いずに書き直したプログラムを作成し提出せよ。

実行例

```
[1, 1, 2, 3, 4, 5, 5, 6, 9]
```

(18.14) **各アルゴリズムの速度比較** これまでに紹介した 4 つのアルゴリズムによる整列関数で、1 万要素のランダムな浮動小数点数からなる配列を整列するのにかかった時間は、手元の環境では以下の通りである。参考のため、Array#sort メソッドに要した時間も記す。

アルゴリズム	バブル	選択	マージ	クイック	sort
時間 (秒)	18.69	6.05	0.07	0.05	0.01

§19 より進んだ課題

これまでに学んだことを踏まえて、進んだ課題を与える。この節の課題はシャープ 2 個 (##) 付きである。これまでのシャープ 1 個の「義務ではない課題」とは異なり、成績への算入方法も別である。

(19.1) **課題 42$^{\#\#}$ 分数の和** 最大公約数が 1 である、1 以上 99 以下の整数 a, b, c $(a \leq b)$ に対して、

$$\frac{1}{a} + \frac{1}{b} = \frac{1}{c}$$

を満たすものをすべて下のように表示するプログラムを作成し提出せよ。ただし、表示の順序は下の例と同じ順序でなくても構わない。

実行例

```
1/2 + 1/2 = 1/1
```

```
1/3 + 1/6 = 1/2
1/4 + 1/12 = 1/3
1/5 + 1/20 = 1/4
1/6 + 1/30 = 1/5
1/7 + 1/42 = 1/6
1/8 + 1/56 = 1/7
1/9 + 1/72 = 1/8
1/10 + 1/15 = 1/6
1/10 + 1/90 = 1/9
1/14 + 1/35 = 1/10
1/18 + 1/63 = 1/14
1/21 + 1/28 = 1/12
1/22 + 1/99 = 1/18
1/24 + 1/40 = 1/15
1/30 + 1/70 = 1/21
1/33 + 1/88 = 1/24
1/36 + 1/45 = 1/20
1/44 + 1/77 = 1/28
1/55 + 1/66 = 1/30
1/60 + 1/84 = 1/35
1/78 + 1/91 = 1/42
```

(19.2) **課題 43## チャンパーノウン定数** 「0.」に続けて、正整数を小さい順に書いて連結した数

$$0.12345678910111213141516171819202122232425262728293 0\cdots$$

をチャンパーノウン定数と呼ぶ。この小数部分を 1 桁ずつ取り出した数列

$$1,2,3,4,5,6,7,8,9,1,0,1,1,1,2,1,3,1,4,1,5,1,6,1,7,1,8,1,9,\ldots$$

を $\{c_n\}$ と表す。キーボードから数値 n を入力すると、c_n を表示する プログラムを作成し提出せよ。

実行例その 1

```
11
0
```

実行例その 2

```
1111
4
```

(19.3) **課題 44## 日曜日の回数** キーボードから 1900 以上の西暦を入力すると、その年の日曜日の回数を表示するプログラムを作成し提出せよ。ただし、1900 年 1 月 1 日は月曜日であることを用い、西暦が次の条件を満たすときを閏年とせよ (グレゴリオ暦)、

> 西暦が 4 の倍数の年を閏年とする。ただし、そのうち、100 の倍数の年は平年とする。ただし、さらに、そのうち、400 の倍数の年は閏年とする。

実行例その 1

```
1971
52
```

実行例その 2

```
2000
53
```

ただし、ともに 1 行目は人間がキーボードから入力したものである。

(19.4) **課題 45## 友愛数** 友愛数とは、異なる 2 つの正整数の組であり、自分自身を除いた約数の和が、互いに他方と等しくなるような数を言う。例えば 220 と 284 は友愛数である。なぜなら、220 の自分自身以外の約数の和は、$1+2+4+5+10+11+20+22+44+55+110=284$ であり、284 の自分自身以外の約数の和は、$1+2+4+71+142=220$ だからである。

4 桁の整数 2 つからなる友愛数の組をすべて求め、表示するプログラムを作成し提出せよ。

実行例

```
1184 1210
2620 2924
5020 5564
:
(後略)
```

(19.5) **課題 46## 辞書式順序** キーボードから、異なるアルファベットからなる単語を入力すると、その単語のアルファベットを並べ換えてできるすべての単語を辞書式順序で並べたとき、元の単語は何番目に現れるかを表示するプログラムを作成し提出せよ。

実行例

```
bca
4
```

ただし、1 行目は人間がキーボードから入力したものである。(b, c, a を並べ換えてできる単語を辞書式順序に並べると

abc, acb, bac, bca, cab, cba

であり、bca はそのうち 4 番目であるから、4 と表示されている。

(19.6) **課題 47## フィボナッチ数列** キーボードから正整数 a を入力すると、フィボナッチ数列の項であって、a 以上になる最初のものを表示するプログラムを作成し提出せよ。

実行例その 1

```
10
13
```

実行例その 2

```
10000
10946
```

ただし、ともに 1 行目は人間がキーボードから入力したものである。

(19.7) **課題 48## 循環節** キーボードから正整数 a を入力すると、分数 $1/a$ を小数で表示したとき、有限小数になるならば「有限小数」と表示し、循環小数になるならばその循環節の長さを表示するプログラムを作成し提出せよ。

実行例その 1

```
250
有限小数
```

実行例その 2

```
13
6
```

ただし、ともに 1 行目は人間がキーボードから入力したものである。

(19.8) **課題 49## 硬貨の和** キーボードから正整数 a を入力すると、a を 1, 5, 10 の和で表す方法を、すべて下のように表示するプログラムを作成し提出せよ。ただし、表示の順序はこの通りでなくてもよい。

実行例その 1

```
12
10x0 + 5x0 + 1x12
10x0 + 5x1 + 1x7
10x0 + 5x2 + 1x2
10x1 + 5x0 + 1x2
```

実行例その 2

```
30
10x0 + 5x0 + 1x30
10x0 + 5x1 + 1x25
10x0 + 5x2 + 1x20
10x0 + 5x3 + 1x15
10x0 + 5x4 + 1x10
10x0 + 5x5 + 1x5
10x0 + 5x6 + 1x0
10x1 + 5x0 + 1x20
```

```
10x1 + 5x1 + 1x15
10x1 + 5x2 + 1x10
10x1 + 5x3 + 1x5
10x1 + 5x4 + 1x0
10x2 + 5x0 + 1x10
10x2 + 5x1 + 1x5
10x2 + 5x2 + 1x0
10x3 + 5x0 + 1x0
```

ただし、ともに 1 行目は人間がキーボードから入力したものである。

(19.9) **課題 50## シーザー暗号** a を b に、b を c に、そして z を a に置き換えるというような字の置き換えによる暗号をシーザー暗号と呼ぶ。アルファベットからなる単語をキーボードから入力すると、アルファベットを 4 つ後ろへずらして (z を越えた場合は a へ循環させて) 暗号化した単語を表示するプログラムを作成し提出せよ。

実行例

```
xylophone
bcpstlsri
```

ただし、1 行目は人間がキーボードから入力したものである。

(19.10) **課題 51## ベズーの方程式** 互いに素な 0 ではない 2 つの整数 a, b をキーボードから入力すると、ベズーの方程式

$$ax + by = 1 \quad (x, y \text{ は整数})$$

の解 (x, y) を 1 つ表示するプログラムを作成し提出せよ。

実行例

```
81
25
(x,y) = (21, -68)
```

ただし、1, 2 行目は人間がキーボードから入力したものである。

(19.11) **課題 52## 連分数** 有理数 a/b は、例えば次のように連分数で表すことができる。

$$\frac{118}{31} = 3 + \frac{1}{1 + \frac{1}{4 + \frac{1}{6}}}$$

この右辺を [3, 1, 4, 6] と書くことにする。

キーボードから 2 つの正整数 a, b を入力すると、有理数 a/b を連分数で表示した結果を下のように表示するプログラムを作成し提出せよ。

実行例その 1

```
81
25
[3, 4, 6]
```

実行例その 2

```
118
31
[3, 1, 4, 6]
```

ただし、ともに、1, 2 行目は人間がキーボードから入力したものである。

(19.12) **課題 53## 分割数** 正整数 n を、いくつかの正整数の和で表す場合の数を分割数と呼ぶ。例えば、7 は、

$$\begin{aligned} 7 &= 6+1 = 5+2 = 5+1+1 = 4+3 = 4+2+1 = 4+1+1+1 \\ &= 3+3+1 = 3+2+2 = 3+2+1+1 = 3+1+1+1+1 \\ &= 2+2+2+1 = 2+2+1+1+1 = 2+1+1+1+1+1 \\ &= 1+1+1+1+1+1+1 \end{aligned}$$

と 15 通りに表せるので、7 の分割数は 15 である。

キーボードから正整数を入力すると、その分割数を表示するプログラムを作成し提出せよ。

実行例その 1

```
7
15
```

実行例その 2

```
100
190569292
```

ただし、ともに、1 行目は人間がキーボードから入力したものである。

§20 雑多な例

これまでに学んだことを踏まえて、いろいろなプログラム例を示す。文法についての詳しい解説は Ruby のウェブサイトのドキュメントを参照のこと。

(20.1) **じゃんけんゲーム** コンピュータがランダムに選んだ手と、人がキーボードから入力した手でじゃんけんをし、勝ち負けを表示する。

じゃんけんゲーム

```
1  print "グー(0) チョキ(1) パー(2): "
2  hand1 = gets.to_i
3
4  hand2 = rand(3)
5  s = ["グー", "チョキ", "パー"]
6  puts "わたしは%s" % s[hand2]
7
8  if hand1 == hand2
9    puts "引き分け"
10 elsif [hand1, hand2] == [0, 1] ||
11       [hand1, hand2] == [1, 2] ||
12       [hand1, hand2] == [2, 0]
13   puts "あなたの勝ち"
14 else
15   puts "あなたの負け"
16 end
```

rand(n) は、0 以上 n 未満の整数をランダムに返す関数である。コンピュータの手を表示する 6 行目では、0 ならばグー、1 ならばチョキ、2 ならばパーと if 文を連ねる代わりに、配列の要素の 0, 1, 2 番目を表示するようにしている。

実行例

```
グー(0) チョキ(1) パー(2): 1
わたしはグー
あなたの負け
```

(20.2) **魔方陣** 3×3 の魔方陣[51]をしらみつぶしに見付けてみる。下の図の a, b, c が決定すれば、他の位置は自動的に決まることも利用する。

a	b	U
Y	c	W
X	Z	V

3x3 魔方陣の例

```
1  nums1 = (1..9).to_a
2  (1..9).each {|a|
3    (1..9).each {|b|
4      (1..9).each {|c|
5        u = 15 - a - b
6        v = 15 - a - c
7        w = 15 - u - v
8        x = 15 - c - u
9        y = 15 - a - x
10       z = 15 - b - c
11       nums = [a, b, u, y, c, w, x, z, v]
12       if nums.sort == nums1 &&
13           y + c + w == 15 &&
14           x + z + v == 15
15         puts "%s %s %s\n%s %s %s\n%s %s %s"%nums
16         puts "-"*5
17       end
18     }
```

[51] 1 から 9 までの整数を 1 つずつ 3×3 に並べ、各行、各列、2 つの対角線の和がすべて等しいようにしたもの。

```
19    }
20  }
```

上のコードでは、a,b,c で 3 重のループを回して、u から z までを計算により求めている。12 行目からの if 文では、1 から 9 までが 1 回ずつ出現しているかと、和が 15 になることが未確認の 2 列を確認している[52]。

実行例の一部

```
2 7 6
9 5 1
4 3 8
-----
2 9 4
7 5 3
6 1 8
-----
:
中 略
:
-----
8 3 4
1 5 9
6 7 2
-----
```

(20.3) **ヒットアンドブロー** 各桁の数字が異なる 4 桁の数を決め、それを当てるヒットアンドブローというゲームがある。入力する度に、答えと位置も合っている数字 (ヒットと呼ぶ) の個数と、位置は合っていないが答えと同じ数字 (ブローと呼ぶ) の個数を表示し、それを手掛りに 4 桁の数を当てる。

ヒットアンドブロー

```
1  def digits(num)
2    [num/1000, (num/100)%10, (num/10)%10, num%10]
3  end
4
5  def print_hit_blow(a, b)
6    as = digits(a)
7    bs = digits(b)
8    hit = 0
9    (0..3).each {|i|
10     if as[i] == bs[i]
11       hit = hit + 1
12     end
13   }
14   same = (as & bs).size
15   puts "ヒット:%s ブロー:%s" % [hit, same-hit]
16 end
17
18 answer = 0
19 while digits(answer).uniq.size != 4
20   answer = rand(10000)
21 end
22
23 (1..999).each {|n|
24   print "入力%3s:" % n
25   input = gets.to_i
26   if digits(input).uniq.size != 4
27     puts "各桁は異なる数字で"
28   else
29     print_hit_blow(answer, input)
30     if answer == input
31       break
32     end
33   end
34 }
35
36 puts "正解"
```

関数 digits は、各桁の数を要素に持つ配列を返す関数である。関数 print_hit_blow では、ヒットとブローの数を表示する。8 から 13 行目でヒットの数を調べている。14 行目では、4 桁の数の配列 as と bs の共通部分の要素数を求めて、ブローの数を求めているが、このままだとヒットの数も数えてしまうので、15 行目でブローを表示するときは、ヒットの数を差し引いて

[52] サイズが小さい魔方陣なのでこのコードでも速度は十分早いが、効率の面ではいくつもの改善点がある。b のループでは、a と b が等しいときは解ではないことが直ちにわかるし、c についても同様である。また、1 から 9 までを 1 度ずつ使用しているかのチェックも、ソートという負担の重い処理をするのではなく、より効率のよい方法を検討する必要がある。

いる。18 行目から 21 行目では答を生成している。各桁の数が異なるものが得られるまでループしている。23 行目からがメインの処理で、999 回という十分多い回数のループにして、30 行目で正解だと判定されると 31 行目でそのループを脱出している。

実行例

```
入力  1:1234
ヒット:1 ブロー:1
入力  2:1425
ヒット:1 ブロー:1
入力  3:1674
ヒット:0 ブロー:1
入力  4:8935
ヒット:2 ブロー:0
入力  5:4035
ヒット:2 ブロー:1
入力  6:0135
ヒット:4 ブロー:0
正解
```

(20.4) **n クイーン** 8×8 のチェス盤に、8 個のクイーンを互いの効きに入らないように置くのが 8 クイーンと呼ばれる問題である。n クイーンとはサイズを $n \times n$ にした問題である。下のプログラムでは 1 行目の `N = 5` を書き換えると n を変更できる。

N クイーン

```
1  N = 5
2  def check(xs)
3    diag1 = (0...N).map {|i| xs[i] - i }
4    diag2 = (0...N).map {|i| xs[i] + i }
5    if diag1.uniq.size == N &&
6        diag2.uniq.size == N
7      print_board xs
8    end
9  end
10
11 def print_board(xs)
12   (0...N).each {|i|
13     print "."*xs[i], "Q", "."*(N-xs[i]-1), "\n"
14   }
15   puts
16 end
17
18 (0...N).to_a.permutation(N) {|xs|
19   check(xs)
20 }
```

18 行目からがメインであり、`Array#permutation(n)` は、要素数 `n` の順列をすべて生成し、ブロックに引数として渡すメソッドである。そうして得られるサイズ `N` の順列 `xs` は、0 から `N-1` までの整数を 1 つずつ含む配列である。これは、チェス盤を 0 行目から `N-1` 行目まであるとしたとき、`i` 行目の `xs[i]` 列目にクイーンがあることを意味する配列である[53]。あとは、斜め方向の効きに入らないことのチェックが必要だが、それは 2 行目から定義されている関数 `check` でチェックしている。具体的には、クイーンを通過する傾き 1 の直線を引きその y 切片を考えたとき、すべての y 切片が異なっていなくてはならず、傾き -1 の直線を考えても同様であることをチェックしている。11 行目からの関数 `print_board` は、盤を表示する。

5 クイーンの実行例

```
Q....
..Q..
....Q
.Q...
...Q.

Q....
...Q.
.Q...
....Q
..Q..

  :
```

[53] クイーンが互いの効きに入らないため、各行各列にクイーンが 1 つしかないことに注意する。

```
 中略
  :

....Q
..Q..
Q....
...Q.
.Q...
```

魔方陣のときのように、重複があろうとしらみつぶしに調べることも可能だが、Array#permutation メソッドを利用して、調べる場合を大幅に減らしているのがポイントである。

(20.5) **TEX ソースの自動出力** Ruby で TEX のソースを出力させて、例えば問題を自動で生成することもできる。ここでは、ランダムに掛け算の九九の問題を 10 問作成してみる。

TeX ソースの自動出力の例

```
1  puts "\\documentclass{jarticle}"
2  puts "\\begin{document}"
3  puts "\\begin{enumerate}"
4  (1..10).each {|i|
5    a = rand(9) + 1
6    b = rand(9) + 1
7    puts "\\item[(%s)]$%s \\times %s =$" % [i,a,b]
8  }
9  puts "\\end{enumerate}"
10 puts "\\end{document}"
```

文字列中のバックスラッシュは、「"\n"」のように特別な意味を持つので、その意味をなくし単なるバックスラッシュにするため、バックスラッシュを 2 度重ねている。

実行例

```
\documentclass{jarticle}
\begin{document}
\begin{enumerate}
\item[(1)]$3 \times 2 =$
\item[(2)]$6 \times 4 =$
\item[(3)]$1 \times 3 =$
\item[(4)]$9 \times 6 =$
\item[(5)]$9 \times 4 =$
\item[(6)]$5 \times 7 =$
\item[(7)]$2 \times 2 =$
\item[(8)]$2 \times 4 =$
\item[(9)]$3 \times 3 =$
\item[(10)]$6 \times 9 =$
\end{enumerate}
\end{document}
```

この出力を TEX ファイルにコピーアンドペーストしたり、リダイレクト[54]したり、Ruby のファイルへの出力の機能を用いてファイルに出力すればよい。

[54]出力先をコマンドプロンプトのウィンドウではなく、指定したファイルにするコマンドプロンプトのシェルの機能。

索 引

Symbols