講義使用向けの文書形態であることに注意して利用してください。 CC-BY での取り扱いとします。
2026-09-27 広瀬雄二

◆まえがき

コンピュータを使いこなしたいと思っている人は多いだろう。パソコンを使って見栄えのよい書類を作成し、好みの音楽をパソコンから流す。そんなことが手軽にできるようになってきた。技術の進歩に伴い、コンピュータを使った作業が飛躍的に分かりやすくなった。そのおかげで、以前は好奇心の強い粘り強い人でなければ使えなかったような代物が、現在では専門の勉強をしなくても、使い始めて間もないうちからそこそこ「使いこなす」ことができるようになっている。

> ほとんどの人ができている程度のことを「使いこなす」と言ってよいのだろうか?

これは「使いこなす」という言葉をどのレベルで考えるかによるだろう。とにかく仕事がこなせればよい、という考え方であれば他人ができているかは関係なく、目的が達成できることこそが使いこなしていることにほかならない。

しかし、本書を目の前にしている読者はそうは考えないだろう。多くの人ができていること、つまり平易なことではあきたらない人にとっての「使いこなす」とは、対象とする道具の深いところまでを理解して、自分の意のままに操れるようになることを指すだろう。実際のところ、現在のパソコンで文書作成の作業などを気軽に行なえるのは、簡単に操作できるように作られているからであり、誰でも操作が分かるのは、あえていうなら「人間がそのように操作するように仕向けられている」からである。言い換えれば、人間がパソコンに操られている部分も少なくないのである。これはやはり「能動的」に使いこなしているとは言えないと感ずる人もいるだろう。

能動的に使いこなすためには、対象物の仕組みを深く理解することが重要である。そのためにもっとも有効な方法が「プログラミング」と言える。プログラミングというとコンピュータに向かってする作業のなかで、もっとも高度な技術を要する理科系的なことのように感ずる人が多いがそれは少し違う。プログラミング言語は、メッセージを伝える相手がコンピュータという点が異なるものの、単なる「ことば」であることに変わりはない。外国語の習得には文法を体系立てて理解することも重要だが、それ以上に実際に会話や読み書きを繰り返すことがすばやい上達につながるように、プログラミング言語も実際に読み書きし、コンピュータと「対話」を繰り返すことが上達の早道となる。また、外国語を学ぶ過程で、その言葉を話す国の文化などの背景知識を得ることが本質的な理解につながるのと同様、プログラミング言語を覚えることはコンピュータの原理に一歩踏み込んだ理解を促すことにもつながる。「コンピュータの都合」をある程度理解することで、普段使っているソフトウェアが出す、意味不明のエラーメッセージが、実は必然的なものだったと気付き、正しい対処法が見えてくるだろう。

このようにプログラミング言語を習得することは、コンピュータに対する総合的な知識の向上の早道である。しかしそれ以上に重要なのは

プログラミングそのものが楽しい

ということである。何回かの試行錯誤を経てプログラムが完全に動くようになったときの快感は何とも言えない。本書がプログラミングという楽しい自己実現方法を楽しむ第一歩の助けとなれば幸いである。

2026年8月 広瀬雄二

本書の読み進め方

◆対象読者

本書は主に過去にプログラミングの経験のない人を対象としている。そのため、「Cでいうところの」とか「Perlでいうところの」といった他言語からの類推を用いた説明は全くしていない。そのため、未経験の人には親切だが、プログラミング経験者にはまどろっこしく感じる部分があるかもしれない。

本書の説明の流れは、実際に初習の学生に教えたときの経験を元に、初学者が誤解しやすい考え方を先回りして正しい方向に導くようにしている。たとえばプログラミングに習熟したのちは、「変数はローマ字より意味のある英単語に」であるとか、「作業変数などはそれとわかる名前にする」だとかの習慣がつくものである。しかし、初学者の挙動を見ていて、言語の全体像が見えないうちはたとえ書体等で意味付けをしていても、「必ずその変数名で書かなければいけない」などといった誤解をしやすいことが分かった。そこで本書の前半部分では、「ほげほげ」などあえて変な単語を用いることで明らかにそこは自分の裁量で決めてよい部分と分かるようにした。

また、本書の説明の順序は、プログラミングの概念が見えてくる前でも分かりやすいものは最初にある程度丁寧に、最初のうち理解しにくい項目は先に簡単な説明をしてからあとで深いところまでもう一度取り上げる、という観点で決めている。決して文法的都合で順番が決まっているわけではない。取り上げる題材も「これをちゃんと知らないと先に進めない」という部分以外は極力説明を省いた。それゆえメソッド類の説明も必要最低限のものに絞っているので、Ruby 言語そのものをしっかり覚えたい場合は、網羅的に書かれた本を次の教科書として傍らに置く必要があるだろう。Ruby独自の機能を使うとプログラムが何分の一にも簡潔に書けることがあるので、将来的にもRubyを主に利用したい場合はRuby 専修書に進んで学習していくとよいだろう。本書では、Ruby以外の言語に進む場合も考慮して、Ruby固有のすぐれた機能は利用を見送ったものが多くなっている。

本書の主眼は、プログラミングを通じてコンピュータの仕組みの基礎を知ること、そしてプログラミングのもつ楽しさを体感することにある。「プログラミングは一部の人ががやること」、という先入観を捨てて、存分に楽しんで欲しい。

◆学習の進め方

本書は、実際に初学者対象に講義を行なう進度を元に章割が決められている。講義一回分ごとにまとめられているので、自習するときの学習量の目安にして欲しい。

本文中には例題となるプログラムが随所にちりばめられている。必ず手元でも同じプログラムを入力して実際に動かしてみて欲しい。プログラミング能力向上の早道は「習うより慣れろ」で、自分の体験に優る先生はない。

もし読者がプログラミングを教える立場にあるのなら、本書で説明が冗長になっているところを参考にすると初心者がつまづきやすい箇所がわかるだろう。

◆想定するプログラミング環境

本書では X Window System でテキストエディタとしてEmacs、仮想ターミナルソフトとして KTerm、シェルとしてzsh、かな漢字変換システムとしてCannaを利用することを想定して説明を進める。ただし、既になじみのテキストエディタがあれば、それを利用して全く構わない。プログラムやデータファイルに日本語を含めるときは日本語EUCコードを利用するものとする。また、文中でKTermとzshでの例示がしてあるが、それらでなければ実現できない機能は利用していないので別のソフトウェアを使うならそれらに置き換えて読んで欲しい。

初学者なのにUnixを前提とすることを意外に思うかもしれないが、近年の Windowsの変遷は、「初心者が自分でプログラミング環境を構築すること」を飛躍的に困難にしている。「環境変数を設定」という部分だけでも前提条件が多様で初学者の高い障壁になっている現実を見ると、普段Unix環境を利用していない場合でも、Rubyだけのためと割り切ってでも Unix環境を導入する方が楽であるように思える。

これからRubyの学習ができる環境を用意するのであれば、以下のいずれかの方法をお勧めする。

1. PC-Unix環境を用意する

   現在では初期インストールでGUI環境まで簡単に揃えることができるので、EmacsとKTermが使えるようにするのも簡単だろう。

2. Mac OS Xを利用する

   Mac OS Xでは Xcode というプログラミング環境のパッケージをインストールするとRubyが使えるようになる。Emacsもターミナルソフトも標準装備されているので、導入作業に全く苦労しない。

3. WindowsでCygwinを導入する

   CygwinとはUnixむけツールをWindows環境に移植したものである。 Cygwin環境で動くRubyがあるのでこれを利用するとよい。

上記に限らず、自力で用意できるのであればどんな組み合わせでも構わない。

◆本書で用いる記法

本書では、書体やレイアウトによって以下のような意味付けをしている。

- コンピュータへの入力

  "gets" のようなタイプライタ書体で書いた英字は、Rubyプログラムとして意味を持つ単語や値、またはそのようなコマンド名であることを意味する。その他コンピュータに対する入力全般を示す。

- 省略可能部分

  文法的な説明の部分で、

  ```
  if a1 [then]
  ```

  のように [] で括った部分はそれが省略可能であることを示す。上記の例では then という単語をそこに書いても書かなくてもよいことを意味する。

- 変動値

  文字列 あるいは、*String*のように書いた語は、その場に実際に「文字列」や「String」と書くのではなく、その語で代表される色々な値でそこが置き換えられることを意味する。たとえば「*文字列*.chomp」と書いてある部分は、「*文字列*」の部分が、"foo" や "あいうえお"あるいは、その他の任意の文字列に相当することを意味する。

- キー入力

  本文中に現れる [Return] という部分は、タイプすべきキー操作であることを意味する。Ctrlキーなどの組み合わせについては、以下のように表記する。

C-*x*	Ctrlキーを押しながら*x*を押す
M-*x*	Metaキーを押しながら*x*を押すか、 ESCキーを押して手を離してから*x*を押す
[Return]	Returnキー(キーボードによってはEnterキー)を押す
< kbd>[Tab]	Tabキーを押す

- 実行例と入力部分

  段落中に現れる

```
% ./hello.rb
Hello, world!
```

  は、対話的な画面での入出力結果を意味する。たとえばコマンド入力行で実行したりエディタに入力する例に相当する。例のうち下線のついている部分が人間が入力した部分を意味する。

- リスト

  段落中に現れる

```
print "ほげほげ"
```

  のような部分は、Rubyプログラムのリスト(またはその一部)であることを意味する。

- 値

  リスト中に

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

  のように=>記号に続けて示した部分は直前の変数や式が持っている「値」を意味する。上の例の場合は x という変数の持つ値が[3, 2, 1]であり、 x.reverseという文自体が[1, 2, 3]という値を持つ、ということを意味する。それらの値が画面に出たりするわけではない。

また、本文中に現れるバックスラッシュ記号 \ は、日本語キーボードで円マーク ￥ と刻印されているキーを押すと出てくる記号である。コンピュータにとってはどちらの記号も同じだが、プログラミングで使う意味としてはバックスラッシュが適切なのでそちらで表すことにする。

第1講 はじめに

◆プログラミングとは

◇プログラミングで得られるもの

コンピュータ(以下計算機)に対する直接的な指示を書いて実行させる。それがプログラミングである。そのために必要な能力は何だろう。多くの人は数学的・工学的知識を真っ先に思い浮かべ、プログラミングは理科系の専売特許のように考えている。そうした知識も重要だが、実際には

- 論理構成力
- 状況判断力
- 予測・推理力
- 創造力

なども必要で、プログラミングを習得する過程でそれらが総合的に養われていく。また、プログラミングを「機械を相手にしている」と決めてかかる風潮もあるが、実際には違う。作ったプログラムを使うのは人間であり、プログラムを作るときには、つねに使う人間のことを想定する必要がある。人間には「思い込み」、「うっかりミス」、「いい加減さ」など不確定要素による行動が多いため、プログラムを作るときには人間がどのように行動するかを全て先回りして考える必要がある。良いプログラムを作るには「思いやり」が不可欠である。

それらを意識しながらプログラミングを繰り返すことで、論理的思考力が高まったり、実生活で人間と直接関るときにもそれまでは予測し得なかった「人間くさい」反応に対応できる能力がついていく。

◇プログラミングの楽しさ

もうひとつ、プログラミングには「楽しい」という重要な側面がある。自分で作ったものが完成したとき、それがうまく動いたとき、使えたとき、きれいにできたとき、料理ならおいしかったとき、人間は嬉しいと感じるとともに、大きな達成感を得る。ものを作るときには、乗り越えられる程度の「困難さ」があると達成感は倍増し、うまくできたときに脳が快感を得る。この快感こそが人間の能力を高める重要な要素となっている。

プログラミングには、初歩の段階で得た知識だけでもほぼ期待どおりのものが作れるという性格がある。それゆえ最初はなかなかうまくいかないと苦労していても必ず完成させることができ、それにより大きな達成感が得られる。また、何度失敗しても計算機は腹を立てたり壊れたりしないので、気の済むまで試行錯誤を繰り返すことができる。さらに進んで、ある程度の水準のプログラムを書けるようになった後には、「誰かに使ってもらう」という喜びも味わうことができる。初歩から上級者まで、全ての段階で楽しめるのがプログラミングの特徴である。

◇プログラミングを学ぼう

高度に発達した工業製品は、我々から「ものの仕組み」を考える余地を奪っている。電子制御によりボタンひとつで全ての操作が終わってしまう電化製品は快適さをもたらす代わりに、工夫、計画性、探求心、創造力、好奇心といった人間の生きる力を育てる機会を減らしている。

そんな中、電子制御の最先端を行く計算機には、みずからの頭で考えて道具を操り、ものが動く仕組みを存分に考えられる余地が残されている。それがプログラミングである。幸いにして現在では最高のプログラミング環境が無料で利用できるのが当たり前になっている。プログラミングを通じて問題解決能力を楽しく磨いていこう。

◆プログラミングに必要なもの

プログラミング言語を覚えただけではプログラムを作ることができない。ひとつのプログラムを作るには、それを書くための道具、機械の言葉に翻訳するための道具、プログラムを動かすための道具、など色々な道具が必要である。

以下で説明するテキストエディタやシェルについて既に利用経験がある場合は、読み飛ばして第2講に進んでよい。

◇コマンドラインとシェル

Terminal(ターミナル/仮想端末)の画面には起動直後以下のような文字列が出ている。

```
yuuji@fiberstorm%
```

[Return]キー(キーボードによっては[Enter])を何度か押すと、同じ文字列が何度も出てくる。このような文字列のことを*プロンプト*といい、「システムに対してコマンドを打ってもよい」状態であることを意味している。プロンプトを出しているのは*シェル*と呼ばれるソフトウェアで、シェルはユーザが打ち込んだ文字列をコマンドとして解釈し、システムにある同名のコマンドを起動したり、それが見つからない場合はエラーメッセージを出したりする役割を持つ。我々がこれから作るRubyプログラムも、起動するときはシェルのお世話になる。シェルに対してコマンドを打ち込む場所、つまりプロンプトの後ろの行のことを *コマンドライン*(コマンド入力行)という。

```
yuuji@fiberstorm%
~~~~~~~~~~~~~~~~~ ~~~~~~~~~~~~~~~~
    ↑                   ↑
  プロンプト        コマンドライン(これから何かコマンドを打つべき場所)
```

本講ではプロンプトを%記号で示す。

◇ディレクトリとファイル

Unixシステムのアカウント(利用権)があるユーザには必ず、自分専用のファイルを保存してよい場所が割り当てられている。この場所のことを *ホームディレクトリ*という。ホームディレクトリには普通のファイルを保存してもよいし、さらに別のディレクトリを作ってファイルを分類して格納してもよい。ホームディレクトリの中ではいくらでもファイルやディレクトリを作ったり消したりできるが、ホームディレクトリの外(上位)のディレクトリには通常書き込みできない。

コマンドラインでpwdコマンドを実行してみよう。コマンド実行の最後には[Return]キーを押す。

```
% pwd
```

```
/home/yuuji
```

返ってくる結果は人によって違う。表示されたのは現在操作対象となっているディレクトリで、ログインして最初に見せられるディレクトリはホームディレクトリである。ホームディレクトリは自分の拠点となるディレクトリで頻繁に参照することからチルダ記号(~)一文字で表してもよい約束になっている。たとえば、~/foo.rb とは、「ホームディレクトリにあるfoo.rbというファイル」を意味する。

また、特別なディレクトリ指定方法として以下の二つも覚えておく必要がある。

.	カレントディレクトリ
..	ひとつ上のディレクトリ

*カレントディレクトリ*とは、現在操作対象となっているディレクトリのことで、作業ディレクトリともいう。カレントディレクトリは起動しているソフトウェアごとに独立して持っている。Terminalを2つ起動している場合は2つのシェルで別々にカレントディレクトリを管理している。

◇ファイル・ディレクトリ操作コマンド

ファイルやディレクトリを操作するコマンドとして以下のものを覚えておこう。

- *ls [オプション] [ファイルorディレクトリ]* - ファイルの一覧表示

  指定した *ファイルorディレクトリ* にあるファイルを一覧表示する。ls とだけ起動するとカレントディレクトリにあるファイル一覧が出る。*オプション* のうちよく使うものを列挙する。

-l	ロングフォーマット(詳細表記)で表示する。
-F	ファイルの種別が分かりやすいようにファイル名の後ろに記号を付けて表示する。ファイル名だけが出るときは普通のファイル、それ以外のときは以下のような意味となっている。 / ディレクトリ * 実行可能なファイル @ シンボリックリンク
-a	ドットで始まるファイル(普段は見えない)も表示する。
-t	更新時刻の新しい順に表示する(通常はアルファベット順)。
-r	並べ換え順を逆にして表示する。
-R	指定したディレクトリ以下全てのファイルを表示する。

  例:

  ```
  % ls -F
  Mail/        help         memo         script/
  TUTORIAL.ja  maildir/     public_html/
  (末尾に / のついているものはディレクトリ)
  % ls -F Mail
  draft/ inbox/ trash/
  % ls -lF Mail
  total 3
  drwx------  3 c101345  st2001   512 Oct 27 01:13 draft/
  drwx------  2 c101345  st2001  2048 Oct 27 01:14 inbox/
  drwx------  2 c101345  st2001  1024 Sep 11 13:58 trash/
  ```

- *pwd* - カレントディレクトリの表示
- *cd [ディレクトリ]* - カレントディレクトリの移動

  指定した *ディレクトリ* にカレントディレクトリを変更する。*ディレクトリ* を省略すると、ホームディレクトリに移動する。

  例:

```
% pwd
/home/yuuji
% cd Mail
% pwd
/home/yuuji/Mail
% ls -F
draft/    inbox/    trash/
% cd inbox
% pwd
/home/yuuji/Mail/inbox
% cd ../draft
% pwd
/home/yuuji/Mail/draft
% cd
% pwd
/home/yuuji
```

- *mkdir 新ディレクトリ* - ディレクトリの作成

  新ディレクトリ を作成する。複数同時に指定してよい。

- *rm [オプション] ファイル* - ファイルの削除

  指定した *ファイル* を削除する。複数指定可。よく使うオプションは以下のとおり。

-r	指定した*ファイル*がディレクトリでもその中を含めてまるごと削除する。ディレクトリを削除する目的で使う。
-f	ファイルが書き込み禁止でも強制的に削除する。

- *mv ファイル1 ファイル2* - ファイル名の変更

  ファイル1 を *ファイル2* という名前に変更する。

- *mv ファイル群 ディレクトリ* - ファイル群の移動

  ファイル群 のファイル(複数指定可)を最後に指定した*ディレクトリ*に移動する。

- *cp ファイル1 ファイル2* - ファイルのコピー

  ファイル1 を *ファイル2* というファイルにコピーする。

- *cp ファイル群 ディレクトリ* - ファイル群のコピー

  ファイル群 のファイル(複数指定可)を最後に指定した*ディレクトリ*にコピーする。

- *cat ファイル群* - ファイルの内容出力

  指定したファイル(複数指定可)の内容を出力する。エディタで作ったファイルの中味をコマンドラインで確認したり、ファイルの中味を別のコマンドに渡したりするときに利用する。

- *less ファイル群* - ファイルの内容閲覧

  指定したファイル(複数指定可)の中味を対話的に閲覧する。起動するとless専用コマンドキーを受け付ける。以下のキー操作を覚えておけば十分だろう。

q	lessを終了してシェルに戻る
SPC	一画面分先に進む
b	1画面分上に戻る
j	1行分先に進む
k	1行分上に戻る

  1画面より長いファイルの中味を目で見たいときにはlessを利用する。

◇Emacsの操作

通常プログラムは*テキストエディタ*というソフトウェアを使って作成し、ファイルとして保存する。本書ではEmacsを利用するものとして必要最低限の利用方法について説明する。最初に全て覚える必要はないが、定期的にここを読み返して最終的に全て覚えられることを目標にするとよい。テキストエディタを使いこなせるかどうかはプログラミング能力の向上速度に大きくかかわる。

■用語について

以後の説明に出てくるEmacsに関する重要な用語を先に説明しておく。

バッファ
: Emacsで編集するために見えている画面(の中味)のこと。普通はファイルの中味そのもの(のコピー)が見える。Emacs起動直後に見えているバッファは scratch buffer といい、どのファイルにも結び付けられていない。

ウィンドウ
: 見えているバッファ全体を囲う範囲のこと。Emacs全体の画面を複数のウィンドウに分割して使うことができる。

ポイント
: 文字を打つと文字の入る場所を*ポイント*という。通常はカーソルの左端である。

■操作の概要

Emacsではキー操作だけで全ての機能を呼び出せるようになっている。それらキー操作はControlキーを押しながら別のキーを押したり、 ESCキーを押してから別のキーを押すようになっている。たとえば、「ファイルの保存」は C-x C-s である。 Controlキーを押しながらxと、Controlキーを押しながらs、を連打して初めて機能の呼び出しとして完結する。覚えるまでしばらく戸惑うかもしれないがプログラミングでは何度も繰り返しプログラムを書き換えるので、基本的なエディタ操作は一瞬で行なえるようになっておかないと時間を浪費するばかりである。

■コマンドの中止

前述したとおりEmacsの機能呼び出しは複数のキーの組み合わせで行なうことが多い。慣れないうちは打ち間違えたり、途中まで打って分からなくなることがあるが、そのような場合は

C-g　　コマンド入力中止

をタイプすればいつでも初期状態に戻る。

■操作の取り消し

間違って文字を消しすぎたりなど、希望と反する修正操作をしたら、

C-/　　編集操作の取り消し(undo)

をタイプすると直前の状態に戻せる。何度も連続してC-/を押すことでファイルを開いたときまでさかのぼってundo(アンドゥ)できる。 undoしすぎた場合はカーソル移動等undo以外の機能を使ったあとで C-/をタイプすれば、undoをundoできる。

■ファイルの操作

下記一覧のうち C-x C-f と C-x C-s は今すぐ確実に覚えておこう。

C-x C-f	ファイルを開く(find-file)
C-x C-s	編集中のファイルを保存する(save-buffer)
C-x C-w	編集中のファイルを別の名前で保存する(write-file)
C-x i	外部ファイルをポイント位置に挿入する
C-x k	バッファの破棄(閉じること)
C-x b	バッファの切り替え
C-x C-m f	ファイル保存文字コードの変更

■ポイント移動

ポイント(カーソル)移動キーとして、最初はC-n、C-p、C-b、C-f だけ覚えておけば十分だろう。矢印キーでもポイント移動できるが、ホームポジションから手を離さねばならず速度的に不利なので C-n、C-p、C-b、C-f に慣れる方が望ましい。

				M-< バッファ先頭へ				
				M-v 1画面戻る				
				C-p 1行戻る				
				↑				
C-a	M-b	C-b				C-f	M-f	C-e
行頭	1語戻る	1字戻る	←	・	→	1字進む	1語進む	行末
				↓				
				C-n 1行進む				
				C-v 1画面進む				
				M-> バッファ末尾へ				

■領域操作

編集バッファの「ここから、ここまで」をまるごとコピーしたり削除したりするには

1. 「ここから」に相当する場所をマークする
2. 「ここまで」に相当する場所に移動して該当する操作をする

という手順で行なう。

C-SPC	ポイント位置を「マーク」する
C-w	マークからポイントまでの領域を削除して記憶する
M-w	マークからポイントまでの領域を記憶する
C-y	C-wやM-wで記憶したものをヤンクする(ペーストする)。

練習として、編集バッファに以下のようなテキストを打ち込んでみよう。

```
Rubyをすこしでも早く覚えたい
```

「すこしでも」を領域削除してみよう。「ここから」にあたる「す」の位置にポイント移動する。

```
Rubyをすこしでも早く覚えたい
```

C-SPCでこの位置をマークする。

「ここまで」にあたるのは「も」の後ろなのでそこにポイント移動する。

```
Rubyをすこしでも早く覚えたい
```

C-w(kill-region)で領域が削除される。

```
Rubyを早く覚えたい
```

■ウィンドウ分割操作

Emacsはウィンドウを分割して一度に複数のバッファを見ながら編集できる。ウィンドウ分割のための主なコマンドは以下のとおり。

C-x 2	ウィンドウを上下2分割
C-x o	(小文字のオー)隣のウィンドウに移動
C-x 1	ウィンドウを1つにする
C-x 0	(ゼロ)今のウィンドウを消す

■編集操作の練習

少し練習してみよう。実際に

> ~/TEST/hoge と ~/TEST/hero ファイルを作成・編集し、最後にそれらを削除

という操作をしてみよう。具体的には、

1. (Terminalで) ~/TEST ディレクトリを作成し
2. (Emacsで) ~/TEST/hoge ファイルを作成し
3. (Emacsで) ~/TEST/hero ファイルを作成し
4. (Emacsで) hogeとheroバッファを行ったり来たりして
5. (Emacsで) hogeとheroバッファを閉じ
6. (Terminalで) ~/TEST ディレクトリをまるごと削除

のような流れで作業する。。

1. ~/TEST ディレクトリの作成

   Terminalでmkdirする。

```
% mkdir TEST
% ls -F
Mail/           TUTORIAL.ja     maildir/        public_html/
TEST/           help            memo            script/
% cd TEST
% ls
(空っぽなので何も表示されない)
```

2. ~/TEST/hoge ファイルの作成

   Emacsで C-x C-f すると編集したいファイル名を聞いてくるので ~/TEST/hoge を入力する。

```
Find file: ~/TEST/hoge
```

   TESTを入力するときに、最初の1、2字入れたところで [Tab]キーを押すと残りが補完される。ファイルを開いたら以下の内容で保存(C-x C-s)する。

```
ほげほげほげー
```

```
やあやあやあー
```

保存できたらTerminalに移り、実際に書き込まれているか確認する。

```
% cat hoge
ほげほげほげー
やあやあやあー
```

3. ~/TEST/hero ファイルの作成

   Emacsに戻り~/TEST/heroファイルを開く。 C-x C-fしてファイル名を入力する。

   ```
   Find file: ~/TEST/hero
   ```

   新規に開いたら以下の内容で保存する。

   ```
   こっちはへろへろです。
   ```

   こちらもTerminalで確認してみよう。

   ```
   % cat hero
   こっちはへろへろです。
   ```

4. hogeとheroバッファの往復

   EmacsでC-x bをタイプすると、切り替えたいバッファ名(ファイル名と同じ)を聞いてくるので hogeまたはheroを入力し相互を往復する。何度か繰り返そう。

5. hogeとheroバッファの破棄

   EmacsでC-x kとタイプすると、閉じたいバッファ名を聞いてくるので hoge または hero を入力してそれを閉じる。未保存のバッファはうまく閉じられないので必ず保存してから C-x kする。バッファ名を入れずにリターンを押すと現在編集中のバッファが閉じられる。

6. ~/TEST ディレクトリの全削除

   Terminalでホームディレクトリに戻る。

   ```
   % cd
   % ls -F
   Mail/           TUTORIAL.ja     maildir/        public_html/
   TEST/           help            memo            script/
   ```

   rmコマンドで、TESTディレクトリを中にあるファイルごと一気に消してみる。

   ```
   % rm -rf TEST
   % ls -F
   Mail/           help            memo            script/
   TUTORIAL.ja     maildir/        public_html/
   ```

以上の操作を3、4度繰り返してみよう。とくに「ファイルの保存」と EmacsとTerminalの往復をスムーズに行なえるようにするとのちのちの作業効率が格段に上がるので練習しておこう。

◇その他の操作

作成したプログラムを実行するときに知っておくと役に立つ操作方法をいくつかまとめておく。

■実行プログラムの強制終了

自分で作ったプログラムを実行中に途中で止めるにはC-cをタイプする。

■実行画面のスクロールバック

実行したプログラムによる出力がスクロールして画面の上の方に消えてしまった場合は、Terminalのスク

ロールバーを利用すればさかのぼって見ることができる。

■Terminalでのコピーペースト

マウスの左ボタンでドラッグすると範囲選択した部分がコピーされる。 Emacsにペーストするときはヤンク(C-y)する。

Terminalでの領域選択のとき

- 範囲指定開始の左ボタンクリックを
  - シングルクリックにすると文字単位での範囲選択
  - ダブルクリックにすると単語単位での範囲選択
  - トリプルクリックにすると行単位での範囲選択

  になる。
- 左ボタンを押してすぐ手を離してもよく、その場合は右ボタンクリックで範囲の終わりを指定する

記憶した領域をTerminalにペーストするにはマウスの真中ボタンをクリックする。

■日本語入力

Terminalでの日本語入力モードは Shift-SPC をタイプして切り替える。Emacsでは C-o で切り替える。

第2講 プログラミングの基礎

◆プログラミング言語と計算機

◇プログラミング言語

計算機は人間が話す言葉を直接理解できるわけではなく、実際には非常に細かい命令、たとえば「今ある数字に1を足せ」のような命令を数字に置き換えたものを解釈実行する。
機械語の例:

```
0000000 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
0000020 01 00 03 00 01 00 00 00 00 00 00 00 00 00 00 00
0000040 88 00 00 00 00 00 00 00 34 00 00 00 00 00 28 00
0000060 08 00 05 00 55 89 e5 b8 99 99 00 00 40 c9 c3 00
  ⋮
  ⋮
```

右下付近にある40というのが「1足せ」という意味をもつ機械語だが、周囲も含めてちっとも想像がつかない。プログラムを作る場合このような数字の羅列は人間には分かりづらいので、記述する場合に非常に手間がかかる。そこで、ある程度まとまった処理、たとえば「画面に "Hello" というメッセージを表示せよ」程度の処理を、人間が読んで分かる表現形式で書くために作られたものが*プログラミング言語*である。

たとえば上記の処理は、各種プログラミング言語では以下のように表現する。

```
Fortran
   WRITE(*,*) "Hello"
Basic
   print "Hello"
Perl
   print "Hello";
Ruby
   print "Hello"
```

機械語と違い、単語を見ればなんとなく指令している内容が分かるだろう。

◇言語処理系

プログラミング言語の文法に則って書かれたプログラムを *ソースプログラム*という。これを実際に計算機に理解できる数値の羅列(機械語)に変換するソフトウェアを *コンパイラ*(Compiler; 翻訳器)という。コンパイラはソースプログラムを機械語に一括変換するので、プログラムを作るときは最初に文法的に完全に正しいものを書く必要がある。

いっぽう、ソースプログラムを読みながら逐一それに必要な処理を呼び出すことでプログラムの実行を行なうソフトウェアがあり、これを*インタプリタ*（Interpreter; 通訳器)という。これから習う Ruby 言語は、インタプリタ型言語であり、ruby というコマンドが、呼び出されたときに逐一プログラムを解析しながら書かれたとおりのことを実行する。

コンパイラ、インタプリタの種別にかかわらず、あるプログラミング言語で書かれたソースプログラムを実際に実行させるところまでの処理を行なうシステムを*言語処理系*という。これから利用する Ruby 言語を動かすシステムも言語処理系の一つである。

◇コンパイラとインタプリタの特徴

コンパイラを用いてあらかじめ全て機械語に直してしまったものを実行する場合は、機械の持つ本来の速度で実行することができるが、インタプリタはそれよりも遅い。インタプリタでは、プログラムを実行時に

1. ソースプログラムを読み込み
2. 意味のあるかたまりごとに分解し
3. それぞれに文法エラーがないか調べあればそれを警告し
4. エラーがなければあらかじめ作っておいた同等の処理を呼び出す

などの処理を同時に行わなければならないので動作速度が非常に遅くなる。反面、インタプリタはソースプログラムのどこかに書き間違いがあっても、とりあえず正しく動く部分だけで実行することができるし、プログラムに間違いがあった場合はその場で動作を止めて該当箇所を画面に表示してくれる。したがってプログラミングに慣れていない人が、間違いを修正しながらプログラムを作成していくのに向いている。

これとは逆に、コンパイラ型言語はあらかじめ全てを機械語に直してから実行するので、計算機の持つ本来の能力でプログラムを動かすことができる。ただし、最初にプログラムの全ての部分の翻訳を完了させなければならないので、プログラムのほんの一部分に間違いがあるだけでもいっさい実行させることができない。

以上のような特徴があるが、現在の計算機の速度はインタプリタの遅さを実感させない程度に向上している。インタプリタ型言語は、初心者の練習だけでなく本格的な用途にまで幅広く対応できる。

◆プログラミングの流れ

Rubyによるプログラミングは次のような流れで行なう。

1. エディタ(Emacs)でプログラムを作成
2. 保存
3. rubyインタプリタの呼出し(実行)
4. 間違いがあれば修正して2に戻る

◇プログラムの作成

簡単なプログラムを作成し、動かし、プログラミングの流れを把握しよう。

まずプログラムを作る前に、*Rubyプログラムをしまうディレクトリ* を決めておこう。そのままホームディレクトリに作ってもよいが、作成プログラムが増えるにつれファイルがあふれて収拾が付かなくなる。

仮に、Ruby関連のファイルをしまうディレクトリを Ruby に決めたとしよう。その場合、

```
% mkdir Ruby
```

として、ディレクトリを作成する。Emacsでこのディレクトリの中にファイルを作るときは C-x C-f (find-

file) で

```
Find file: ~/Ruby/ファイル名
```

と指定する。 ~/Ruby/ で、先程作成(mkdir)した、Ruby用のディレクトリの中にファイルを作ることを意味する。

それでは、~/Ruby/program.rb というファイルを作って動かしてみよう。まずファイルを開くC-x C-f

```
Find file: ~/Ruby/program.rb
```

プログラムとして以下の内容を作成する。こまめに保存（C-x C-s)することを忘れない。

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
print "Hello\n"
```

◇Rubyプログラムの実行

上記の操作で ~/Ruby/program.rb が完成する。

プログラムを実行するときは Terminal の画面で最初に

```
% cd Ruby
```

としてディレクトリを移動しておく。

作成したRubyプログラムを実行する方法は二つある。比較的簡単だが毎回同じ手間がかかる方法(1)と、最初に手間がかかるが二回目以降楽になる方法(2) である。

1. Rubyインタプリタを指定する方法

   ```
   % ruby program.rb
   ```

2. プログラム中に Ruby インタプリタを埋め込む方法

   プログラムの1,2行目を

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

   にし、最初に一度

   ```
   % chmod +x program.rb
   ```

   すると、以後、

   ```
   % ./program.rb
   ```

   で直接実行できる。

> 2.の方法の1行目の3文字目以降の /usr/local/bin/ruby は、読者のシステムによって異なる可能性がある。
>
> ```
> % which ruby
> ```
>
> と起動した結果で置き換えて欲しい。

本講の説明では、2.の方法を前提とする。上記の実行例で示した chmod の実行は以後示さないが、プログラムを最初に保存した直後には各自chmod +xによりプログラムを実行可能な状態に変更することを忘れずに行なうこと。

◆問題を解くプログラム

つぎのような問題を考えよう。

- 人口10万人の都市がある。1年に10%ずつ増加するとした場合、人口が20万人になるのは何年後か。

これを解くプログラムをRubyで作ると以下のようになる。

population.rb

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

p=p0=100000
y=0
while p < 2*p0
  p *= 1.1
  printf("%3d年\t%d人\n", y+=1, p)
end
```

このプログラムを population.rb という名前で作成し、実際に実行してみよ。

◇手順

1. ファイルを開く

   C-x C-f (find-file)して population.rb を開く。ファイルを作成するディレクトリに注意する。

2. 保存する

   C-x C-s

   こまめに保存すること!

3. 実行する

   Terminalに移り、(必要ならディレクトリもcdで移り)

```
% chmod +x population.rb
% ./population.rb
```

とする。chmodは、一つのプログラムに対して最初の一回だけやれば良い。

実行結果:

```
  1年    110000人
  2年    121000人
  3年    133100人
  4年    146410人
  5年    161051人
  6年    177156人
  7年    194871人
  8年    214358人
```

実行結果とプログラムを見比べ、プログラムの各部分がどんな意味を持つのかを自分なりに想像してみよ。

◇インデント

上記のプログラムでは、「while...」の行と、その次の「p *= 1.1」では書き始めの位置が違っている。

行の最初の文字を書く桁位置を変えることをインデント(字下げ)といい、プログラミングや構造的文書を書く際には非常に重要な意味を持つ。 EmacsでRubyプログラムを作成する場合、1行入力し終えたときに [Tab] キーを押すと自動的に適切なインデントに揃えてくれる。

今後、プログラムを作成するときには、各行の入力や修正が完了したら必ず[TAB]キーでインデントを揃えよう。そうすることでプログラムの入力間違いが軽減できる。

◆間違いの修正

プログラムが間違いなく一発で動くことはまれである。必ずどこか間違えるものなのでそれを修正しなければならない。ここでは一つの間違いプログラムを修正していく手順を詳しく示す。この節は最初は読み飛ばし、練習問題を解くときに実際にたくさんの間違いに遭遇してから戻って読んでも構わない。

ためしに以下のプログラムを~/Ruby/mistake.rbという名前で保存しよう。「間違い」を修正するためにわざと間違えているプログラムなので一字一句正確に入力しよう。念のため、誤っている部分には 下線が施してある。

mistake.rb

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

put "整数を入れてください"
n=gets.to_i
if m%2 == 0 then
  print "偶数です\n
else
  print "奇数です\n"
```

保存したらTerminalで実行属性を付けておく。

```
% chmod +x mistake.rb
```

典型的な間違いの例とその修正方法を示していく。

1. 「インタプリタ指定の間違い」

   まずは実行してみよう。

   ```
   % ./mistake.rb
   zsh: ./mistake.rb: bad interpreter: /usr/local/bin/luby: no such file or directory
   % ls -lF mistake.rb
   -rwxr-xr-x  1 yuuji  wwwadmin  123 Jul 27 22:23 mistake.rb*
   ```

   ファイルが実際にあるのに「./mistake.rb なんてファイルはない」というエラーが出ている。プログラム1行目のインタプリタの指定が間違っている。 1行目を以下のように修正して保存しよう。

   ```
   #!/usr/local/bin/luby
   ```

   ↓

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

2. 「引用符の閉じ忘れ」

   1行目の修正と保存が終わったら再度実行してみよう。直したのにエラーが増える。

   ```
   % ./mistake.rb
   ./mistake.rb:8: syntax error, unexpected local variable or method, expecting `end'
     print "奇数です\n"
   ./mistake.rb:8: unterminated string meets end of file
   ```

```
  print "奇数です\n"
./mistake.rb:8: syntax error, unexpected end-of-input, expecting `end'
  print "奇数です\n"
```

たくさんのエラーが出て、なおかつ最後の方に

```
    unterminated string
```

というメッセージがある場合は "" または '' を閉じていない場合である。このようなときは、たいてい最初に出てくるエラーメッセージに着目すればよい。今回は

```
mistake.rb:8: unterminated string meets end of file
```

とあるので、引用符の閉じ忘れを探す。今回の場合は 8行目を修正する。

```
print "偶数です\n
```

↓

```
print "偶数です\n"
```

エラーメッセージのファイル名の後ろに「:8:」とあるのは「プログラムの8行目に間違いがあるよ」という意味である。 Rubyインタプリタは必ず間違いのある場所を行番号表示するので必ずこれを確認する。Emacsで指定した行番号の位置に移動するには編集バッファの下の方に

```
-かんな-EJE:---F1  mistake.rb     All L3   ...
```

のような反転表示行の「L3」のような数字を参考にする。あるいは、

```
M-x goto-line [Return]
8 [Return]
```

とすると8行目に直接移動できる。

3. 「endの閉じ忘れ」

修正して保存したらまたプログラムを実行してみよう。

```
% ./mistake.rb
mistake.rb:8: syntax error, unexpected end-of-input, expecting `end'
  print "奇数です\n"
```

「expecting `end'」と表示された。 expecting は「期待している」ということで、end があるはずなのに、ないと言われている。指定通り、プログラムの末尾に

```
end
```

と書いた行を追加して保存しよう。

4. 「単語の打ち間違い」

修正したらまた実行しよう。

```
% ./mistake.rb
./mistake.rb:3:in `<main>': undefined method `put' for main:Object (NoMethodError)
Did you mean?  puts
               putc
```

「undefined」とは「未定義の」という意味だが、Rubyプログラムの場合綴り違いをおかしている時に発生するエラーである。エラーメッセージのとおり3行目に間違いがある。"Did you mean?" はもしかして……では? ということで、その次にある puts か putc の書き間違いではないかと指摘してくれている。この場合、 putではなくputsが正しい。

```
put "整数を入れてください"
```

↓

```
puts "整数を入れてください"
```

修正後もう一度実行するとまたエラーが起こる。

```
% ./mistake.rb
整数を入れてください
8
./mistake.rb:5:in `<main>': undefined local variable or method `m' for main:Object (NameError)

if m%2 == 0 then
   ^
```

これも「undefined」なので綴り間違いである。5行目の mの下に ^ 記号は間違っている場所の指摘である。上で利用している変数nに直す。

```
 if m%2 == 0 then
```

↓

```
 if n%2 == 0 then
```

現段階では、Rubyの文法自体を理解する前なのでエラーメッセージの意味も理解しがたいが、必ずエラーのある行番号が表示されることを覚えておくとすばやく対処できる。

練習問題 2

1. 画面に「世界のみなさんこんにちは!」と3回出力するプログラム hw.rb を作成せよ。
2. 以下のプログラム gatagata.rb をインデントの揃ったものに修正せよ。

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

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

第3講 変数・制御構造

◆積算プログラム

まずは、意味のある動きをするプログラムを作り、実行してみよう。次のプログラム accum.rbを入力して、実際に実行してみよ。

accum.rb

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

sum = 0
while true
  STDERR.print "足したい数(終了は q): "
  line = gets.chomp
  if line == "q" then
    break
  end
  sum += line.to_i
  printf("これまでの合計は %d です\n", sum)
end
printf("合計は %d です\n", sum)
```

> 全ての行で [Tab] キーを押して正しいインデント位置に揃えること。

実行結果は以下のようになる。

```
% ./accum.rb
足したい数(終了は q): 1
これまでの合計は 1 です
足したい数(終了は q): 2
これまでの合計は 3 です
足したい数(終了は q): 3
これまでの合計は 6 です
足したい数(終了は q): 4
これまでの合計は 10 です
足したい数(終了は q): q
合計は 10 です
```

実行結果とプログラムを見比べ、以下の点について自分なりの予想を立てよ。

1. プログラム中何度も繰り返し処理されているのはどの範囲か?
2. キーボードからデータを読み込んでいるのはどの部分か?
3. breakは何をするものか?
4. breakのところが処理されるのはどういうときか?
5. sumという単語は何を意味しているか?
6. +=という記号は何を意味しているか?
7. printfは何をするものか?
8. printfのところにある %d は実行結果に出てきていないがどうなってしまったのか?

予想を立ててから次に進もう。

◆値と変数

計算機はなんらかのデータを受け取って、それに対して加工や計算などをした結果を吐き出す。動作中のプログラムの中では、やりとりされるデータは全て*値*として存在している。

◇数値と文字列

計算機の中で保持される値は大きく分けて*数値* と*文字列*に分類される。数値は加減乗除算のできる値で、文字列は文字が並んだ値である。Rubyでは文字列をダブルクォート(")か、シングルクォート(')で括って表記する。

例:

数値

```
5 123 3.5 -2
```

文字列

```
"Hello" "123" "太郎" '1+1'
```

数値の 123 と、文字列の "123" は足し算の結果が異なる。

例:

数値

```
123+123 → 246
```

文字列

```
"123"+"123" → "123123"
```

◇真偽値

「ある条件が成り立つか否か」、を表す値のことを*真偽値*という。

```
「条件が成立する」      →   true
「条件が成立しない」    →   false
```

例:

- trueとなる条件式

  ```
  5 > 3
  2 < 9
  ```

- falseとなる条件式

  ```
  5 < 3
  2 > 9
  ```

プログラムは、それを実行するときの状況に応じて処理を切り替える。後述する制御構造がそのための構文で、処理切り替えの判定規準に条件式を書く。

なお、Rubyの場合「空っぽ」を意味する値として *nil* がある。これも偽を意味する。

◇変数

変数 とは「値」を入れる「箱」のようなもので、計算機のメモリ内のどこかに確保される。ほとんどのプログラミング言語で利用できる仕組みである。

変数には好きな名前が付けられる。数学の変数と違って 2文字以上の名前でもよい。ただし、使える文字は英数字とアンダースコア(_)に限られ、なおかつ最初の文字は英小文字でなければならない。

> Rubyにはいくつかの変数の種類があるが、本講で扱う変数はローカル変数のみとする。ローカル変数とはプログラムの一部分で代入した結果が、他の部分に影響を及ぼさないものをいう。

変数に値を入れることを*代入*という。代入には = を使う。変数には数値でも、文字列でも、配列でも、Rubyで使えるものならどんな値でも入れてよい。数値(10進数)はそのまま、文字列は ""または、''で括る。

正しい代入例

```
x = 5
taiju = 50
name = "MOGAMI Taro"
```

まちがった代入例

```
12go = "Let's GO"（変数名は数字で始められない）
x = 太郎（文字列は引用符で括らなければならない）
```

代入は複数同時にできる。

```
x, y, z = 1, 2, 3
```

上記の例は、x=1、y=2、z=3になる。

◇変数の参照

変数に代入されている値を取り出すことを*参照*という。変数をそのまま書けば参照できる。

```
x = 5
y = x + 3     ← xの値を参照し、5 + 3 が計算される。
```

◆制御構造

プログラムは通常上から下へ一直線に解釈実行される。この流れを変えるのが*制御構造* である。以下で説明する各構造の「条件」の部分には、なんらかの条件式を書く。条件式は true または false を返す。よく使う制御構造を以下に示す。現段階ではきわめてよく使う if, while の意味をよく理解しておこう。

- *if…then…elsif…else…end*

```
if a1 [then]
 ほげほげ
elsif a2
 げほげほ
else
 うはうは
end
```

条件 *a1* が成立している(true)ならば「*ほげほげ*」の部分を実行。*a1*が成り立たなくて（falseで)、*a2*が成立していれば「*げほげほ*」の部分を実行。いずれでもなければ「*うはうは*」の部分を実行する。

elsif と else は不要なら省略できる。

- *while…end*

```
while a1
 とりゃ
end
```

条件 *a1* がtrueである間「*とりゃ*」の部分を繰り返し実行する。何度も繰り返す処理のことを*ループ*という。

- *case…when…when…when…else…end*

```
case a1
when b1,[,b2,..]
 くすくす
when d1
 こそこそ
else
 せこせこ
end
```

最初の *a1* の値を順次 when の直後と比べて行き、マッチしたときにその直後の文を実行する。たとえば、*a1* が b2 に等しいかマッチしたら「*くすくす*」の部分を、 *a1*がd1に等しいかマッチしたら「*こそこそ*」の部分を、どれでもなければ「*せこせこ*」の部分を実行する。 elseは必要なければ省略可能。

- *for…end*

```
for var in obj
 えいえいおー
end
```

ここでobjは単なる変数でなく、複数の値を持っている「配列」などを指定する。すると、objの先頭の値から順に var 変数に代入してその個数だけ「*えいえいおー*」の部分を繰り返す。実際の使用例は「配列」の講義のときに登場する。

- *break、next、redo*

これらは、while/until/for ループの中で使う。 break はそこでループの実行を中止しループを終了する。 next はループの残りを実行せず、次のループに突入する。 redoはその回のループを先頭からやりなおす(つまりループ入口の条件判断はしない)。

実際の使用例は第4講で登場する。

◆Rubyで利用できる演算

◇算術演算

Rubyで数値計算を行なうには、数学とほぼ同じ書き方で記述する。ただし、掛算記号は*、割算記号は/で表す。括弧は丸括弧（ ）一種類だけを使用し、式全体が1行で完結するように書く。

代表的な演算子を列挙する。

演算子	意味
+	加算
*	乗算
-	減算
/	除算
%	剰余
**	巾乗(べき乗)

一つの式に演算子が複数現れるときは、優先順位の高いほうが先に計算される。上記の演算子を優先順位の高い順に並べると以下のようになる。

高	←→	低
**	* / %	+ -

この順位は基本的に数学と同じで、たとえば

```
1+2*3
```

は、+よりも*が高い優先順位なので、 2*3 が先に計算されるので答は7となり、

```
1+2*3**2
```

は、*よりもさらに順位の高い**が最初に計算されるので答は1+2*9=1+18=19 となる。括弧（ ）があれば、先に括弧内が計算される。内側の括弧の方が高い優先順位を持つ。

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

は、1+2が最初に、次に *3 が、最後に**2 が計算される。

数式との対応をいくつか挙げておこう。

数学	Rubyの式
1+2×3	1+2*3
(1+2)÷3	(1+2)/3
$\frac{x+y+z}{3}$	(x+y+z)/3

◇代入を伴う演算

次のものは変数への代入を伴う。

代入演算子	意味
=	通常代入 x=5
+=	加算代入 x+=5 でxが5増える
*=	乗算代入 x*=5 でxが5倍になる
-=	減算代入 x-=5 でxが5減る
/=	除算代入 x/=5 でxが1/5になる
%=	剰余代入 x%=5 は x = x%5 と同じ
**=	巾乗代入 x**=5 は x=x**5と同じ

変数xの値が10であると仮定したときに、代入演算によってxがどう変わるかを示す。

代入演算	その後のxの値
x += 1	11
x *= 2	20
x -= 1	9
x /= 2	5
x %= 3	1
x **= 2	100

◇論理演算

制御構造の判定条件式等に利用する演算子のことを *論理演算子*という。基本的なプログラムでも頻繁に利用するものを以下に示す。

比較演算子	意味
==	左辺と右辺が等しいか
<	左辺が右辺より小さいか
<=	左辺が右辺以下か
>	左辺が右辺より大きいか
>=	左辺が右辺以上か
&&	「かつ」
||	「または」
!	否定
not	否定

いくつか例を示す。

- もしanswer変数の値(文字列)が "yes" と等しかったら「ほげほげ」

```
if answer == "yes" then
  ほげほげ
end
```

- もし x が12以下で、なおかつ xが1以上なら「ほげほげ」

```
if x <= 12 && x >= 1 then
  ほげほげ
end
```

- もし y が "はい" と等しいかまたは、 y が "yes" と等しい場合は「yesだってさ」そうでなければ「yesじゃないってさ」と出力する

```
if y == "はい" || y == "yes" then
  printf("yesだってさ\n")
else
  printf("yesじゃないってさ\n")
end
```

◆よく使うメソッド

◇メソッドとは

あらかじめ決められた(複雑な)処理を行なってくれるものを *メソッド*という。メソッドはRuby言語にあらかじめ備わっているものもあるし、(自分を含めた)誰かが作るものもある。

最初のプログラムによく出てきた printf もメソッドのひとつである。メソッドを呼び出すときは

> 何に対して.どうする(必要な情報)

という形にする(ピリオドに注目)。「どうする」の部分がメソッド名(たとえば printf)に相当する。「何に対して.」と「必要な情報」は必要ないメソッドもある。

メソッド名の直後に与える「必要な情報」のことを *引数(ひきすう)*という。

◇基本的メソッド

何かを入力して結果を出力するプログラムを作るために最低限必要なメソッドを以下に示す。

- *場所.gets* - 文字列の入力

  指定された場所から文字列を読み込んでその値を返す。「*場所.*」を省略するとRubyプログラム起動時に与えたファイルから読み込む。起動時にファイルを与えない場合は*標準入力*から読み込む(次講: gets へのデータの与え方で詳説)。文字列の末尾は必ず改行文字が入る。改行文字は *"\n"* で表す。

  使用例:

  ```
  foo = STDIN.gets
  ```

  上の例は、標準入力(次講で説明)から読み込んだ文字列をfoo 変数に代入している。

- *文字列.chomp*

  *文字列*の末尾が改行文字('\n') である場合にそれを切り取る。改行文字でなければなにもしない。

  使用例:

  ```
  line = gets.chomp
  ```

  上の例は、標準入力から1行読み込んで得た文字列の末尾から改行文字を除去した文字列を作成し、line 変数に代入している。

  chompは、元の文字列のコピーを作ってそこから末尾の改行文字をとったものを返す。chomp とほぼ同じメソッドにchomp!がある。 chomp! は、元の文字列のコピーを取らず直接改行をはぎ取ってその結果の文字列を返す。

- *文字列*.*to_i*

  文字列 を整数に変換した値を返す。

  使用例:

  ```
  hoge = "123"
  x = hoge.to_i
  ```

  文字列 "123" をそれに対応する数値に変換した結果 123を返し、xに代入している。

  ```
  gehogeho = gets
  k = gehogeho.to_i
  ```

  標準入力から読んだ文字列をそれに対応する正数に変換し、 kに代入している。

◇積算プログラムの復習

accum.rbのプログラムの各行の働きの説明を付けたのが以下のリストである。

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

sum = 0                 # sum変数を0とする
```

```
while true              # 対応するendまでをひたすら繰り返す
  STDERR.print "足したい数(終了は q): "
  line = gets.chomp     # 数値を入力させ line 変数に代入
  if line == "q" then   # 入力したものが "q" ならば…
    break               # whileループを抜ける
  end                   # ifでの場合分けはここで終わり
  sum += line.to_i      # lineを整数に直したものを sum に加算
  printf("これまでの合計は %d です\n", sum)
end                     # whileに対応するend
printf("合計は %d です\n", sum)
```

各行にある#記号より後ろの部分がその行で行なっている内容の説明文である。あらかじめ予想したものと比べ、個々のメソッドの役割を再確認しよう。

なお、#記号以降の部分をコメントという。プログラムの実行に影響を与えないので自由な文を書くことができる。分かりやすい注釈を入れながらプログラムを作る習慣を付けよう。

◇入出力を行なうプログラム例

読み込んだ文字列を2回出力するプログラムを動かして gets, chompメソッドの動きを観察しよう。

double.rb

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

while true
  print "好きなことは(qで終了)?: "
  koto = gets
  if koto == "q\n" then
    break
  end
  printf("%sだ、", koto)
  printf("%sだ楽しいな。\n", koto.chomp)
end
```

以下は、プログラムを起動し「祭」と入力した例である。

```
% ./double.rb
好きなことは(qで終了)?: 祭
祭
だ、祭だ楽しいな。
好きなことは(qで終了)?: q
```

「祭 [Return]」と入力すると、getsメソッドがそれを受け取り "祭\n" という文字列を作って返す。

```
koto = gets
```

の部分で、 "祭\n" という文字列が koto 変数に代入される。下から3行目の

```
  printf("%sだ、", koto)
```

によって出力されるのは、「祭」+「改行文字」であるが、次の行の

```
  printf("%sだ楽しいな。\n", koto.chomp)
```

は、文字列をchompしているので出力される「祭」の直後の改行がなくなっていることに注意する。

練習問題 3

1. 以下のRubyの各文を実行したあとのxの値を答えよ。

イ、
```
a = 5
b = 3
x = a + b
```

ロ、
```
x = 5
y = 3
x = x + y
```

ハ、
```
x = 5
x *= 4
```

ニ、
```
a = "9"
b = "99"
x = a+b
```

2. 以下のRubyの各文を実行した場合、"Yeah!\n" は何回出力されるか。

イ、
```
x = 5
if x > 5 then
  print "Yeah!\n"
end
```

ロ、
```
x = 5
while x > 0
  print "Yeah!\n"
  x -= 1
end
```

ハ、
```
x = 0
while x < 100
  if x%2 == 0 then
    print "Yeah!\n"
  end
  x += 1
end
```

3. 以下の日本語で書かれたものをRubyのif文を使って記述せよ。なお、整数aが整数bで割り切れるかどうかの判定をしたいとき、a%b が0に等しければ「割り切れる」、等しくなければ「割り切れない」とする。

イ、もし、yが400で割り切れたら「閏年」と出力する。

ロ、もし、yが4で割り切れないなら「閏年でない」と出力する。

ハ、もし、yが4で割り切れてなおかつyが100で割り切れないなら「閏年」と出力する。

ニ、もし、yが4で割り切れないかまたは、 yが100で割り切れてなおかつyが400で割り切れないなら「閏年でない」と出力する。

なお、作成時には以下のようなファイルを作成し、 ???の部分を埋める形でif文を書くとやりやすい。

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

print "西暦: "
y = gets.to_i
???
```

第4講 値の型変換

◆getsへのデータの与え方

以下のプログラムを emj.rb という名前で保存して実行してみよう。

emj.rb

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

print "英語は何点?: "
english = gets.chomp.to_i

print "数学は何点?: "
math = gets.chomp.to_i

print "国語は何点?: "
japanese = gets.chomp.to_i

printf("合計は%d点です\n", english+math+japanese)
```

```
% ./emj.rb
英語は何点?: 80
数学は何点?: 66
国語は何点?: 54
合計は200点です
```

プログラムを普通に起動するとgets メソッドに与えるデータはキーボードから入力することになる。このように入力が少ない(3個)のデータを一度入力するくらいならキーボードから入れるのも苦でないが、これを何度も、あるいは大量に手で入れるのは厳しい。本格的なプログラムに与えるデータはファイルに保存しておく。

emj.rb に与えたのと同じ並びの3つのデータからなるファイル emj.dat を作ろう。

emj.dat

```
80
66
54
```

cat コマンドを使って emj.dat ファイルの中味を画面に出力してみる。

```
% cat emj.dat
80
66
54
```

ファイルに保存したデータをプログラムの入力として与えることができる。以下の3つの方法は、どれも emj.dat に保存してあるデータをRubyプログラムのgetsに与えるためのコマンドラインである。

1. ./emj.rb < emj.dat
2. cat emj.dat | ./emj.rb
3. ./emj.rb emj.dat

上記のいずれかの方法でプログラムを起動すると以下のような出力画面が得られる。

```
英語は何点?: 数学は何点?: 国語は何点?: 合計は200点です
```

一見、画面が乱れてしまったように見えるが正しい結果が出ている。これは、ファイルの内容が自動的に

getsメソッドに供給されているからである。

gets メソッドがデータを読む場所は一概に「キーボード」とは言えないことに注意しておこう。

◇標準入力と標準出力

Rubyプログラムを起動するとき、プログラム名の後に何も指定しない場合に getsメソッドがデータを読む場所のことを正しくは *標準入力*という。Rubyプログラムに限らず、全てのコマンドの標準入力はファイルや別のプログラムの出力結果に変更することができる。

```
% ./emj.rb          # 標準入力は端末(キーボード)
% ./emj.rb < file   # 標準入力は file ファイルの中味
% cmd | ./emj.rb    # 標準入力は cmd コマンドの出力結果
```

Rubyプログラムのprintfメソッドは、通常端末画面に結果を出力する。プログラムの起動方法を変えるとこの出力先を変更することができる。

```
% ./emj.rb          # 標準出力は端末(画面)
% ./emj.rb > file   # 標準出力は file ファイルに保存される
% ./emj.rb | cmd    # 標準出力は cmd コマンドの入力として送られる
```

◆マッチ棒取りゲーム

以下のようなルールの、2人で遊ぶ「マッチ棒取りゲーム」がある。

1. マッチ棒が場に何本かある
2. そのうち一本は「外れマッチ棒」である
3. 2人で順番に場にあるマッチ棒を取っていく
4. 一人が一度に取れるマッチ棒は1本から3本
5. 最後に残った「外れマッチ棒」を取った人の負け

色々なバリエーションがあるが「最後の一本を取った者が負け」というルールは共通している。

いま、AさんとBさんがマッチ棒取りゲームをする場合を考える。

- 最初にマッチ棒が場に17本あった
- Aさんが先攻でマッチ棒を取り始めた

二人ともマッチ棒取りゲームに熟練していると仮定すると、どちらが勝つだろうか。

◆マッチ棒取りゲームのプログラム

以下のプログラムを保存して実際に実行してみよ。

match.rb

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

match = 17
```

```
print "1から3本の範囲でマッチ棒を取ります。最後の1本を取った方の負け!\n"

while match > 0
  printf("あなたの番: マッチ棒が%d本あります\n", match)
  print "何本取りますか?(1..3): "
  toru = gets.chomp.to_i
  if toru < 1 || toru > 3 then
    print "1から3の範囲で取って下さい.\n"
    redo
  end
  match -= toru
  if match < 1 then
    print "最後の1本を取りました。あなたの負けです! やあい\n"
    break
  end
  printf("わたしの番: マッチ棒が%d本あります\n", match)
  com_toru = 3-toru+1
  printf("私は%d本取りました。\n", com_toru)
  match -= com_toru
  if match < 1 then
    print "最後の1本を取りました。わたしの負けです! くやちいい!!\n"
    break
  end
end
```

実際に実行すると以下のようになる。

```
1から3本の範囲でマッチ棒を取ります。最後の1本を取った方の負け!
あなたの番: マッチ棒が17本あります
何本取りますか?(1..3): 3
わたしの番: マッチ棒が14本あります
私は1本取りました。
あなたの番: マッチ棒が13本あります
何本取りますか?(1..3): 2
わたしの番: マッチ棒が11本あります
私は2本取りました。
あなたの番: マッチ棒が9本あります
何本取りますか?(1..3): 1
わたしの番: マッチ棒が8本あります
私は3本取りました。
あなたの番: マッチ棒が5本あります
何本取りますか?(1..3): 0
1から3の範囲で取って下さい.
あなたの番: マッチ棒が5本あります
何本取りますか?(1..3): 1
わたしの番: マッチ棒が4本あります
私は3本取りました。
あなたの番: マッチ棒が1本あります
何本取りますか?(1..3): 2
最後の1本を取りました。あなたの負けです! やあい
```

実際に何度も実行してみると分かるが、マッチ棒をどういう本数で取っても必ず負ける。繰り返し遊んだあとで、ソースプログラムについて以下の予想を立てよう。

1. プログラム中何度も繰り返し処理されているのはどの範囲か?
2. match変数は何を意味するものか?
3. toru変数は何を意味するものか?
4. 「1から3の範囲で取ってください」と出るのはどういうときか?

◆値の型変換

getsでデータを読み込む場合、たとえ数字が入力されてもそれは必ず「文字列」として読み込まれる。数値として計算に利用したい場合は文字列を数値に変換する必要がある。逆に、数値を文字列に戻すことが必要になることもある。各々の変換をしてくれるメソッドを紹介する。

◇文字列から数値へ

- *to_i* (to Integer)

  数字が並んでいる文字列を10進数と見なして整数の数値に変換する。

  "123".to_i → 123

  "3.1415".to_i → 3

- *to_f* (to Floating Point Number)

  数字が並んでいる文字列を10進数と見なして浮動小数点数に変換する。

  "3.1415".to_f → 3.1415

小数を表現できる数値のことを*浮動小数点数*(Floating Point Number)という。一般に計算機では、整数と浮動小数点数は別の形式で内部処理される。このため、整数を割り算した結果は小数点以下が切り捨てられ、やはり整数になる。

上記の to_i, to_f いずれも、変換前の値は、文字列でなくとも構わない。たとえば、

```
pi=3.1415
q=pi.to_i
```

も正しい使い方で、qの値は3となる。

Rubyを含めた多くの言語では*小数点を含まない数は整数* として処理される。それを意識しないと割り算をするときに結果が異なるので注意が必要である。

```
x=5                     # xには「整数」の5が入る
y=x/2                   # 2で割っても「整数」なので2.5 → 2 となる
printf("%f\n", y)
2.000000                # %f を使ってもyが2なので2.0となる
```

割り算をする前には、割る方か割られる方のどちらかをto_f で浮動小数点数に変換しておかなければならない。以下のいずれでも、 yは正しく2.5となる。

```
# その(1)
x=5.0                   # 小数点を付けると最初から浮動小数点数
y=x/2                   # 5.0が浮動小数点数なので割り算も浮動小数点数
# その(2)
x=5                     # この時点ではまだ整数
y=x/2.0                 # 2.0が浮動小数点数なので割り算も浮動小数点数
# その(3)
x=5                     # この時点ではまだ整数
y=x/2.to_f              # 整数の2を浮動小数点数に変換
# その(4)
x=5                     # この時点ではまだ整数
y=x.to_f/2              # xを浮動小数点数に変換してから2で割る
```

◇数値から文字列へ

プログラムの中で数値を処理する場合でも、その結果を出力するためにはまた文字列に戻す必要がある。そ

のため

- 数値を文字列化する
- 数値を文字列化したものを直接出力

のいずれかの方法を知っておく必要がある。これまで利用していた printf は後者に相当する。まずprintf について説明し、続いて数値を文字列化する to_s, sprintf メソッドについて説明する。

- *printf* (print with format)

  最初の引数の文字列の中にある %X を、2個目以降の引数で与えた値に置き換えて出力する。

  ```
  printf("ほげー%X1うりゃ%X2", X1, X2)
  ```

  のようにすると、最初の$%X_1$が X_1の値に置き換えられ、次の$%X_2$が X_2の値に置き換えらる。 $%X_1$、 $%X_2$の部分には、 %d、%x、%f、%sなどが使える。

  - *%d* - (Decimal; 10進数)

    対応する値を10進整数の文字列に置き換える
  - *%x* - (heXa decimal; 16進数)

    対応する値を16進整数の文字列に置き換える
  - *%f* - (Floating point number)

    対応する値を10進浮動小数点数の文字列に置き換える
  - *%s* - (String)

    対応する値を文字列に置き換える

  %とd, x, f, s の間には置き換えたあとの桁数を指定する数を入れることができる。たとえば、

  ```
  printf("合計%5d円です。\n", price)
  ```

  とすると、priceの値が5桁に満たなくても5桁分確保して出力する。たとえば、price=50 だとしても、

  ```
  合計    50円です。
      ^^^^^
  ```

  と出力する。

  小数(浮動小数点数)に変換する%f では、全体の桁数と、小数点以下の桁数を指定できる。

  ```
  printf("答は%5.2fです。\n", answer)
  ```

  とすると、「全体で5桁、そのうち小数点以下2桁」で出力する。たとえば、answser=2.5 なら、

  ```
  答は 2.50です。
      ^^^^^
  ```

  と出力する。

  > Ruby-1.9以降は、日本語文字列の表示幅を合わせることを標準ではしてくれない。日本語文字列の表示幅を合わせたい場合は、プログラムの coding を euc-jp にしたうえで、プログラムの先頭に
  >
  > ```
  > Encoding.default_external = "binary"
  > ```

を記述するとよい。

桁数の指定に負の数値を与えると、「左詰め」して出力する。たとえば、正の値で桁数指定すると

```
printf("%10s, %20s\n", "中町太郎", "ナカマチタロウ")
```

とすると、

```
      中町太郎    ,          ナカマチタロウ
^^^^^^^^^^      ^^^^^^^^^^^^^^^^^^^^
```

と右詰めで出力するが、桁数に負の数を与えると

```
printf("%-10s, %-20s\n", "中町太郎", "ナカマチタロウ")
```

とすると、

```
中町太郎      ,  ナカマチタロウ
^^^^^^^^^^      ^^^^^^^^^^^^^^^^^^^^
```

と左詰めで出力する。

printfで数値を出力するのではなく、文字列に変換して別の変数に入れるようなときは以下のメソッドを利用する。

- *sprintf*

  printf で出力されるものを文字列として返す。以下の例を見よ。

```
month = 3
day   = 1
printf("%02d%02d", month, day)
#（文字列"0301"が出力される）

s = sprintf("%02d%02d", month, day)
#（文字列"0301"が変数sに代入される）
```

- *to_s* - (to String)

  すべての型の値を文字列に変換する。

練習問題 4

1. 整数を読み込み、それと同じ回数だけ「きみは天才!」と出力するプログラム gomasuri.rb を作成せよ。
2. 永遠に「きみは天才!」と出力し続けるプログラム homegoroshi.rb を作成せよ。プログラムを止めるには C-c をタイプする。
3. 底辺(cm)と高さ(cm)を読み込み、長方形の面積を計算した結果を出力するプログラム rectangle.rb を作成せよ。
4. 身長(cm)を入力させ、120cmから180cmの範囲内なら「搭乗できます」、範囲外なら「申し訳ありませんが搭乗できません」と出力するプログラム jetcoaster.rbを作成せよ。
5. 現在の年齢と西暦を読み込み、100歳になるまでの西暦と年齢を1行毎に出力するプログラム kingin.rb を作成せよ。完成したものを実行した例は以下のとおり。

```
% ./kingin.rb
現在の年齢は?: 1
今年の西暦は?: 2005
2006年であなたは2歳です。
2007年であなたは3歳です。
2008年であなたは4歳です。
   :（以下続く）
```

```
2104年であなたは100歳です。
```

6. 買った商品名と金額を順番に入力し続け、レジのレシートのように桁を揃えてきれいに出力するプログラム shopping.rb を作成せよ。商品名としてq を入れると入力を打ちきり、合計を出力するようにすること。実行例は以下のとおり。

```
% ./shopping.rb
商品名は?: バナナ
おいくら?: 198
バナナ          198円
商品名は?: コーラ
おいくら?: 37
コーラ           37円
商品名は?: マッチ棒
おいくら?: 66
マッチ棒          66円
商品名は?: q
お買い上げありがとうございました。
合計は    301円です。
```

マッチ棒取りゲーム match.rb と double.rbを参考にすると良い。

第5講 集合処理

◆配列

◇配列とは

これまで使っていた変数には値を一つしか入れていなかったが、複数の値をまとめて扱うこともできる。複数の値をまとめて保持するものが*配列*である。 Rubyでは、配列のことを *Array* という。

たとえば、変数 x を使う場合を考えてみる。

```
x = 5
```

とすると、x という名前の箱に、 5という値ただ一つが入る(下図参照)。

x
5

x に配列を入れることにすると何個でも値を入れられる。配列は複数の要素をカンマで区切って並べたものを大括弧 [ ] で括って表記する。

```
x = [5, 6, 3, 2, 9]
```

xには、5つの値が順序よく並んだものが代入される。

x				
5	6	3	2	9

試験の成績、売り上げ、などなど、計算機で処理したいと思う「世の中にある数」は、どれも1個だけではない。ひとつの項目に関する数が何件も存在するので、計算対象となる数は配列に入れて処理するのが普通である。

◇配列の要素

上記の配列は、5個の値が並んだものである。配列に含まれる一個一個の値のことを*要素*という。任意の位置の配列の要素にアクセスするには*添字*を付けて表現する。

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

変数の直後についている [0]から[4]の部分が添字で通常は0から始まる整数を指定する。

*変数[添字]*と書けば、「添字」のところに相当する要素を参照することもできるし、そこに代入することもできる。

```
printf("x0=%d\n", x[0])
```

とすると、先頭要素である5が出力され、

```
x[4] = 10
```

とすると、(0から数えて)4番目(つまり末尾)の要素が10に変わり、配列全体は

```
[5, 6, 3, 2, 10]
```

と変わる。

配列を持つ変数を添字なしで書くと配列全体を意味する。したがって、

```
y = x
```

とすると、変数yに配列全体が代入される。

◇配列への値の追加手順

配列を利用する場合、最初から要素が何個になるか分かっているとは限らない。 Rubyでは必要に応じて配列の要素数を増やすことができる。ある変数を配列として利用したい場合は、最初に要素数ゼロの空っぽ配列を代入する。大括弧の中に何も書かないと「空の配列」を意味する。

```
x = []
```

この状態から以下のようにすれば配列に値を追加できる。

```
x[0] = "Hello!"
x[1] = 5
x[2] = 3.1415
```

しかしながらこのように配列の末尾の添字を具体的に指定する方法はあまり使わず、配列用のメソッド <<で要素を末尾に順次追加するほうが利用しやすい。

```
x << "Hello!"
x << 5
x << 3.1415

 :
(以下何個でも)
```

同じことは配列に備わるpushメソッドを用いて以下のようにも書ける。

```
x.push("Hello!")
x.push(5)
x.push(3.1415)
```

◇配列からの値の取り出し手順

配列内の要素は、一つ一つ取り出して利用する。要素の値だけを使うときは eachメソッドまたは for文を使用し、添字が必要なときは each_with_indexメソッドまたはwhile文を使う。

```
a = [9, 6, 4, 3, 2]
```

のときにaに配列が入っている場合の繰り返し手順を以下に示す。

■要素だけ必要な時の繰り返し - each, for

eachメソッドで取り出し変数を指定して繰り返し

```
a.each do |x|
```

```
  puts x
end
```

for文に取り出し変数を与えて繰り返し

```
for x in a
  puts x
end
```

実行結果はいずれも以下のようになる。

```
9
6
4
3
2
```

■添字と要素を同時に利用する繰り返し - each_with_index, whileループ

添字が0から始まることに注意する。

each_with_indexメソッドで要素と添字を入れる変数を指定して繰り返し

```
a.each_with_index do |x, i|
  printf("第%d要素(%d番目)は %d です。\n", i, i+1, x)
end
```

while文で繰り返し

添字を表す補助変数を利用して、添字0から配列の個数より 1小さい数まで繰り返す。配列のlengthメソッドで個数を調べる。

```
i = 0                           # ループ用補助変数で他と衝突しなければなんでもよい
while i < a.length              # a.lengthは配列aの要素数（後述）
  printf("第%d要素(%d番目)は %d です。\n", i, i+1, a[i])
  i += 1                        # 補助変数 i の値を1増やす
end
```

実行結果はいずれも以下のようになる。

```
第0要素(1番目)は 9 です。
第1要素(2番目)は 6 です。
第2要素(3番目)は 4 です。
第3要素(4番目)は 3 です。
第4要素(5番目)は 2 です。
```

◇配列を操作するメソッド

配列に対して行なえる、次のメソッドを覚えておこう。

- *length*

  配列の長さを返す。たとえば、

  ```
  x = [1, 2, "foo"]
  ```

  であれば、x.length は 3 となる。

- *size*

  配列の長さを返す。length メソッドと同じ。

- *shift*

  配列の先頭の要素を取り除いた上で、取り除かれた要素の値を返す。たとえば、

  ```
  y = ["1", "abc", "bar"]
  ```

  であれば、y.shift は "1" となる。さらに、先頭が取り除かれて y 自体は ["abc", "bar"] となる。

- *sort*

  配列の要素を小さい順に並べ換えた結果を返す。たとえば、

  ```
  z = [7, 5, 1, 22]
  ```

  ならば、z.sort は

  ```
  [1, 5, 7, 22]
  ```

  となる。並べ換えた結果をあとで利用したい場合は、別の変数に代入する。

  ```
  y = z.sort
  ```

  sortを、文字列を要素に含む配列に使うと、辞書順に並べ換えた結果を返す。

  ```
  meibo = ["やまだたろう", "なかまちたろう",
           "いいもりはなこ", "あいたあきら"]
  ```

  ならば、meibo.sort は、

  ```
  ["あいたあきら", "いいもりはなこ",
   "なかまちたろう", "やまだたろう"]
  ```

  となる。なお、sort は、元の配列のコピーを取ってから並べ換え作業をするので、meibo.sort としても、 meibo変数の値は変わらない。meibo変数の値そのものの順番を壊して並べ換えたいときは

  ```
  meibo.sort!
  ```

  とする。元の値を直接操作することを*破壊的操作*という。

- *reverse*

  配列の要素の順序を逆にした配列を返す。たとえば、

  ```
  y = [1, 5, 7, 22]
  ```

  なら、y.reverse は、

  ```
  [22, 7, 5, 1]
  ```

  となる。

  sort, reverse を組み合わせると、逆順に並べ換えた結果が得られる。

  ```
  meibo = ["やまだたろう", "なかまちたろう",
           "いいもりはなこ", "あいたあきら"]
  ```

  のとき、

  ```
  meibo.sort.reverse

  => ["やまだたろう", "なかまちたろう", "いいもりはなこ",
      "あいたあきら"]
  ```

  となる。

  reverseもsort同様、元の変数の値は変わらない。変えたい場合は、reverse! メソッドを使う。

◆コマンドラインとの情報のやりとり

◇コマンドライン引数

Rubyプログラムを実行するときに、プログラム自身に与えた引数を Rubyプログラムから利用できる。たとえば、

```
% ./test.rb a b c d
```

のように起動した場合、与えた引数 `a', `b', `c', `d' は、自動的に配列変数 *ARGV*に代入される。具体的には、

```
ARGV[0] = "a"
ARGV[1] = "b"
ARGV[2] = "c"
ARGV[3] = "d"
```

ARGV			
ARGV[0]	ARGV[1]	ARGV[2]	ARGV[3]
"a"	"b"	"c"	"d"

というふうに値が格納される。これを利用すると、プログラム起動時に与えた情報で動作を決定するものが作れる。

たとえば、1から指定した数までの総和を求めるときに、

```
% ./sum.rb 100
```

で100までの総和を求めるようなプログラム（sum.rb)は以下のように書ける。

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

if ARGV[0] == nil
  STDERR.puts "起動時に指定した値までの自然数を全部足します."
  STDERR.puts "例: ./sum.rb 100  --- 1から100までの総和を求める."
  exit(1)
end

sum = 0
goal = ARGV[0].to_i     # 文字列なので整数に直す

goal.downto(1) do |x|
  sum += x
end

printf("1から%dまでの自然数の総和は%dです\n", goal, sum)
```

◇STDERR

プログラム先頭で ARGV[0] が nil かどうか比較しているが、これは対応する引数が与えられなかった場合の判定である。コマンドラインで引数を指定しなかった場合(あるいは個数が足りない場合)の処理も考慮する必要がある。

このように、プログラムを起動するときに利用者が起動方法を間違えた場合には適切なメッセージを端末に出力する必要がある。そのような場合はたんに printf するのではなく、 *STDERRに対してprintfする* 必要がある。たんに printf すると、

```
% ./sum.rb > hogehoge
```

のように起動した場合、エラーメッセージさえもファイル（上記の場合*hogehoge*)に書かれるため、利用者の目に触れずに終わってしまう。

◇終了ステータス

Rubyプログラムの実行が終わるときに、プログラム全体が「うまくいった」か「うまくいかなかったか」をシステムに対して教えることができる。この情報のことを*終了ステータス*という。

終了ステータスは整数で表し、

- プログラムがうまく行ったとき
  → 0
- うまく行かなかったとき
  → 0以外

とする決まりになっている。sum.rb でエラーメッセージを表示したあと、

```
exit(1)
```

としているのは、システムに対し「予定した仕事ができずに終わっちゃった」ことを示すためである。

*exit*は、直ちにプログラムの実行を終了しシステムに終了コードを返す。プログラム実行中にエラーが発生した場合などはプログラムを継続する意味がなくなるため、exitを利用して終了する。

◆CSVと配列

◇レコードとフィールド

我々が日常的に利用する「データ」は多くの場合*表形式* で2次元の形で表現できる。

たとえば、次のような電話帳データを考える。

漢字氏名	カタカナ氏名	電話番号	メイルアドレスその1	住所

全ての項目が揃った1単位のデータのことを*レコード*、レコードの中にあるひとつひとつの項目のことを*フィールド*という。この例に従うデータ並びとしては以下のものが考えられる。

公益太郎	コウエキタロウ	0234-567-8901	koeki@koeki.example.jp	魔界
飯森花子	イイモリハナコ	0234-567-8902	iimori@koeki.example.jp	結界

各項目がすべて揃ったものを「レコード」、漢字氏名やカタカナ氏名などの最小項目単位を「フィールド」という。表計算ソフトなどでは1レコードを1行に、 1フィールドを1セルに書き込んで表現する。表計算ソフトなどではデータの集合を保存するときにはデフォルトでそのソフト固有の保存形式で記録している。このため、なんらかのデータを専用ソフトで作成してファイルに保存したものは、その保存形式に対応したソフトを使わない限り開くことができない。

これに対し、あらゆる種類のソフトウェアで読み書きできる形式がいくつかある。そのうちの一つが*CSV*形式である。

◇CSVの構造

複数のフィールドをフィールドごとにカンマ(,)で区切り、 1レコードを1行におさめたテキストファイル形式をCSV形式という(Comma Separated Value)。CSV 形式はアプリケーションに依存しない一般的なデータファイル形式で、データ処理を行なうプログラムであればほぼ全てのソフトウェアから利用できる。このため、誰にでも参照してもらいたいデータファイルは CSV 形式にして保存・受け渡しするのが基本である。また、Ruby には CSV ファイルの読み書きを容易に行なえるライブラリがあるため、「データ入力を表計算ソフトで、計算処理をRubyプログラムで行なう」といった効率的なデータ処理設計が可能である。

CSV形式は、値をカンマで区切って列挙する。各値は、

- 数値はそのまま
- 文字列はそのまま、もしくはダブルクォートで括る

ようにする。フィールドに空白文字かカンマを含む文字列が来るときは必ずダブルクォートで括る。たとえば、

- Hello（文字列）
- 123（整数）
- 456（文字列）
- Hello, world（文字列）

という4つのフィールドをまとめたレコードを CSV で表すときは

```
Hello,123,"456","Hello, world"
```

のように記す。なお、文字列自身にダブルクォートが含まれるときは、ダブルクォートを重ねて表す。たとえば「He said "Hello".」という文字列は、

```
"He said ""Hello""."
```

となる。

◇RubyのCSVライブラリ

Ruby付属の CSV ライブラリを利用すると、正しく記述された CSV ファイルを一括で読み込むことができる。

■CSVファイルの読み込み

CSV ファイルからデータを読むには CSV.read メソッドを使う。

```
require 'csv'
data = CSV.read(CSVファイル)
```

簡単なCSVファイルを作ってCSVライブラリから読み込んでみる。以下のようにして word.csv を作る。

```
% cat > word.csv
bird,2
cat,4
octopus,8
 C-d
```

このようなCSVファイルをCSV.read()で読み込むと:

- 1行分の各項目(レコード)を集めた*配列*（内側の配列）
- それをさらに集めた*配列*（外側の配列）

つまり、『配列の配列』が用意される。

```
          ↓全体の配列(配列の配列)↓
<------------------------------------------>
[["bird", "2"], ["cat", "4"], ["octopus", "8"]]
 <----------> <----------> <-------------->
 ↑1行目の配列  ↑2行目の配列 ↑3行目の配列
```

irbを使って読み込む実験をしてみる。

```
% irb
irb> require 'csv'
=> true
irb> tango = CSV.read("word.csv")
=> [["bird", "2"], ["cat", "4"], ["octopus", "8"]]      # 「配列の配列」の形になっている
irb> tango.length
=> 3                                # 3行(レコード)なので長さは3
irb> tango[0]
=> ["bird", "2"]                    # 第0要素は先頭のレコード(の配列)
irb> tango[0][1]
=> "2"                              # 先頭レコードの第1(2つ目)の要素は"2"
```

このCSVファイルを読み取って、内容を出力するプログラム例を示す。

word-print.rb

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

tango = CSV.read("word.csv")

tango.each do |row|             # 1レコードの値の集合が配列でrowに入る
 # rowには [第1列の項目, 第2列の項目] の配列が毎回入る
 printf("[%s]の足は%d本です\n", row[0], row[1].to_i)
end
```

実行例は以下のとおり。

```
% ./word-print.rb
[bird]の足は3本です
[cat]の足は4本です
[octopus]の足は8本です
```

◇問題

1. 以下の内容をword2.csvに保存してから実行すると、第1列の内容を最初に出力したあと gets で待機し、[Enter] を押すと第2列の内容を出力するプログラム word2.rb を作成せよ。

   ```
   花より,団子
   泣きっ面に,蜂
   三度目の,正直
   二度あることは,三度ある
   ```

   実行例を以下に示す。

   ```
   % ./word2.rb
   「花より」といえば?: [Enter]
   「団子」です。
   「泣きっ面に」といえば?:
   ```

2. 以下の表の内容を表すCSVファイル tohoku.csv を作成せよ。

青森県	りんご	白鳥
岩手県	桐	キジ
宮城県	ミヤギノハギ	雁
秋田県	ふきのとう	やまどり
山形県	紅花	オシドリ
福島県	ネモトシャクナゲ	キビタキ

3. 上のCSVファイルの1行目のレコード(青森県とりんごと白鳥の3つ組)を、配列の表記で書き表せ。

   答は ruby -r csv -e 'p CSV.read("tohoku.csv")[0]' で得られるのでその前に考えてみよ。

4. 上の tohoku.csv を CSV.read で開き、以下のように各レコードの内容を出力するプログラム tohoku.rb を作成せよ。

```
% ./tohoku.rb
青森県の花はりんご，鳥は白鳥です。
岩手県の花は桐，鳥はキジです。
宮城県の花はミヤギノハギ，鳥は雁です。
秋田県の花はふきのとう，鳥はやまどりです。
山形県の花は紅花，鳥はオシドリです。
福島県の花はネモトシャクナゲ，鳥はキビタキです。
```

◆成績処理の例

配列を利用して、試験の得点の処理をするプログラムを作成してみよう。国語の試験の結果ファイルとして以下のようなデータを作成しよう。

score.csv

```
山田太郎,50
中町太郎,90
飯森花子,91
鶴岡一人,60
酒田三吉,52
三川一二三,12
```

このデータを読み込み、全員の平均点を計算したうえで、個人個人の平均点との得点差も合わせて出力するようなプログラムを作ってみよう。

平均点を計算するためには、まず全員のデータを読み込んで、得点と人数を求める必要がある。さらに、個人個人の平均点との差を出力するためには、全員の得点を記憶しておく必要があるので、配列に、名前・得点を入れておかなければならない。

まず最初に、データファイルを全部読み、名前と得点を配列に記憶しよう。 CSVデータを読む部分は次のように書ける。

```
seiseki = CSV.read("score.csv")
```

ただし、この書き方だとデータファイル名を score.csv 以外にできないので、プログラム起動時に自由に指定できるようにする。このためにARGV配列を用いて、CSV.readでは、

```
seiseki = CSV.read(ARGV[0])
```

としておき、ターミナルから実行するときにコマンドライン引数として指定する。

```
% ./プログラム名.rb score.csv
```

のようにすると、プログラム名の直後に書いた単語が ARGV[0] に入る。

score.csvファイルをCSV.readで読み込むと、
1行目のレコードは ["山田太郎", "50"],
2行目のレコードは ["中町太郎", "90"], ……
のように配列化され、さらにそれが配列化されたものが seiseki 変数に入る。

続いて、名前用、得点用2つの配列を用意するとともに合計得点を記録する変数を加えるようにした上で、これをRubyプログラムに直すと以下のようになる。

```
name  = []
score = []
sum = 0                          # 合計点計算用
seiseki.each do |row|            # 1レコードずつ取り出したものが順次rowに入る
  # rowには ["山田太郎", "50"] のような形式が順番に入る
  name  << row[0]                # 第0要素は氏名
  score << row[1].to_i           # 第1要素は得点の文字列なので .to_i で整数化
  sum += row[1].to_i
end
```

これで変数 sum に合計点が入り、配列の length メソッドで要素数、すなわち人数が分かるので平均点が計算できる。

```
average = sum.to_f/score.length          # name.lengthで割ってもよい
```

割り算をする場合は整数をto_fメソッドで浮動小数点数に変換することを忘れないようにしよう。

最後に、全員の、氏名、得点、平均点との差、をループで出力する。

```
i = 0
print "--氏名--------------+-得点-+-平均との差--\n"
while i < score.length
  printf("%-20s %5d   %5.1f\n", name[i], score[i], score[i]-average)
  i += 1
end
```

添字変数 i を自動的に増やす以下の方法もある。

```
print "--氏名--------------+-得点-+-平均との差--\n"
for i in 0..(score.length-1)             # 0 から(要素数-1)まで繰り返す
  printf("%-20s %5d   %5.1f\n", name[i], score[i], score[i]-average)
end
```

N..*M* は、2つの自然数 *N* から *M* (*N*<*M*)まですべての整数の集合を表す。

以上を組み合わせてプログラムは完成する。

score.rb

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

require 'csv'                    # CSVライブラリを呼び出す
require '
```

kprintf.rb

```
# kanji-aware printf for both Ruby1.8 or 1.9+
class String
  require 'kconv'
  if defined?("".force_encoding)
    def toeucbin()
      self.toeuc.force_encoding("binary")
```

```
end
```

上記プログラムで利用している kprintf.rb を同じディレクトリに保存してから実行してみよう。

```
% ./score.rb score.csv
--氏名--------------+-得点-+-平均との差--
山田太郎                50    -9.2
中町太郎                90    30.8
飯森花子                91    31.8
鶴岡一人                60     0.8
酒田三吉                52    -7.2
三川一二三              12   -47.2
```

◆ハッシュ

◇ハッシュとは

身の回りにあるものや人には様々な性質が備わっている。そのような性質を記す方法は様々なところで用いられている。

特定のキャラクタに対する属性を管理する場合を考える。たとえば以下の一覧にあるものを表現したい。

[0]



名前	ドナポン
属性	甘
最大MP	250
初期MP	20

[1]



名前	ラジャイン
属性	土
最大MP	450
初期MP	2

[2]



名前	リッタン
属性	沼
最大MP	150
初期MP	30

[3]



名前	音助
属性	波
最大MP	100
初期MP	80

図：キャラクタ集の例

上記の最初のキャラクタの場合、

- 名前 → ドナポン
- 属性 → 甘
- 最大MP → 250
- 初期MP → 20

という関係性を持つ。このように、特定のキーワードに対応する値を持つ関係性の集合を*ハッシュ*といいう。ここで、各関係の左辺にある元となるキーワードのことを*キー(key)*、右辺にある対応値のことを*バリュー(value)*という。上の例では以下のようになる。

キー	バリュー
"名前"	"ドナポン"
"属性"	"甘"
"最大MP"	250
"初期MP"	20

◇Rubyでのハッシュ表現

Rubyではこの対応関係を

```
{キー1 => バリュー1, キー2 => バリュー2, ...}
```

の形式で表す。上のキャラクタ例は以下のように表現する(値の部分の文字列と数値の違いに注意)。

```
{"名前" => "ドナポン", "属性" => "甘", "最大MP" => 250, "初期MP" => 20}
```

この値を何らかの変数に入れて処理することになる。変数aに入れて、「名前」に対応する値を取り出すには以下のようにする(実際にirbで試してみること)。

```
% irb
irb> a = {"名前" => "ドナポン", "属性" => "甘", "最大MP" => 250, "初期MP" => 20}
irb> a["名前"]
=> "ドナポン"
```

このように、ハッシュを持っている変数にキーを添え字に指定するとバリューを取り出せる。

逆に、値を設定するときは添字付きで代入する。

```
irb> a["最大MP"] = 350
=> 350
irb> a
=> {"名前"=>"ドナポン", "属性"=>"甘", "最大MP"=>350, "初期MP"=>20}
```

◇ハッシュオブジェクト

ハッシュ形式の値はプログラムを書くときに定まるものだけではない。たとえば以下のような新キャラクタを、プログラム実行後に利用者に好きな値を入力して決めてよいことにしたい場合を考える。

[3]	
	
名前	???
属性	???
最大MP	???
初期MP	???

このような場合は、キーとバリューのペアを一つも持っていない *空のハッシュ値* を入れてから代入させる。

```
newchar = {}            # {} が空のハッシュ値
```

```
print "名前は何にしますか?: "
newchar["名前"] = gets.chomp
print "属性は何にしますか?: "
newchar["属性"] = gets.chomp
print "最大MPはいくつにしますか?: "
newchar["最大MP"] = gets.to_i
print "初期MPはいくつにしますか?: "
newchar["初期MP"] = gets.to_i
```

空のハッシュ値は Hash.new でも生成できる。

```
newchar = Hash.new              # newchar = {} と同じ
```

ハッシュ形式の値のことを*ハッシュオブジェクト*といい、 Hash.new で空のハッシュオブジェクトが生成できる。

newchar.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
newchar = Hash.new                      # newchar = {} と同じ
print "名前は何にしますか?: "
newchar["名前"] = gets.chomp
print "属性は何にしますか?: "
newchar["属性"] = gets.chomp
print "最大MPはいくつにしますか?: "
newchar["最大MP"] = gets.to_i
print "初期MPはいくつにしますか?: "
newchar["初期MP"] = gets.to_i
puts newchar
```

newchar.rbを保存して動かしてみよ。

```
% ./newchar.rb
名前は何にしますか?: たむどん
属性は何にしますか?: 沼
最大MPはいくつにしますか?: 999
初期MPはいくつにしますか?: 10
{"名前"=>"たむどん", "属性"=>"沼", "最大MP"=>999, "初期MP"=>10}
```

◇問題1

1. 図1の[1]のキャラクタのもつ性質をハッシュで表し、変数bに代入する文を書け。
2. 図1の[2]のキャラクタのもつ性質をハッシュで表し、変数cに代入する文を書け。
3. 山形県に関して、「都道府県名」「県庁所在地」「人口」をキーとするハッシュを作り、変数ymgtに代入する文を書け。 irbで実行して確認すること。
4. 秋田県に関して、「都道府県名」「県庁所在地」「人口」をキーとするハッシュを作り、変数aktに代入する文を書け。 irbで実行して確認すること。

※都道府県ごとの人口は 総務省:住民基本台帳に基づく人口、人口動態及び世帯数 の「住民基本台帳人口・世帯数、令和4年人口動態（都道府県別）」などを参考にするとよい。

◇ハッシュ値の集合

ハッシュ値は一つの個体の性質の集合を表すことができる。一般的には個体が複数存在するので、これらを配列の形で表現することが多い。

図1の[0]から[3]の4つのキャラクタは以下のようにハッシュ表現できる。

```
# [0]
{"名前" => "ドナポン", "属性" => "甘", "最大MP" => 250, "初期MP" => 20}
# [1]
{"名前" => "ラジャイン", "属性" => "土", "最大MP" => 450, "初期MP" => 2}
# [2]
{"名前" => "リッタン", "属性" => "沼", "最大MP" => 150, "初期MP" => 30}
# [3]
{"名前" => "音助", "属性" => "波", "最大MP" => 100, "初期MP" => 80}
```

4つの各値を要素とする配列にする。つまり、

```
[ハッシュ値1, ハッシュ値2, ハッシュ値3, ハッシュ値1]
```

の形式で並べると以下のようになる。

```
[
 {"名前" => "ドナポン", "属性" => "甘", "最大MP" => 250, "初期MP" => 20},  # ←カンマに注意
 {"名前" => "ラジャイン", "属性" => "土", "最大MP" => 450, "初期MP" => 2},
 {"名前" => "リッタン", "属性" => "沼", "最大MP" => 150, "初期MP" => 30},
 {"名前" => "音助", "属性" => "波", "最大MP" => 100, "初期MP" => 80}
]
```

変数に代入して、個別にアクセスしてみる。irbで以下を実行してみよ。

```
% irb
irb> chara = [
 {"名前" => "ドナポン", "属性" => "甘", "最大MP" => 250, "初期MP" => 20},
 {"名前" => "ラジャイン", "属性" => "土", "最大MP" => 450, "初期MP" => 2},
 {"名前" => "リッタン", "属性" => "沼", "最大MP" => 150, "初期MP" => 30},
 {"名前" => "音助", "属性" => "波", "最大MP" => 100, "初期MP" => 80}
]
irb> chara
=>
[{"名前"=>"ドナポン", "属性"=>"甘", "最大MP"=>250, "初期MP"=>20},
 {"名前"=>"ラジャイン", "属性"=>"土", "最大MP"=>450, "初期MP"=>2},
 {"名前"=>"リッタン", "属性"=>"沼", "最大MP"=>150, "初期MP"=>30},
 {"名前"=>"音助", "属性"=>"波", "最大MP"=>100, "初期MP"=>80}]
# 先頭要素[0]を取り出す
irb> chara[0]
=> {"名前"=>"ドナポン", "属性"=>"甘", "最大MP"=>250, "初期MP"=>20}
# 4番目[3]を取り出す
irb> chara[3]
=> {"名前"=>"音助", "属性"=>"波", "最大MP"=>100, "初期MP"=>80}
# 4番目[3]の名前を取り出す
irb> chara[3]["名前"]
=>"音助"
# 4番目[3]の初期MPを上書きして90に更新する
irb> chara[3]["初期MP"]=90
=> 90
irb> chara[3]
=> {"名前"=>"音助", "属性"=>"波", "最大MP"=>100, "初期MP"=>90}
```

この(chara)配列に、新たなキャラクタのハッシュ値を追加する例を示す。charaは配列なので要素として追加する。

```
irb> chara << {"名前"=>"ハテナ", "属性"=>"謎", "最大MP"=>400, "初期MP"=>0}
=>
[{"名前"=>"ドナポン", "属性"=>"甘", "最大MP"=>250, "初期MP"=>20},
 {"名前"=>"ラジャイン", "属性"=>"土", "最大MP"=>450, "初期MP"=>2},
 {"名前"=>"リッタン", "属性"=>"沼", "最大MP"=>150, "初期MP"=>30},
 {"名前"=>"音助", "属性"=>"波", "最大MP"=>100, "初期MP"=>90},
 {"名前"=>"ハテナ", "属性"=>"謎", "最大MP"=>400, "初期MP"=>0}]
```

5個目(第4)要素が増えた。

◇問題

1. ある八百屋にあった3品目の単価リスト:

   りんご:150 みかん:30 いちご:20

   をハッシュ形式で表現し、それを変数yaoyaに代入する文を書け(答は何通りもある)。
2. 1のyaoya変数に「梨:120」を表現する値を代入する文を書け(答は何通りもある)。
3. 東北6県の各県について「都道府県名」「県庁所在地」「人口」をキーとするハッシュを作り、それらを6要素の配列としたものを変数 tohoku に代入する文を書け。

◆CSVとハッシュ

◇ヘッダ付きCSV

2科目分の試験の得点を記した score2.csv を考える。

```
山田太郎,50,70
中町太郎,90,80
飯森花子,91,60
鶴岡一人,60,45
酒田三吉,52,82
三川一二三,12,98
```

このようなデータを渡されても50と70が何を意味しているのか不明なので、通常は見出しとなるヘッダ行をつけて以下のように書く。

score2h.csv

```
氏名,数学,英語
山田太郎,50,70
中町太郎,90,80
飯森花子,91,60
鶴岡一人,60,45
酒田三吉,52,82
三川一二三,12,98
```

表計算プログラムでみると以下のように見えるはずである。

	A	B	C
1	氏名	数学	英語
2	山田太郎	50	70
3	中町太郎	90	80
4	飯森花子	91	60
5	鶴岡一人	60	45
6	酒田三吉	52	82
7	三川一二三	12	98

CalcやExcelを用いて上記のように値を入力し、保存するときに「テキストCSV形式」を選択すると楽にCSVファイルを作成できる。

1行目が見出しとなる
ヘッダ行を持つCSVファイルは CSV.read を用い

```
CSV.read(CSVファイル名, headers: true)
```

とすると、ヘッダ構造を持った値になる。具体的には以下のように利用する。

```
seiseki = CSV.read("score2h.csv", headers: true)
```

こうすると、seiseki 変数は、行ごとの値をヘッダ行の項目名で取り出せる値を集めた配列になる。

score2h.csvを保存してから、以下のようにirbで読み取る実験をしよう。

```
% irb -rcsv     # CSVライブラリを最初からrequireする
irb> seiseki = CSV.read("score2h.csv", headers: true)
irb> seiseki[0]         # 先頭要素を取り出す
=> #<CSV::Row "氏名":"山田太郎" "数学":"50" "英語":"70">
irb> seiseki[0]["数学"]         # 先頭要素の "数学" 値を取り出す
=> "50"
```

このように、各レコード(行)のうち、項目に対応する値を「添字」の形でアクセスできる。単純に全部のCSVデータを出力するプログラム例を示す。

score-csvheader.rb

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

seiseki = CSV.read(ARGV[0], headers: true)
seiseki.each do |row|            # レコードごとに値をrowに入れて繰り返す
  name = row["氏名"]             # 添字 "氏名" で氏名欄の値
  math = row["数学"].to_i        # 添字 "数学" で数学欄の値 .to_iで整数化
  eng  = row["英語"].to_i        # 添字 "英語" で英語欄の値 .to_iで整数化
  printf("%sさんの数学は%d点、英語は%d点です。\n", name, math, eng)
end
```

実行例を示す。

```
% ./score-csvheader.rb score2h.csv
山田太郎さんの数学は50点、英語は70点です。
中町太郎さんの数学は90点、英語は80点です。
飯森花子さんの数学は91点、英語は60点です。
鶴岡一人さんの数学は60点、英語は45点です。
酒田三吉さんの数学は52点、英語は82点です。
三川一二三さんの数学は12点、英語は98点です。
```

◇列の取り出し

headers:trueで読んだ値からは、特定の列のみ取り出せる。

```
seiseki = CSV.read("score2h.csv", headers: true)
```

で代入されている前提でseiseki["数学"]は、数学のみを「縦に」集めた配列、つまり

	A	B	C
1	氏名	数学	英語
2	山田太郎	50	70
3	中町太郎	90	80
4	飯森花子	91	60
5	鶴岡一人	60	45
6	酒田三吉	52	82
7	三川一二三	12	98

のB列の値の部分を集めた配列

```
["50", "90", "91", "60", "52", "12"]
```

となる(文字列であることに注意)。同様に、

```
seiseki["英語"]
```

は、

```
["70", "80", "60", "45", "82", "98"]
```

となる。このことから、たとえば「数学の合計点」を求めるには以下のようにすればよい。

```
sum = 0
seiseki["数学"].each do |pt|
  sum += pt.to_f
end
```

> 配列にあるsumメソッドを用いると以下のように簡単に書ける。
>
> ```
> sum = seiseki["数学"].sum{|pt| pt.to_f}
> ```
>
> 同様に平均点は、以下で求められる。
>
> ```
> avg = seiseki["数学"].sum{|pt| pt.to_f}.to_f/seiseki.length
> ```

◇ヘッダの取り出し

headers:trueで読んだ値から、ヘッダ(見出し項目)一覧を取り出せる。

```
seiseki = CSV.read("score2h.csv", headers: true)
```

と代入されている場合、見出し行を集めた配列はheaders にアクセスすることで得られる。

```
seiseki.header
["氏名", "数学", "英語"]
```

	A	B	C
1	氏名	数学	英語
2	山田太郎	50	70
3	中町太郎	90	80
	:		

これを利用すると、見出し行付きのCSVファイルから自動的に見出しを読み取って集計できる。キャラ一覧CSV を読み取り、項目名で分類したものを出力してみる。

byheader.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'csv'
csv = CSV.read(ARGV[0], headers:true)
for field in csv.headers        # csv.headers.each do |field| でも同じ
  printf("「%s」列の値一覧: %s\n", field, csv[field])
end
```

実行例:

```
% ./byheader.rb score2h.csv
「氏名」列の値一覧: ["山田太郎", "中町太郎", "飯森花子", "鶴岡一人", "酒田三吉", "三川一二三"]
「数学」列の値一覧: ["50", "90", "91", "60", "52", "12"]
「英語」列の値一覧: ["70", "80", "60", "45", "82", "98"]
```

◇問題

1. score-csvheader.rb を改良して、数学の平均点、英語の平均点の両方を求めて出力するプログラム avg-me.csv を作成せよ(答は何パターンもある)。

   実行例:

   ```
   % ./avg-me.rb score2h.csv
   数学の平均点は59.2点です。
   英語の平均点は72.5点です。
   ```

2. score2h.csv の英語の右隣りの列に、見出し "国語" を追加して、6名の行の右側にも国語の点数を(適当に)追加したCSVファイル score3h.csv を作成せよ。

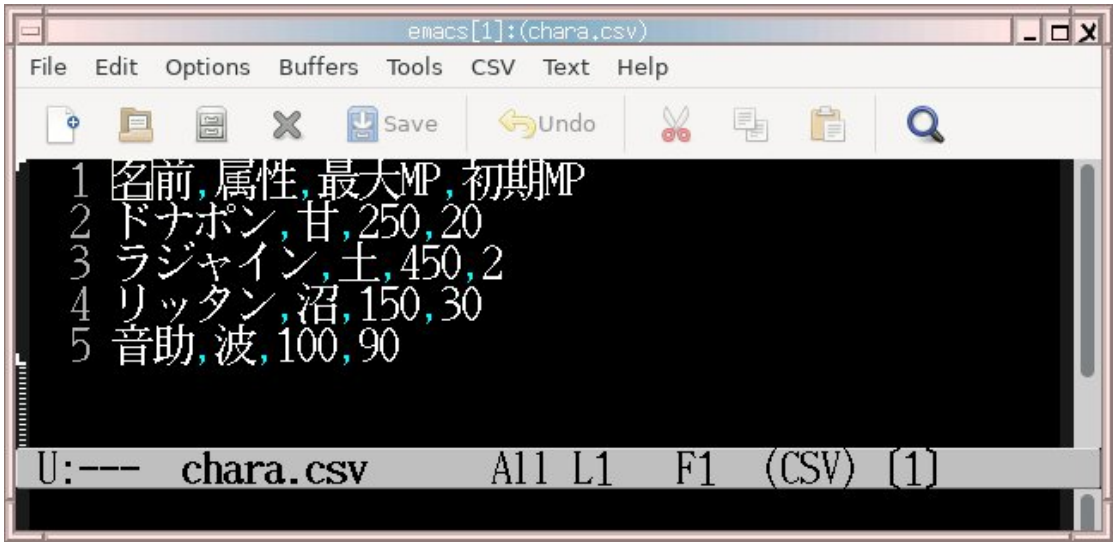
3. 上で作成したscore3h.csvを読み取り、誰がどの科目で何点を取ったかを出力するプログラム score3h.rb を作成せよ。 score-csvheader.rbを見本にするとよい。実行例を以下に示す。

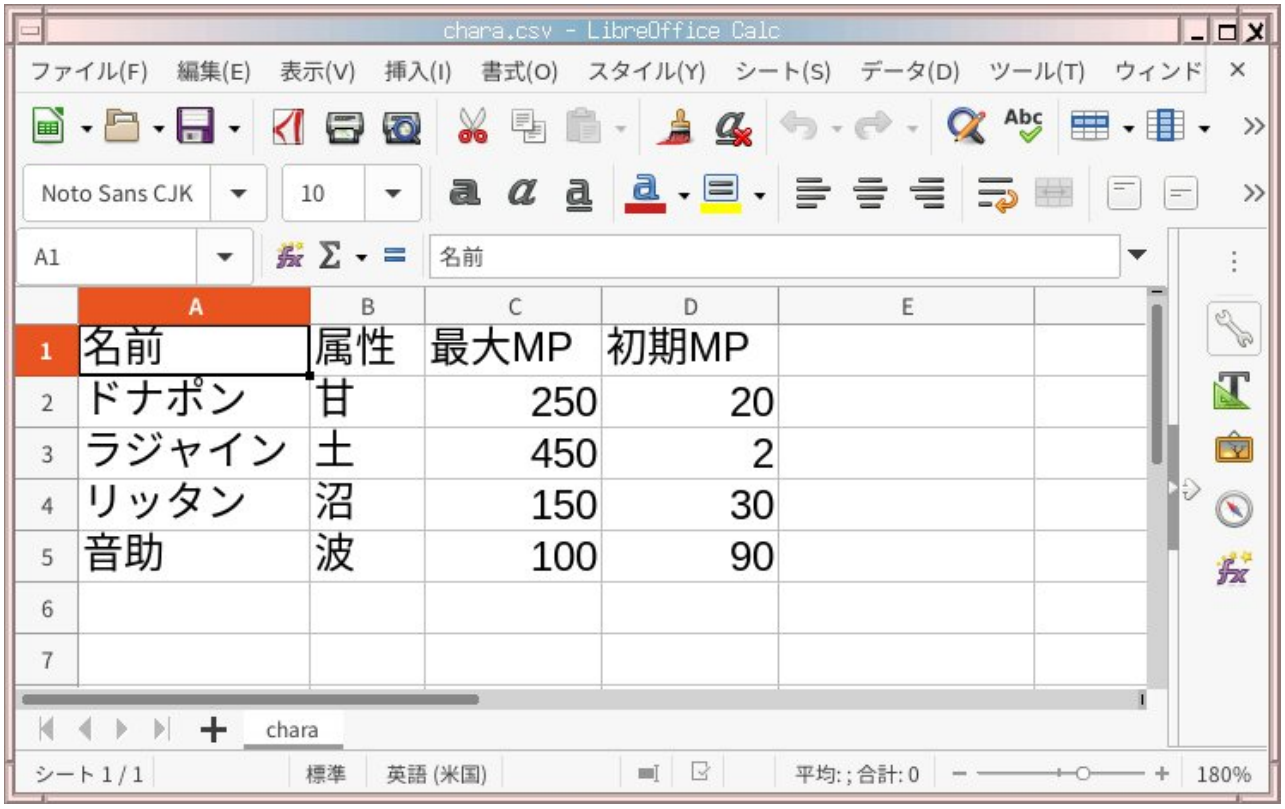
   ```
   % ./score3h.rb score3h.csv
   山田太郎さんの数学は50点、英語は70点、国語は60点です。
   中町太郎さんの数学は90点、英語は80点、国語は85点です。
   (以下略)
   ```

4. 上で作成したscore3h.rbとavg-me.rb を参考にして、score3h.csvにある数学、英語、国語の各平均点を計算して出力するプログラムavge-mej.rb を作成せよ。実行例を以下に示す。

   ```
   % ./avg-mej.rb score3h.csv
   数学の平均点は59.2点です。
   英語の平均点は72.5点です。
   国語の平均点は61.8点です。
   ```

5. キャラクター一覧にある値を保持するCSVファイルを作成せよ。ファイル名は chara.csvで保存せよ。入力完了後の画面例を示す。





	A	B	C	D	E
1	名前	属性	最大MP	初期MP	
2	ドナポン	甘	250	20	
3	ラジャイン	土	450	2	
4	リッタン	沼	150	30	
5	音助	波	100	90	
6					
7					

6. 上で作成したchara.csvを読み込んで、存在する「属性」値一覧を出力するプログラム chara-attr.rbを作成せよ。属性値を持つ配列はそのまま printf の %s で出力してよい(以下実行例)。

```
% ./chara-attr.rb
属性一覧: ["甘", "土", "沼", "波"]
```

7. ■【課題7a】

自分が趣味としてよく覚えているものについて、見出し項目を適切に整理した形でデータ化したCSVファイルを favorite.csv のファイル名で作成せよ。レコード(データ行数)は10件以上とし、項目(列数)は5カラム以上とする。

【データの例】
ゲームのキャラデータ、スポーツ選手一覧、記録一覧、歴代〇〇、花言葉など、地理的な数値一覧、……などなど

※注意※ *1行目は必ず項目名とし、2行目以降に実際のデータ* を書くこと。必ず csvcheck.rb を保存し、

```
% ./checkcsv.rb favorite.csv
```

を実行し、エラーがないことを確認せよ。

8. ■【課題7b】

上で作成したCSVファイル mydata.csv を読み込み、特定の項目のみを選んで出力するようなプログラム mydata.rb を作成せよ。

上記の例chara.csvで実行した場合の礼を示す。

```
% ./mydata.rb chara.csv
属性、最大MP、初期MP のどれを出しますか: 最大MP
ドナポンの最大MPは250です。
```

ラジャインの最大MPは450です。
リッタンの最大MPは150です。
音助の最大MPは100です。

9. ■【課題7c】

上記mydata.rbを改良し特定の項目に注目した絞り込み検索ができるプログラム mydata2.rb を作成せよ。

上記の問題で作成した chara.csv を読んで選別するプログラムを作成した例を示す: → キャラ選別作成プログラム

提出は任意とする。提出する場合はemailにて、宛先 *ruby-1-07@e.koeki-u.ac.jp* に送信する。

Subjectに

```
#7c ハッシュ
```

と記入し、本文の先頭に

氏名:　　公益太郎
学籍番号:　c125xxx
クラス:　　XX

を入れて、上記宛先に送ること。その後のレポート本体は

以下の順番で記載せよ。

一、作成したデータファイルとプログラム(本文中にコピー)

二、プログラムを動かした結果(結果画面のコピーペースト)

三、参考文献

四、感想

五、プログラム・データファイルの添付

の順で書くこと。参考文献を正しく書けていない場合は減点する(正規表現の回)参照。

◇参考:CSVファイルの作り方

CSVファイルの作成は以下のいずれかで行うとよい。

1. 表計算ソフト(Calcなど)で編集
2. Emacs(のcsv-mode)で編集

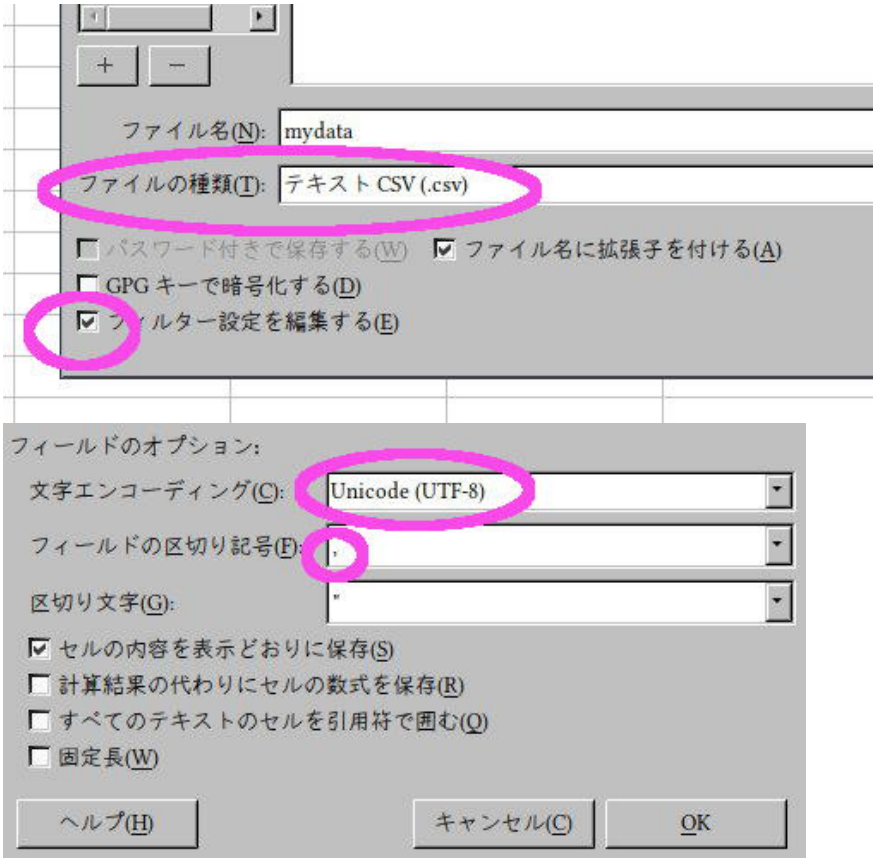
※注意 標準的なExcelでは期待通りの CSVファイルを作ることはできない(後述)。EmacsまたはLibreOffice/Calcで作ることを推奨する。

■Calcの場合

LibreOffice/Calcを起動し、データを入力したら

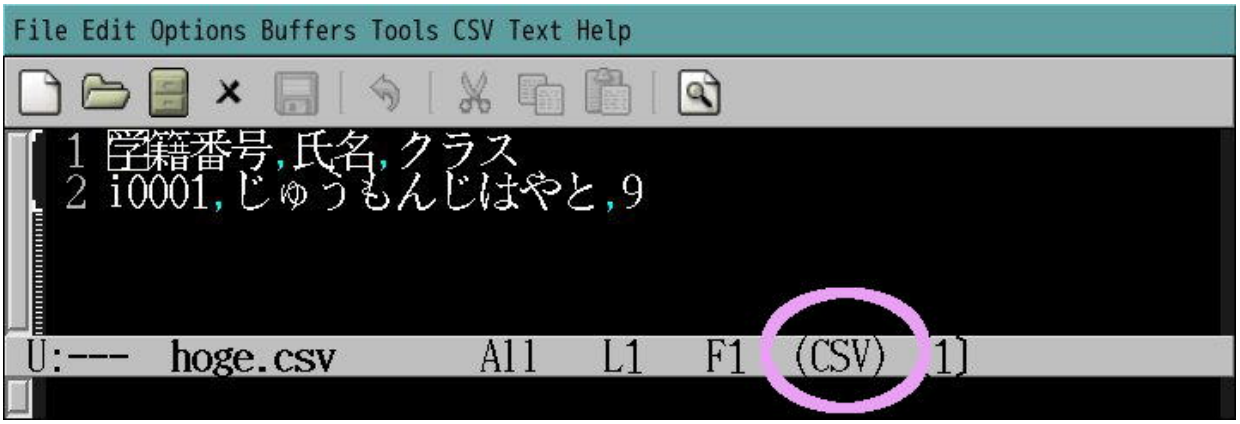
1. メニュー: ファイル → 名前をつけて保存
2. ファイル名入力右下保存種別の「ODF表計算ドキュメント」を「テキストCSV(.csv)」に変え、「フィル

ター設定を編集する」にチェックしてからファイル名をつけて保存 → 文字エンコーディングは「UTF-8」、フィールドの区切り記号はカンマ



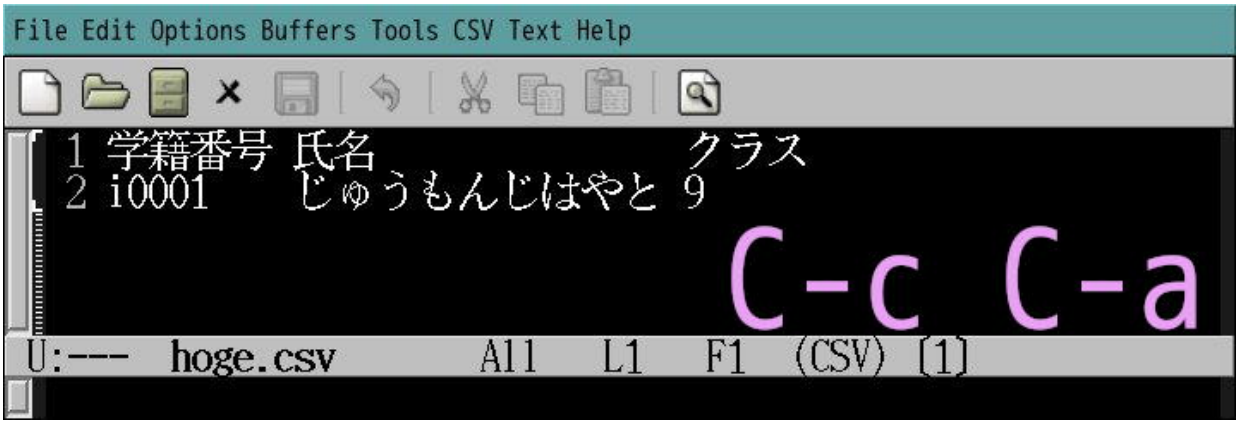
■Emacsの場合

CSVファイルを開いたときに以下のようにモードラインに「(CSV)」と出ていれば専用編集機能が使える。



（ついていない場合は相談）

この状態で C-c C-a をタイプするとCSVのフィールドごとに桁が揃って表示される。



項目を1つ入れたら Tab キーを押すと次の項目に進む。

第6講 正規表現

◆正規表現とは

電話帳や郵便番号簿から目的のデータを探すときはどうするだろう。おそらく先頭から1、2文字(イニシャル)を利用して、検索ボタンなり索引を使って探すだろう。誰が登録されているか覚えていればよいが、未知のものから探すのはたいへんである。また、データの先頭の文字を忘れていたり、「サトウ」なのか「サイトウ」なのかとか、「イカラシ」なのか「イガラシ」なのかなど、途中があやふやだったりすると検索機能では探せない。

なにかを検索するときに、そのデータに含まれている文字列のパターンを汎用的に指定する方法がある。そのうち、もっとも一般的であるものが *正規表現*である。

◇正規表現の例

たとえば、「サトウ」と「サイトウ」どちらかを探したい場合、正規表現では次のように検索パターンを書く。

```
サイ?トウ
```

こうすると、"サトウ" または "サイトウ" 両方にマッチするようになる。特別な文字 ? を指定すると、「その直前の文字があってもなくてもよい」という特別な意味になる。また、別の特別な文字 [ ] を使っても、特別な意味を持った指定ができる。たとえば、

```
イ[カガ]ラシ
```

とすると、"イカラシ" でも "イガラシ" でもマッチするようになる。特別な文字 [ ] の中に何文字か文字を列挙すると、「その中のどれかの文字が来ればよい」という指定になる。

> 上記の正規表現の例は特殊な文字の働きを分かりやすくためにもっとも単純に書いたものであり、実際にRubyプログラムに正規表現を書くときはそのままでは使えないので注意すること。

◇egrepコマンドによる正規表現の実験

実際に名簿のようなデータを作って、正規表現で検索する練習をしてみよう。最初に検索のデータとなるファイルを作っておこう。以下のような名簿データファイル、meibo.csv を作成しよう。

```
ナマエ,Name,氏名,居住地
サトウ ムネユキ,SATOH Muneyuki,さとう胸幸,青葉台
ナカマチ タロウ,NAKAMACHI Taro,中町太郎,酒田
イイモリ ハナコ,IIMORI Hanako,飯森花子,飯森山
サイトウ テツヤ,SAITOH Tetsuya,斎藤徹夜,白夜の国
ゲゲノ キタロウ,GEGENO Kitarou,下々野喜太郎,魔界村
ナニワノ タロー,NANIWANO Taro,浪花之太朗,探偵騎士王国
ヒミ ヨンタロウ,HIMI Yontarou,悲見四太郎,甲子園
ヤマタイ ヒミコ,YAMATAI Himiko,邪馬台氷見子,大和
イモリ テツコ,IMORI Tetsuko,井森徹子,井戸
```

これを利用して検索してみる。まずはローマ字でパターンを指定してみよう。 Unixには正規表現を使って

ファイルから特定の行を検索してくれる egrep というコマンドがある(Rubyとは別物)。 egrep コマンドは

```
% egrep "正規表現パターン" [ファイル群…]
```

のように起動し、0個以上の複数のファイルから「正規表現パターン」にマッチする行だけを選んで出力してくれる。ファイルを0個指定すると、 *標準入力*から読み込んだデータから検索する。

■SAITOH または SATOH の検索

SAITOH または SATOH のいずれかを含む行を全部選んでみよう。この場合、大文字の I があってもなくてもよいので、正規表現パターンは "SAI?TOH" となる。

```
% egrep "SAI?TOH" meibo.csv
```

正規表現のパターンを "" で括っているのは、途中にある? 記号をファイル名マッチだと思われないようにするためである。このようにコマンドラインで正規表現パターンを指定するときは、パターンを"" または '' で必ず括ること。

実行結果は次のようになるだろう。

```
% egrep "SAI?TOH" meibo.csv
サトウ ムネユキ,SATOH Muneyuki,さとう胸幸,青葉台
サイトウ テツヤ,SAITOH Tetsuya,斎藤徹夜,白夜の国
```

■大文字小文字の同一視

もう少し欲張って、楽になるように検索パターンを考えよう。パターン指定にいちいち大文字を打つのは面倒だし、もしかしたらデータファイルの方で Satoh と書いてしまう場合もあるので、大文字でも小文字でもどちらでもマッチするようにしてみよう。検索のときに大文字小文字を区別しないためには egrep コマンドに -i オプションを指定すればよい。

```
% egrep -i "sai?toh" meibo.csv
サトウ ムネユキ,SATOH Muneyuki,さとう胸幸,青葉台
サイトウ テツヤ,SAITOH Tetsuya,斎藤徹夜,白夜の国
```

■日本語(カタカナ)での検索

カタカナで検索してみよう。「サトウ」でも「サイトウ」でもよいように、「サイ?トウ」というパターンで検索してみよう。

> Terminalでの日本語入力は Shift-SPC でON/OFFする。カタカナへの変換は C-k で行なう。

```
% egrep -i "サイ?トウ" meibo.csv
サトウ ムネユキ,SATOH Muneyuki,さとう胸幸,青葉台
サイトウ テツヤ,SAITOH Tetsuya,斎藤徹夜,白夜の国
```

■その他の検索の練習

続いて、以下の条件にマッチする人を検索してみよう(パターンを "" で括るのを忘れずに)。

- 「タロウ」の名のつく人
- 「イモリ」という苗字の人
- 下の名前に「ヒミ」がつく人

> 上記の検索実験は期待した結果と少し違うものが出てくることを実感するのが目的である。

◇Rubyで使う正規表現

egrepと違い、Rubyでは正規表現を / / で括って指定する。たとえば、egrep の例で指定した正規表現 "sai?toh" はRubyでは、

```
/sai?toh/
```

と表記する。これに注意して、「指定したファイルから sai?toh というパターンを検索する」 Rubyプログラム saito.rb を作成すると次のようになる(後述の説明のため途中から行番号を付した)。

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

1: meibo = CSV.read(ARGV[0], headers:true)
2: meibo.each do |row|
3:   if /sai?toh/i =~ row["Name"]
4:     print row.to_s
5:   end
6: end
```

まずは、実際に実行してみよう。egrep コマンドと同様検索したいファイルを引数として与えよう(検索パターンはRubyプログラム内に記述してあるのでコマンドラインからは指定しなくてよい)。

```
% chmod +x saito.rb
% ./saito.rb meibo.csv
サトウ ムネユキ,SATOH Muneyuki,さとう胸幸,青葉台
サイトウ テツヤ,SAITOH Tetsuya,斎藤徹夜,白夜の国
```

プログラム(本体部分)の内容を一行ずつ見ていこう。

1. meibo = CSV.read(ARGV[0], headers:true)

   コマンドラインで指定した meibo.csv がARGV[0]に代入され、それがヘッダ付き CSV として読み込まれる(headers:true)。データ全体がmeibo変数に代入される。

2. meibo.each do |row|

   meibo 変数にあるレコードを1つ1つ取り出して繰り返す。 1レコード(1行にある全フィールドの集合)はブロック変数 row に代入されてレコード回数分だけ繰り返される。つまり、一回目には row 変数に以下の値が入る(CSVrow形式)。

   ```
   => #<CSV::Row "ナマエ":"サトウ ムネユキ" "Name":"SATOH Muneyuki" "氏名":"さとう胸幸"
   "居住地":"青葉台">
   ```

3. if /sai?toh/i =~ row["Name"]

   このifで、各レコード中のNameフィールドを取り出し、それが正規表現にマッチしているかの判定を行なう。「正規表現にマッチしているか?」を意味する条件式は

   ```
   正規表現 =~ 文字列
   ```

   のように書く。マッチしている場合は、実際にマッチした部分が *文字列* の何文字目かを表す整数を返す。マッチしない場合はnilを返し、条件不成立となる。

   /sai?toh/i は、Rubyによる正規表現の指定である。後の / の直後にある i は「大文字小文字を区別し

ない」という意味を付加する働きを持つ。つまり、 egrep コマンドの -i オプションは、Rubyでは、正規表現の直後に i を付けることに相当する。

正規表現の後ろの =~ は、「正規表現がマッチすれば…」という比較演算子である。通常1行ずつ入力した文字列をこの右辺に書く。ちなみに「マッチしなければ」を意味する演算子は !~ である。

=~ の右辺に指定した「文字列」が正規表現の比較対象となる文字列である。

4. print row.to_s

   row 変数には複数のフィールドが入っている。それらを元の(CSVファイルに書き込むような)文字列に to_s メソッドで変換し、printで出力する。もちろん、一つ上の行にif があるので、正規表現にマッチした場合だけ該当行が出力される。

5. end

   ifに対応するend

6. end

   eachに対応するend

◇問題

上記 saito.rb を元に以下のように改良したものを作成せよ。

1. 検索パターンを「サイ?トウ」に変えて、検索対象フィールドをナマエ(第1項目)として検索した結果を示す saito-katakana.rb
2. 検索対象を1レコード全体(row.to_s)として検索する saito-line.rb

解答例(解いてから見ること): saito-katakana.rb, saito-line.rb

◆正規表現の特殊文字

正規表現では、"あ" という文字を指定すると、「あ」という文字そのものにマッチする。"AB"という文字列を指定すると、「AB」という文字列そのものにマッチする。「AB」が*行の途中にあっても構わない*。アルファベットや日本語など普通の文字や文字列を指定すると、それ自身とマッチする。

通常の文字以外に、特別な記号を使うことで、文字そのもの以外にマッチする表現とすることができる。特別な意味を持つ記号について解説する。検索のときに特別な意味を持つ文字のことを *メタキャラクタ* という。

◇代表的なメタキャラクタ

Rubyで扱えるメタキャラクタのうち、代表的なものを示す。

. (ピリオド)

記号 . は、*任意の1文字にマッチ*する。たとえば、正規表現 ai.awa は、"ai" のうしろに何でもよいので何か1文字が来て、そのあとに "awa" が続くようなもの全てにマッチする。たとえば、以下のようになる。

正規表現	照合する文字列	マッチするか?
/ai.awa/	aikawa	○
	aizawa san	○
	maisawa	○
	aimoto	×
	ai awa	○

照合する文字列のうち実際にマッチしている部分には下線が施してある(以下の例も同様)。

ピリオド自身を探したいときは\.とする。バックスラッシュには特殊文字の働きを打ち消す意味がある。以後の [ ] ? * + ^ $ | ( ) についても同様で、それら自身を探したいときにはそれぞれ、\[\] \? * \+ \^ \$ \| \(\) というパターンを指定する。バックスラッシュ自身は\\で探す。

[ ](大括弧)

大括弧 *[ ]* は、その中に列挙した文字のどれか1文字にマッチする。たとえば、正規表現 [abc] は、文字 a, b, c のどれかにマッチする。さらに、正規表現 ai[sz]awa でいくつかの文字列と照合を行なったときの結果は次のようになる。

正規表現	照合する文字列	マッチするか?
/ai[sz]awa/	aikawa	×
	aizawa san	○
	maisawa	○
	aimoto	×
	ai awa	×

大括弧の中で -(ハイフン) を使うと文字の範囲を指定できる。たとえば正規表現 [0-9] は、文字 0 から文字 9 の範囲の全ての文字、つまり 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 のいずれにもマッチする。もし、マッチする文字としてハイフン自体を指定したいときは大括弧の先頭に記述する。たとえば、正規表現 [-a-z0-9] は、「ハイフン、または aからz のどれか、または0から9のどれか1文字」にマッチする。

大括弧の中の先頭に ^ (山記号; キャレット) を指定すると、そのあとに列挙した文字*以外* であるものにマッチする。たとえば、正規表現 [^a-z] は、aからz 以外の文字であればどれでもマッチする。

?

記号 ? は、*直前のパターンが0回か1回出現する* という意味を付加する。直前のパターンは文字でも記号でもよい。たとえば、正規表現 sai?toh は、? の直前が i なので、iが0個でも1個でもよいことを意味する。さらにたとえば、正規表現 sa[ei]?ki は、直前のパターンが [ei] なので、「eかi」の文字が0個でも1個でもよいことを意味する。つまり、sa[ei]?kiにマッチするのは、 saeki, saiki, saki のどれかに限ることになる。

直前のパターンが0文字の場合にもマッチするので、 ?を正規表現の最後に付けたパターンで検索した結果は付けないものと同じ結果となる。
例: sai? は sa で検索するのと同じ結果になる(saの次にiがあった場合にマッチする範囲のみが異なる)。

***

記号 * は、*直前のパターンが0回以上出現する* という意味を付加する。直前のパターンは文字でも記号でもよい。たとえば、正規表現 sai*toh は、* の直前が i なので、iが0個以上なら何個でもよいことを意味する。つまり、 satohでもsaitohでもsaiitohでもsaiiiiiitohでもマッチする。さらにたとえば、正規表現 sa[ei]*ki は、直前のパターンが [ei] なので、「eかi」の文字が0個以上なら何個でもよいことを意味する。

正規表現 .* は、直前のパターンが . (任意の文字)なので、何でもよい文字列が何文字続いてもよいことを意味する。つまり、どんな文字列でもマッチする(よく使う)。典型的には 東北.*大学 のように2つのパターンの間に .* を挟み、「『東北』と『大学』の間はなんでもいいや、なくてもいいや」という検索をしたいときに使う。

直前のパターンが0文字の場合にもマッチするので、 *を正規表現の最後に付けたパターンで検索した結果は付けないものと同じ結果となる。
例: sai* は sa で検索するのと同じ結果になる(saの次にiがあった場合にマッチする範囲のみが異なる)。

+

記号 + は、*直前のパターンが1回以上出現する* という意味を付加する。回数が「1回以上」という点を除いて記号 * と全く同じと考えてよい。

直前のパターンが1文字の場合にもマッチするので、 *+が正規表現の最後になることはない* (+を最後に書くのなら、+を省略しても同じ行がマッチする)。

^

記号 ^ を正規表現の先頭に指定した場合のみ、*文字列の先頭にマッチ*する。たとえば、正規表現 ^sato は、先頭が sato から始まるもののみにマッチする。文字列の途中に sato が含まれてもマッチしない。つまり、以下のようになる。

正規表現	照合する文字列	マッチするか?
/^sato/	sato san	○
	satoh desu	○
	I am sato	×
	Sato!	×

$

記号 $ を正規表現の末尾に指定した場合のみ、 *文字列の末尾にマッチ*する。たとえば、正規表現 sato$ は、sato で終わる文字列のみにマッチする。

|

記号 |(縦棒; パイプ) は、「または」の意味で、 | の左右のパターンどちらかにマッチすればよいことを意味する。たとえば、正規表現 SAITO|IIMORI は、 SAITO または IIMORI のどちらかにマッチする。

( )

丸括弧 ( ) は、長い正規表現の一部分だけを括るときに利用する。たとえば、正規表現 a(wa|ma|yanokou)ji は、| で選択する単語の範囲を限定しているので、結果として awaji, amaji, ayanokouji のどれかを含むものにのみマッチする。

また、正規表現 a(re)+ は、+ の直前のパターンを括弧で括り (re) としているので、 are, arere, arererere, arererere, ... などにマッチする。もし ( ) で括らずに、are+ とすると、 + の直前が e だけになるので、are, aree, areee, ... にマッチすることになる。

このように、正規表現の一部を丸括弧で括って一つのかたまりにすることを、*グルーピング*という。元の文字列のうち、グルーピングした部分にマッチした部分文字列はあとで抽出することができる。

\数字

バックスラッシュ \ の後ろに数字 N を続けると、元の正規表現のN番目の括弧にマッチした文字列そのものに置き換えられる。たとえば、正規表現 (bye|ciao) \1 は、最初の括弧で "bye" または "ciao" に

マッチする。次の \1 は、1番目の括弧でマッチした文字列そのものに置き換えられるので、最初にマッチしたものが "bye" なら "bye" 、"ciao"なら"ciao"にマッチする。つまり、 "bye bye" または "ciao ciao" のみにマッチする。もし、 \1 を使わずに、正規表現を（bye|ciao）(bye|ciao）とすると、2個目の括弧も "bye" "ciao" どちらでもよくなるので、"bye bye", "bye ciao", "ciao bye", "ciao ciao" どれにでもマッチすることになる。\数字 を使うと、1行の中にある文字列が繰り返して現れるパターンを記述することができる。

\b

正規表現中の \b は、「単語の境界」にマッチする。たとえば、正規表現 station で検索すると "stationery" や "attestation" がマッチするが、正規表現 \bstation\b は、"s" の前と "n" の後ろが単語の境目になっているものにしかマッチしないため、"…… station ……" のように単語として独立した "station" を含むものにしかマッチしない。

\s

正規表現中の \s は、あらゆる空白文字にマッチする。スペースやTAB文字、改行文字などにマッチする。

\S

正規表現中の \S は 空白文字以外の全ての文字にマッチする。ちょうど\sの逆である。

\d

正規表現中の \d は、0から9(半角)のどれかにマッチする。たとえば、正規表現 \d+ は、0から9が1回以上くり返したものにマッチするので、数字と思われるもの全てにマッチする。

\D

\D は、数字以外の全ての文字にマッチする。 \dの逆である。

\w

\w は、単語を構成する文字のどれか、つまり英数字 [0-9A-Za-z_] や日本語文字のどれかにマッチする。ただし、日本語文字にマッチするのは次節の正規表現オプションで日本語検索を有効にした場合に限る。

\W

\W は、\w の逆で単語構成文字*以外*の1字にマッチする。

◇正規表現オプション

/ /で括った正規表現の直後に文字を追加すると正規表現の検索方法を変えることができる。たとえば、デフォルトでは英字の大文字と小文字を区別した検索を行なうが、正規表現指定の直後にiを付けて、 /abcde/i とすると、大文字と小文字を区別せずに検索するようになる。これを*正規表現オプション*という。このようなオプションのうち覚えておくべきものを紹介する。

- *i* - 大文字小文字を区別しない
- *m* - 文字列中に改行が含まれていても . でマッチするようになる

◇正規表現の指定方法

最初に紹介したRubyでの正規表現指定方法は、パターンを / / で括るというものだった。それ以外にもいくつか方法がある。プログラム中で正規表現を指定するには以下のどれかを使う。

- /パターン/

  もっとも簡単な指定方法。既に利用した。

- %r,パターン,

  %rの直後の文字でパターンを挟む。上記の例では ,(カンマ) だが、括弧で括ったり、括弧以外の任意の文字で挟むことができる。%r(パターン)でも %r|パターン|でも%r!パターン! でもよい。ただし、挟むための右側の文字がパターン中に現れてはいけない。本講では使わないが、すこし複雑な正規表現を指定するときにはどうしても必要。たとえば、/ をたくさん含む文字列を検索したいときはパターンの区切り文字に / 以外を利用できるこの表記が便利で、日付けらしき文字列を探すときに

  ```
  %r:\d\d\d\d/\d\d?/\d\d?:
  ```

  などとすると「4桁の数値/1、2桁の数値/1、2桁の数値」にマッチするパターンが書ける。%rを使わず / / で括ると、

  ```
  /\d\d\d\d\/\d\d?\/\d\d?/
  ```

  となり、スラッシュの境目が分かりづらい。

- Regexp.new(パターン文字列)

  文字列を正規表現に変換する。プログラム中で

  ```
  pattern = "sai?toh"
  ```

  という値がありこれを正規表現 /sai?toh/ に変換したいときは、

  ```
  regexp = Regexp.new(pattern)
  ```

  とする。本講で使う。プログラム作成時ではなく、起動したときに検索パターンを読み込みそれ正規表現に変換するのであれば、 Regexp.newが必要。

  大文字小文字を区別して欲しくない場合は、第2引数に trueを指定する。

  ```
  regexp = Regexp.new(pattern, true)
  ```

  とすると /sai?toh/i と同じことになる。

  > true の代わりに
  >
  > - Regexp::IGNORECASE（大文字小文字を区別しない）
  > - Regexp::MULTILINE（. が改行にマッチする）
  > - Regexp::EXTENDED（拡張）
  >
  > を指定することもできるが本講では扱わない。

◆正規表現検索の練習

最初に作成した meibo.csv のデータ

```
ナマエ,Name,氏名,居住地
サトウ ムネユキ,SATOH Muneyuki,さとう胸幸,青葉台
```

```
ナカマチ タロウ,NAKAMACHI Taro,中町太郎,酒田
イイモリ ハナコ,IIMORI Hanako,飯森花子,飯森山
サイトウ テツヤ,SAITOH Tetsuya,斎藤徹夜,白夜の国
ゲゲノ キタロウ,GEGENO Kitarou,下々野喜太郎,魔界村
ナニワノ タロー,NANIWANO Taro,浪花之太朗,探偵騎士王国
ヒミ ヨンタロウ,HIMI Yontarou,悲見四太郎,甲子園
ヤマタイ ヒミコ,YAMATAI Himiko,邪馬台氷見子,大和
イモリ テツコ,IMORI Tetsuko,井森徹子,井戸
```

を利用して、いくつか検索練習してみよう。

利用するプログラムは、さきほどの saito.rb だと決まったパターンしか検索できないので以下のように変更しよう。

meibosearch.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'csv'
meibo = CSV.read(ARGV[0], headers:true) # ./meibosearch.rb meibo.csv と起動する

print "検索パターン: "
pattern = STDIN.gets.chomp!
regexp = Regexp.new(pattern, nil)

meibo.each do |row|
  line = row.to_s
  if regexp =~ line
    print line
  end
end
```

プログラム中の、STDIN は標準入力を意味する。 Rubyプログラム起動時にファイルを引数に与えると gets を利用したときにそのファイルからデータを読む。しかし、STDIN.gets とすると、ファイルを引数に与えたとしても標準入力から1行読む。これを、chomp!で改行を切り取り、さらに Regexp.new メソッドに渡して正規表現にしている。

■ひらがなを含む行の検索

ひらがなは正規表現［ぁ-ん］で表現できる。

```
% ./meibosearch.rb meibo.csv
検索パターン: [ぁ-ん]
サトウ ムネユキ,SATOH Muneyuki,さとう胸幸,青葉台
サイトウ テツヤ,SAITOH Tetsuya,斎藤徹夜,白夜の国
```

ちいさい「ぁ」は la (えるえー)で入力する。

■「サ」で始まる人全てを検索

行の先頭は ^ で指定する。^サ とすると行の先頭が「サ」で始まるもののみにマッチする。

```
% ./meibosearch.rb meibo.csv
検索パターン: ^サ
サトウ ムネユキ,SATOH Muneyuki,さとう胸幸,青葉台
サイトウ テツヤ,SAITOH Tetsuya,斎藤徹夜,白夜の国
```

■同じ文字(空白以外)が繰り返す行の検索

グルーピングを使うと繰り返しを記述できる。空白以外の文字は \S である。これを括弧でグルーピングし

て、\数字 で参照すればよい。正規表現は (\S)\1 となる。

```
% ./meibosearch.rb meibo.csv
検索パターン: (\S)\1
イイモリ ハナコ,IIMORI Hanako,飯森花子,飯森山
ゲゲノ キタロウ,GEGENO Kitarou,下々野喜太郎,魔界村
```

「イ」と「ゲ」が2字連続で出てくるのでこれにマッチする。

■下の名前に「ヒミ」が付く人を検索

苗字は行頭なので ^ 記号で絞り込みできるが、下の名前は行の途中なので絞り込めない。苗字と下の名前の区切りにスペースがあることを利用し、

行頭 + *苗字* + *空白* + *下の名前*

というパターンを記述する。苗字は単語だったら何でも構わないので「1文字以上の非空白文字」を意味する \S+ でマッチングさせる。下の名前は「ヒミ」が最初に来ても最後に来ても途中に来ても、「ヒミ」全部でもよいようなパターンを指定するので \S*ヒミ\S*\b とする。

```
% ./meibosearch.rb meibo.csv
検索パターン: ^\S+\s+\S*ヒミ\S*\b
ヤマタイ ヒミコ,YAMATAI Himiko,邪馬台氷見子,大和
```

ちなみに「ヒミを含む」という部分のパターンを .*ヒミ.* などとしてしまうと、1つめの.* の部分で空白を越えてマッチする可能性があるため、マッチした「ヒミ」が必ずしも下の名前の位置にあるとは限らなくなってしまう。

■名前が「たろう」の人

名簿を見ると「タロウ」と「タロー」がいるので、「タロ」の次が「ウ」でも「ー」でもマッチするよう タロ[ウー] とする。ただし名前の途中にそれらが含まれるものは除外したいので単語境界を意味する\bで絞り込みを行なう。

```
% ./meibosearch.rb meibo.csv
検索パターン: \bタロ[ウー]\b
ナカマチ タロウ,NAKAMACHI Taro,中町太郎,酒田
ナニワノ タロー,NANIWANO Taro,浪花之太朗,探偵騎士王国
```

練習問題 5

1. 以下のようなテキストがある(mpg123el.txt)。

```
1:The mpg123.el Emacs-Lisp program is a front-end to
2:mpg123 audio player and OggVorbis audio player. You
3:can select and play an music files from the list in
4:your Emacs's buffer with familiar interface.
```

以下の正規表現で照合を行なうときにマッチする行を全て行番号で答えよ。なお、テキストの左端の数字とコロンは行番号を表すためのものでテキストには含まれない。

イ、/./

ロ、/e./

ハ、/the/

ニ、/the/i

ホ、/[A-Z]/

ヘ、/^[A-Z]/

ト、/[^A-Z]/

チ、/[a-z][A-Z]/

リ、/\d/

ヌ、/\d*/

ル、/\d+/

ヲ、/\d*[A-Z]/

ワ、/\d.*[A-Z]/

カ、/(.)\1/

2. まず、以下のプログラムを reex.rb という名前で保存し実行できるようにせよ。

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

print "検索パターン: "
pattern = STDIN.gets.chomp!

if %r,^/.*/[ei]?$, =~ pattern
  regexp = eval pattern
else
  regexp = Regexp.new(pattern, nil)
end

while line=gets
  if regexp =~ line
    printf("%2d:%s%c[33;2m%s%c[m%s", $., $`, 27, $&, 27, $')
  end
end
```

「./reex.rb mpg123el.txt」のように起動して問題1の各設問の正規表現パターンを入れ、実際にマッチする部分を確認せよ。

第7講 計算機の内部表現

◆2進数と16進数

Rubyでのプログラミングではあまり計算機の中の都合を意識しなくてよいようになっている。しかし、凝ったプログラムを作ろうとしたり、 Ruby以外で少し複雑な計算などを行なわせようとするとき、あるいはとくにコンピュータグラフィックスなどの基礎知識を習得したい場合には、計算機の内部構造、とくに2進数の性質を知らないとスムーズに理解できないことがある。しっかりと覚えておきたい。

◇2進数

計算機の内部では、数値も文字も画像も全て 1 と 0 の*2進数* の状態で保存している。人間にとって分かりやすい10進数でなく2進数が使われるのは、計算機が何かを記憶するための格納場所である*メモリ*に書き込むためには電気的にONかOFF（あるいは磁気的にNかS)かの形で書くことになるからである。これを何個も組み合わせて一つのデータ格納領域とする。たとえば、ONとOFFの箱を8個用意して、次のようなデータを入れる。

OFF	ON	OFF	OFF	OFF	OFF	OFF	ON

これは模式的な絵であるが、計算機のメモリの中では、電気がONになっていたり、磁石がN極を向いている状態を想像するのが実状に近い。このOFFとONの状態を2進数にしよう。

0	1	0	0	0	0	0	1

2進数の 01000001 になる。2進数を10進数に直すときは、各桁に以下のような倍率を掛けたものを足して行けばよい。

128倍	64倍	32倍	16倍	8倍	4倍	2倍	1倍
0	1	0	0	0	0	0	1

これは、64×1 + 1×1 = 65 となる。

◇16進数

いくら計算機が内部で2進数を使うとはいえ、2進数のまま数値を書いたのでは長くて読みづらい。かといって、10進数に直すのは計算が面倒である。そのため、2進数4桁をまとめて16進数にして数値を表記することが多い。

> たとえば、10進数でもまとめて表記することがある。長さの尺度が「メートル」のまま距離を表現したら大変なので、10進数3桁(10の3乗=1000)ごとにまとめて「キロ」メートルで表現することがある。逆に、10の-3乗メートルを「ミリ」として表記したりする。また、金額を表記するときも3桁毎にまとめて 2,500,000円 のように10の3乗ごとにまとめてあたかも1000進法のようにして表すことがある。このように何桁かをまとめて単位を付けたりすることは日常的にも身近なことである。

2進数4桁をまとめると、2**4=16個の数を表すことができる。

2進数	16進数	10進数
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	a	10
1011	b	11
1100	c	12
1101	d	13
1110	e	14
1111	f	15

ONとOFFの箱で出てきた、

OFF	ON	OFF	OFF	OFF	OFF	OFF	ON

つまり

0	1	0	0	0	0	0	1

は、2進数で 01000001 だが、これを16進数で表すと4桁毎に変換すればよいので

- 0100 → 4
- 0001 → 1

となる。2進数 01000001 は、16進数で41である。ちなみに41は文字コードでいえば、アルファベット大文字Aに該当する。つまり、Aという文字を計算機の内部に記憶させたいときは

0	1	0	0	0	0	0	1

のようにデータが格納されることになる。

◇16進数と10進数の変換

■16進数を10進数へ

16進数を10進数に変換するには、各桁に16の倍数を掛けたものを足す。たとえば、16進数 56ce を10進数に直してみよう。

4096倍 (16**3)	256倍 (16**2)	16倍	1倍
5	6	c	e

cの桁は10進数で12を意味することと、eの桁は10進数で14を意味することに注意して計算すると、

```
4096*5 + 256*6 + 16*12 + 14 = 22222
```

となる。

■10進数を16進数へ

10進数を16進数に変換するには、

1. 10進数を直接16進数に変換する
2. 10進数を2進数に変換してから16進数に変換する

の2とおりの方法がある。

1. 10進数を直接16進数に変換する

   元の10進数を割れなくなるまで16で割り、割ったときの余りを逆に拾って行けばよい。たとえば、10進数64206を16進数に直してみよう。

```
16) 64206    (余り)
16)  4012 …… 14 → e
16)   250 …… 12 → c
16)    15 …… 10 → a
16)     0 …… 15 → f   ↑下から上に読む
```

   割り算の余りを下から順に読んだ、0xface が16進の値となる。

2. 10進数を2進数に変換してから16進数に変換する

   元の10進数を2進数に変換する。

```
2)  64206  (余り)
2)  32103 …… 0
2)  16051 …… 1
2)   8025 …… 1
2)   4012 …… 1
2)   2006 …… 0
2)   1003 …… 0
2)    501 …… 1
2)    250 …… 1
2)    125 …… 0
2)     62 …… 1
2)     31 …… 0
2)     15 …… 1
2)      7 …… 1
2)      3 …… 1
2)      1 …… 1
2)      0 …… 1    ↑ 下から上に読む
```

余りを下から上に読むと、

```
1111101011001110
```

となるが、これを1の位(右側)から4桁毎に区切る。

```
                ← 右から4桁毎に区切る
1111,1010,1100,1110
```

区切った2進数を16進数に直していく。

```
1111,1010,1100,1110
 ↑    ↑    ↑    ↑
 f    a    c    e   (16進数)
```

◇Rubyプログラムでの扱い

Rubyプログラムでは、プログラム中に書く数値の先頭に付ける記号(接頭辞)で何進法かを指示できる。

接頭辞	意味
なし	10進数
0b	2進数
0	8進数
0x	16進数

0b1010 は、2進数の1010（10進数でいえば10を)表すことになり、0x10f0 は、16進数の 10F0（10進数でいえば4336)を表すことになる。

プログラム中に、0b1010と書こうが 10 と書こうが、0xa と書こうが、一度読み込まれれば計算機の内部ではどれも10進数でいうところの10として扱われ、どれも全く同じである。これを出力するときに何進法にするかは printf の%つきフォーマットで選択することができる。

以下に、printf の%フォーマットで数値を何進法で出すかの意味を示す。

%フォーマット	何進法で出力されるか
%b	2進法
%o	8進法
%d	10進法
%x	16進法

これを利用すると、16進数を10進数に直したり、その逆をしたりすることをRubyプログラムに任せることができる。

```
printf("16進の%xは10進で%dです\n", 0x555, 0x555)
↓
16進の555は10進で1365です
```

```
printf("10進の%dは2進で%bです\n", 2002, 2002)
↓
10進の2002は2進で11111010010です
```

◆文字コード

計算機で文字を扱う場合、内部では文字に付けた番号を元に管理している。計算機で利用できる文字には全て番号がついている。この番号のことを *文字コード*という。

◇ASCIIコード

abcdefg... ABCD..., 0123..., !@#$ () などの、日本語入力OFFのまま入れられる文字や記号には全て*ASCIIコード*(American Standard Code for Information Interchange)が割り当てられている。

ASCIIコード表を以下に示す。

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

この表は16進数での文字コード表記となっている。最左列の縦に並ぶ0-7が 16進数の10の位で、最上行の横に並ぶ0-Fが16進数の1の位である。たとえば、アルファベットの大文字Aは、「4の行」の「1の列」にあるので、16進数の「41」が文字コードとなる。以下、16進数の数字を表すときは、先頭に 0x を付けて表す。「16進数41」==「0x41」

表中、文字コード 0x00 - 0x1Fは、*制御文字*といい主に端末を制御するための文字コードが並んでいる。代表的なものとしては、

- 0x00 NUL - 空文字（文字列の終端を表すことが多い）
- 0x07 BEL - 端末のベルを鳴らす
- 0x08 BS - バックスペース(BackSpace)
- 0x09 HT - 水平タブ(Horizontal Tab)
- 0x0A LF - 改行(Line Feed)
- 0x0D CR - 復帰(Carriage Return; リターン)
- 0x1B ESC - エスケープ(Escape)

などがある。これらの制御文字をキーボードから入力するときは、制御文字の文字コードに0x40を足した文字を、Ctrlキーを押しながらタイプする。つまり、0x09のHTは、0x40を足した0x49がI なので、C-i が対応するキーとなる。また、慣習的に 0x00のNULは C-SPC(Ctrlを押しながらスペース)に対応させることが多い。まとめると

0x00 NUL
  C-@（またはC-SPC）
0x01 SOH - 0x1A SUB
  C-a - C-z
0x1B ESC - 0x1E RS
  C-[- C-^
0x1F US
  C-_ または C-/

という風にキー入力できる。ただし、現在ではこれらの制御文字を直接画面に出すことはほとんどないので、C-英字 などのキー入力にはカーソル移動したり、画面を消したりするなどの「機能」が割り当てられていることが多い。

実際に、これらの制御文字そのものを入力したいときには

- シェル上(Terminal)

  C-vを押してから制御文字

- Emacs上

  C-qを押してから制御文字

をタイプする。ためしに、BELコードを使って、端末(Terminal)のベルを鳴らしてみよう。Terminalで次のようにタイプする。

```
% echo "^G"
  (^Gの部分は C-v C-g とタイプする)
```

Terminalで順に e c h o スペース " C-v C-g " とタイプすると、画面に echo "^G" と出ているはずなので、それを確認したら最後に [Return] を押す。

ピッ

と音が出るはずである。ここで利用した echo というコマンドは、後続する引数をそのまま出力するものである(シェルの内部コマンド)。

また、0x1B のESC コードも端末制御をするのによく用いられる。ESC コードに続いて一連の決まった文字を端末に出力すると、端末の文字の種類を変えたりすることができる。有名なものに、文字色を変える エスケープシーケンスがある。

- ESC [30m → 文字色を黒に
- ESC [31m → 文字色を赤に
- ESC [32m → 文字色を緑に
- ESC [33m → 文字色を黄色に
- ESC [34m → 文字色を青に
- ESC [35m → 文字色をマジェンタに
- ESC [36m → 文字色をシアンに
- ESC [37m → 文字色を白に
- ESC [m → 文字の属性を元に戻す

ためしに、画面に黄色の文字を出してみよう。ESC コードはシェル上とRubyプログラムでは \e で表せる。C-v ESC とタイプしよう。

```
% echo "Hello \e[33mworld"
Hello world
```

のように色の変わったメッセージが表れる。

◇Rubyプログラムでの文字コードの扱い

他のプログラミング言語と比較すると、Rubyでは文字コードのことを意識しなくてもプログラミングできる範囲が広くなっている。上記のように画面に出すメッセージの色を変えたいようなときは文字コードを意識する必要がある。

ある文字コードのものを画面に出力したい場合は、printf のフォーマット文字列のところに %c を指定する。たとえば、

```
printf("%c", 0x07)
```

とすると、文字コード 0x07 つまりBELが出力され、端末のベルが鳴る。やってみよう。

```
% irb
irb(main):001:0> printf("%c", 0x07)
```

文字コードを実際の文字に変換するのが %c であるならばその逆は何だろう。ある文字の文字コードを得るには ?文字.ord を使う。プログラム中に ?a.ord と書けば、それは 文字 'a' の文字コード(つまり0x61)と書いたのと同じことになる。?9.ord と書けば '9' の文字コード(つまり0x39)と書いたのと同じことになる。次のプログラムは、ASCIIコード表のうち、'!'(0x21) - 'z'(0x7A) の範囲の文字全てを出力するものである。

ascii.rb

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

char = ?!.ord                    # ! の文字コード(==0x21)

while char <= ?Z.ord
  printf("ascii-code %3d(0x%02x): %c\n", char, char, char)
  char += 1
end
```

> 上記のプログラム ascii.rb を実際に実行してみること

◇文字列と文字コード

Rubyプログラム中の文字列は、あたかも文字コードの数値が並んで入っている配列のように考えることができる。たとえば、"Hello" という文字列を考えよう。各文字の文字コードを調べると、Hは0x48、eは0x65、lは0x6c、o は0x6fとなる。これらの数値が並んで配列に入ったものをイメージしよう。

0	1	2	3	4
H	e	l	l	o
0x48	0x65	0x6c	0x6c	0x6f

文字列 "Hello" の最初の部屋(0号室)を取り出すときは、配列のときと同様大括弧[ ]を利用する。

```
printf("%c\n", "Hello"[0])
↓
H
printf("%x\n", "Hello"[0].ord)     (.ordにより文字コードが得られる)
↓
48   (これは16進数。以下も同じ)
```

```
printf("%x\n", "Hello"[1].ord)
↓
65
```

(以下同様)

これを文字列の先頭から、末尾まで繰り返すプログラムは以下のようになる。

hellocode.rb

```
#!/usr/local/bin/ruby
```

```
# -*- coding: utf-8 -*-

hello = "Hello"
i = 0

while i < hello.length
  printf("%c → 0x%x\n", hello[i], hello[i].ord)
  i += 1
end
```

◆文字列のエンコーディング

計算機でASCII文字以外を含む文字列(たとえば日本語文字列）を表現する方法は複数ある。その方式のことを*エンコーディング*という。

◇代表的なエンコーディング

日本語の処理をする場合，以下のエンコーディングを使い分ける必要がある。

JISコード(iso-2022-jp)

主に電子メイルで用いられる。

日本語EUCコード(euc-jp)

日本語に特化したプログラム処理を作成する場合に主に用いられる。

UTF-8コード(utf-8)

多国語を同時表現する場合に用いられる。

gets 等により，Rubyプログラムで読み込むデータに日本語が含まれる場合，それに対して正規表現マッチングを行なうなら読み込んだデータと，スクリプトのエンコーディングを合わせる必要がある。本講では，スクリプトを

```
# -*- coding: utf-8 -*-
```

で書いているので，マッチ対象文字列も utf-8 にする必要がある。

◇日本語エンコーディングの変換

kconvライブラリを用いると，文字列のエンコーディング変換を簡単に行なえる。プログラムで

```
require 'kconv'
```

とするとそれ以後，文字列において以下のメソッドが利用できる。

文字列.tojis	元の文字列をJISコードに変換した文字列を返す
文字列.toeuc	元の文字列を日本語EUCコードに変換した文字列を返す
文字列.toutf8	元の文字列をUTF-8コードに変換した文字列を返す

gets等で読み取るデータに含まれる日本語文字列が utf-8 以外，あるいは不明である場合，以下のようにして正規表現検索を行なう必要がある。

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

```
require 'kconv'

while line = gets
  if /正規表現/ =~ line.toutf8
    … 処理 …
  end
end
```

もし，Rubyプログラム自身を UTF-8 ではなく EUC-JP コードで作成した場合は以下のようにすることになる。

```
#!/usr/local/bin/ruby
# coding: euc-jp
require 'kconv'

while line = gets
  if /正規表現/ =~ line.toeuc
    … 処理 …
  end
end
```

Rubyプログラム自身のものと違うエンコーディングの文字列の加工を行なう場合は，まずエンコーディングを揃えてから加工し，その後で元の文字列エンコーディングに戻す必要がある。このためには，Kconv.guess() で元のエンコーディングを取得し，出力のときにこれを利用する。以下の例は，読み込んだ各行の先頭2字を取り出して元のエンコーディングで出力するものである。

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

while line = gets
  coding = Kconv.guess(line)    # エンコーディングを判定し coding に保存
  if /(..)/ =~ line.toeuc       # euc-jpに変換して判定($1もeuc-jpとなる)
  puts Kconv.kconv($1, coding)  # 元のエンコーディングに戻して出力
  end
end
```

このように，Kconv.kconv() メソッドは，第1引数に指定した文字列を，第2引数に指定したエンコーディングに変換する。

練習問題 8

1. 以下の空欄を埋めよ。

10進数	2進数	16進数	7進数
2006			
	101010		
		bee	
			365

2. ASCIIコード表を参考に、以下の文字列に含まれる文字を ASCIIコードに直せ。

```
Japan!
```

第9講 実践プログラミングへの転換

前半の折り返し点として、楽しいプログラムを楽しく作るための知識をまとめた。練習段階を脱却して独創的なものを作っていこう。ゲーム性のあるものを作るのに必要なメソッドと、プログラム作成過程で役に立つ方法をいくつか紹介する。

◆短いRubyコードの実行

Rubyのメソッドの働きをちょっと確認したい場合は以下の方法を覚えておくとよいだろう。

-e オプション

rubyコマンドに-eオプションを付けると、次のコマンドライン引数をRubyの文と解釈して直接解釈実行してくれる。 Rubyの文全体が1個の引数になるように、シングルクォート（''）で括る。

```
% ruby -e 'printf("Hello!\n")'
Hello
```

irbコマンド

Rubyの文を1行ずつ対話実行する。コマンドラインから irb と実行してみよう。

```
% irb
irb(main):001:0>
```

ここに、Rubyの文を入れ、[Return]を押すと即座に結果が得られる。

```
irb(main):001:0> x = 5
5
irb(main):002:0> y = x**3
125
```

◆乱数

ゲーム性のあるプログラムを作るためには、プログラムを起動する度に違う値を返すメソッドが不可欠である。計算機では何らかの決まりに沿って見かけ上ランダムな数を順次発生させられる。厳密にはこれを *擬似乱数*という。

- *rand*

  乱数を発生する。

  ```
  rand(自然数)
  ```

  とすると、0から*自然数*未満の整数の乱数を発生する。

  ```
  rand(0)
  ```

  とすると0から1未満の実数値で乱数を発生する。

乱数の種を初期化するsrand メソッドで生成される乱数を変えることができる。以下のプログラム断片は、じゃんけんのグーチョキパーをランダムに出すものである。

```
janken = ["グー", "チョキ", "パー"]
srand()                          # 乱数の種を初期化
```

```
hand = rand(3)                  # 3未満の整数(0, 1, 2)を発生
printf("私の手は %s です.\n", janken[hand])
```

上のようにsrand()と引数なしで呼ぶとその時の時刻などを元に推測困難な種となるが、たとえばsrand(10)のように既知の値を与えるとその種による乱数列が出るので、毎回同じ乱数を出させてデバッグするときなどに有用である。

◆文字列操作

文字列の一部を取り出したり、切り取ったりすることができる。ゲーム性や実用性の高いものを作るときには、文字列の一部を表示したり一部を別の単語に置き換えたりすることが必要になる。これまで触れなかった少し複雑な文字列操作メソッドを紹介する。以下の説明では*S* を文字列とする。

- *S *自然数*

  自然数 の数だけ元の文字列*S* を繰り返した文字列を返す。区切り線などを表示するときに便利。

  ```
  puts "-"*79
  ```

  とすると、79個の -（と改行)を出力する。

- *S[開始位置..終了位置]*

  文字列*S*の*開始位置*から*終了位置* までの文字列を返す。最初の文字の位置を0とする。位置に負の数を指定すると文字列の終端から数え始める。たとえば、-1は最後の文字で、 -2は最後から二番目の文字である。

  例:

  ```
  "abcdefq"[0..2]
   => "abc"

  "abcdefq"[4..-1]
   => "efg"

  "abcdefq"[-3..-1]
   => "efg"
  ```

  文字列の長さを調べるには、文字列に備わる length メソッド、または size メソッドを使う。

  ```
  "abcde".size
   => 5
  "abcde".length
   => 5
  ```

  文字を1字ずつ取り出すにはたとえば以下のようにする。

  ```
  x = "abcdef"
  i = 0;
  while i < x.length
    printf("%c\n", x[i])
    i+=1
  end
  ```

- *sub, sub!, gsub, gsub!*

  subメソッドは、sub(正規表現，置換後文字列）の形式で利用して第1引数の「正規表現」にマッチする部分を「置換後文字列」に置き換える。置き換えは最初にマッチしたものに対してだけ行なわれる。

  例:

  ```
  "abcdef".sub(/bc/, "FOO")
   => "aFOOdef"
  ```

```
"This is a pen".sub(/is/, "IS")
 => "ThIS is a pen"
```

sub! は、subと同様だが、元の文字列自身を直接書き換える(破壊的操作)。つまりsub は元の文字列とは別に新しい文字列を作って返すが、sub! は新しい文字列を作らないので、元の文字列は保存されない。

例:

```
x = "This is a pen"
y = x.sub(/is/, "IS")
x
 => "This is a pen"
y
 => "ThIS is a pen"


x = "This is a pen"
y = x.sub!(/is/, "IS")
x
 => "ThIS is a pen"
y
 => "ThIS is a pen"
```

gsubは、正規表現にマッチしたものを 全て置き換える。

例:

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


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

◆時間

Rubyプログラムが動いているときの現在時刻はTimeを利用して取得する。

- *Time.now*

  現在の時刻を返す。秒数でカウントしたい場合は to_iメソッドを介して利用するとよい。

たとえば10秒以内になにかの処理をやらせる場合は以下のようにする。

```
start = Time.now.to_i
while true
  ……何かの処理……
  if Time.now.to_i - start > 10
    puts "時間切れです"
    break
  end
end
```

プログラムの実行を一時停止したいときには *sleep*メソッドを使う。

- *sleep(秒数)*

指定した*秒数* だけ一時停止する。1秒未満の指定も可能。

◆繰り返し

◇要素全てに対する繰り返し

配列に複数の値が入っているときの繰り返し処理の書き方にはいくつかある。以下の処理はどれも同じ働きをする。

```
sum = 0
x = [10, 20, 15, 44, 8, 3]
for i in x do
 sum += i
end
```

```
sum = 0
x = [10, 20, 15, 44, 8, 3]
x.each do |i|
 sum += i
end
```

```
sum = 0
i = 0
x = [10, 20, 15, 44, 8, 3]
while i < x.length
 sum += x[i]
 i += 1
end
```

◇指定した回数の繰り返し

指定した回数繰り返すには以下のどれかを用いるとよい。以下の例はどれも10回繰り返す。

```
i = 0
while i < 10
  繰り返し
  i += 1
end
```

```
for i in 1..10 do
  繰り返し
end
```

```
1.upto(10) do |i|
  繰り返し
end
```

```
(1..10).each do |i|
  繰り返し
end
```

◆ファイルの入出力処理

ファイルの内容を1行ずつ読み込みながら処理を続ける場合の書き方はいくつかある。以下の例は、どれも "datafile" というファイルの中味を1行ずつ読み、それを line という変数に代入しつつ処理を繰り返すもの

である。

```
open("datafile", "r") do |f|
  while line=f.gets
    繰り返し
  end
end
```

```
f = open("datafile", "r")
while line=f.gets
    繰り返し
end
f.close
```

```
IO.foreach("datafile") do |line|
  繰り返し
end
```

◆デモメッセージの出力

ここまでの知識を応用して、あるファイルの内容をゆっくりと出力するプログラムを作ってみよう。

roll.txtというファイルに以下のような内容が書いてあるとする。

```
西暦20XX年、宇宙創造の神は
破滅に向かう地球を救うべく、
とある場所にあるコンピュータに
一つのメッセージを託しました。
人類に残された道はそのメッセージを
手に入れること。
そのためには……


          N u m b e r   M a g i q !!
```

最後の行には10個の空白を入れておく(理由は後述)。

プログラムを起動すると、このテキストファイルの1行目が、画面の右側から動いて移動してくる。

```
          ←  西暦20XX年
              (左に向かって移動していく)
```

行の先頭に達すると次の行が出てくる。

```
西暦20XX年、宇宙創造の神は
          ←  破滅に向か
            (これも左に向かって移動していく)
```

同様に最後の行まで動きが分かるようにゆっくりと出力していく。このようなプログラムを作ろう。

◇文字列の切り出し

文字列の位置が分かりやすいように以下のような文字列を考えよう。

```
x="0123456789"
```

これを先頭の文字から1文字ずつ出てくるようにするには、

"0"	x[0..0]
"01"	x[0..1]
"012"	x[0..2]
"0123"	x[0..3]
:	:
:	:
"0123456789"	x[0..9]

のように1字ずつ増やしながら切り出す。0から始めて x.length-1まで部分文字列を取り出せばよい。

```
x="0123456789"
0.upto(x.length-1) do |i|
  puts x[0..i]
end
=>
0
01
012
0123
01234
012345
0123456
01234567
012345678
0123456789
```

◇行頭空白の作成

再び x="0123456789" の例で考えると、全部で10文字なので先頭1字だけが右端に出てくるときは、それより前に9字分の空白があればよい。これを一般化すると、文字列の長さがlengthだったときに、先頭i字だけを出力するなら、length-1-i字分の空白を入れればよい、となる。

これを考慮してループを作ると以下のようになる。

```
x="0123456789"
length=x.length
0.upto(length-1) do |i|
  puts " "*(length-1-i) + x[0..i]
end
=>
         0
        01
       012
      0123
     01234
    012345
   0123456
  01234567
 012345678
0123456789
```

◇デモ出力プログラムの完成

以上のことを組み合わせると以下のようなプログラムになる。

roll.rb

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

```
IO.foreach("roll.txt") do |msg|
  msg.chomp!
  length=msg.length
  0.upto(length-1) do |i|
    STDERR.print "\r" + " "*(length-1-i) + msg[0..i] + "  "
    sleep(0.05)
  end
  print "\n"
end
```

プログラム中に現れる "\r" は、ASCIIコード13番の CR(Carriage Return)で、端末出力時必ず行の先頭に復帰させるための制御文字である。また、実際に出力する行のところで、

- printではなくSTDERR.printとしている。
- 末尾に空白文字列 " " を余分に出力している。

という工夫がある。これらの理由を知るには、実際にその部分を取ったプログラムを実行してみるとよい。sleep の引数を変えてみると表示する速度が変わるので、最適な速さを探してみるとよいだろう。また表示元テキストroll.txtの末尾に余計な空白を入れておいたのは、全て表示し終わったあとの余韻を残すためである。

◆プログラミングスタイル

◇読みやすいスタイル

Ruby言語でプログラムを書く場合、文法的にあってさえいればプログラムは動く。しかし、プログラムを作成していく過程では、何度も読み直して修正して行ったり、他の人に見てもらう必要があるため、一目見ただけでプログラムの構造が分かりやすいような形式で書いていくことが望ましい。実際、構造が分かりやすいように気を付けて書いたものは完成までにかかる時間が短くなる。

プログラムを書く場合のスペースの空け方、括弧の置き方など、色々な流儀があるが、統一の取れたスタイルは読みやすい。以下のルールを守って書くとプログラムは非常に見やすくなる。

- インデント(字下げ位置)を揃える

  Emacsであれば全ての行で [Tab] キーを押すことで自動的に揃えられる。

```
if y == 3 then
x = 0
print "foo\n"
end
```

```
    ↓
```

```
if y == 3 then
  x = 0
  print "foo\n"
end
```

  揃えることで if, while, for などの制御構文がどこから始まりどこで終わっているかが一目瞭然となる。C-SPC でマークした位置からポイント位置までをまとめてインデントするには M-C-\ (または M-x indent-region [Return]) をタイプする。

- カンマの後はスペースをあけ、前にはあけない

  引数区切りなどのカンマの後には1つスペースを入れる。

```
printf("%sの残りのマッチは%d本です\n", who, match)
```

- 括弧、大括弧の内側には空白を入れない

  一般的に空白を多めにするとスッキリして見やすくなるが、入れすぎるとどこが大事なのかがつかみにくくなる。とくに括弧の内側は中に数式を入れることが多いので、数式を分かりやすくするためには括弧の内側には空白を入れない方がよい。

```
× …… x[ i ]
○ …… x[i]
```

上記のルールのうち、*インデント位置を揃える* ことはプログラムのendの数が合わなくなるというミスを減らすためだけでなく、プログラムの論理構造を視覚的に分かりやすくするために非常に重要である。

◆デバッグ

コンピュータに与えるプログラムに含まれる論理的誤りのことを *バグ* という。

プログラムにひそんだバグを取り除く作業のことを*デバッグ* という。デバッグするためには、プログラムが動くときのことを頭の中で追いかける必要があるので、文法エラーを取り除くのよりはるかに時間がかかる。プログラムを作る過程では、最初に書き進めて行く時間よりも後から間違いを直す(デバッグする)時間の方が長い。熟練するにつれ、デバッグにかかる時間が短くなり、同時に最初からバグの少ないプログラムが作れるようになる。

学習が進むにつれ作成するプログラムも徐々に複雑になっていくので、プログラミングに掛ける時間を節約する意味で、

- 文法エラーを起こしにくい書き方
- デバッグのしかた

を覚えておこう。

◇エラーを起こしにくい書き方

開けたらすぐ閉める!

if, while, open などは必ず対応する end を書く必要がある。にもかかわらずend を忘れてエラーになることが最初のうちは多い。end を後で入れるから忘れるのである。endで閉じるべき構文を書いたら、すぐに前もってend を書くようにすると忘れない。

```
if y%7 == 3 then
  [ ] ←endを書いてから中味を書き始める
end
```

括弧や引用符 ( ) [ ] "" '' も同様で閉じ忘れると大量のエラーメッセージに見舞われる。開けたらすぐ閉じて戻って中を書くようにすると絶対に閉じ忘れがない。

```
printf("")    まず閉じ引用符、閉じ括弧を打ってから
   ↓
printf("答は...")  戻って中味を入力
```

変数名は分かりやすく

大事な意味を持つ変数を、xとかx2 無味乾燥な名前にせず、maxninzu, goukei といった値の意味が想像しやすい名前にする。

また、変数名の綴り間違いでエラーになるのは悲しいので、積極的に補完を利用する。Emacsであれば単語を2、3文字打った状態で M-/ を押すとバッファ内に同じ単語があったときに単語の残り部分を補ってくれる。

大事なところにコメントを付ける

制御構造や条件分岐にはかならず意味があるはずである。意味を表すようなコメントを書いておく。

無意味なコメントの例

```
point=[]                        # point配列を用意
sum=0                           # sumを0にする
i=0                             # iを0にする
while line=gets                 # getsの間繰り返す
  point[i] = line.to_i          # point[i]にline.to_iを代入
  sum += point[i]               # sumにpoint[i]を足す
  i += 1                        # iに1を足す
end
```

意味のあるコメントの例

```
point=[]                        # 全員分の得点を記憶する配列
sum=0                           # 合計を保存する変数をクリア
i=0                             # 今何人目か
while line=gets                 # 入力が続く間繰り返す
  point[i] = line.to_i          # i人目の得点を記憶
  sum += point[i]               # 合計得点に追加
  i += 1
end
```

Emacsであれば、M-;(メタセミコロン)をタイプすると簡単に決まった桁位置に # が入れられる。

◇デバッグのしかた

■printfデバッグ

プログラミングの初歩の段階では、変数の値が未定義だったり、変数の値を変えずにループを作って暴走させてしまったりと、変数の値が予期せぬ状態になっていることが原因のバグが多い。これを確認するために、現在の変数の値を要所要所で出力する printf を差し込むとよい。

```
line=gets
while match > 0
  printf("debug: 今のマッチは%d本\n", match);
   :
   :
end
```

このように、printf を挟んで、そのときの変数状態を表示してデバッグする方法を、*printfデバッグ* といい、初歩的ではあるが複雑なプログラミングのときにも有効な方法である。

■pメソッドの利用

Ruby特有の便利なメソッドとして、*p* がある。pは1つ以上の引数を受け取り、文字列は文字列と分かるように、配列は全ての要素を列挙して配列と分かるように、その他の構造の値も目で見て分かりやすい形で出力し

てくれる。

p.rb

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

number = 5
string = "えっほえっほ"
a = [100, "foo", "ばあ"]

p number, string, a
```

上記のプログラムを実行すると以下のようになる。

```
% ./p.rb
5
"えっほえっほ"
[100, "foo", "ばあ"]
```

配列の値がどうなっているか確かめるときにはpメソッドが有用である。

◇デバッガ

プログラムの動きを止めながら、1行ずつ実行しては変数の値を確認することができる。このような道具を*ソースレベルデバッガ* といい，EmacsでRubyプログラムを作成する場合には rubydb が使える。

コマンドラインからプログラムを起動するときに、 rdbg によって行う。たとえば以下のようにする。

```
% rdbg program.rb
```

と入力する。すると，デバッグモードでプログラムが起動する。この画面のデバッガプロンプト「(rdbg)」で入力できるコマンドの代表的なものには以下のものがある。

- s (step)

  ステップ実行。1行ずつ実行する。

- n (next)

  次の行まで実行。メソッド呼出しはトレースされない。

- c (continue)

  継続実行。ブレークポイントで止まる。

- b (break)

  「b *行番号*」などとして、ブレークポイントを設定する。引数を省略すると設定されているブレークポイント一覧を出す。

- d (delete)

  「b *行番号*」などとして、指定したブレークポイントを除去する。引数を指定すると全て除去する。

- p (print)

  「p *変数*」などとして，p の後ろに書いた変数の値を表示する。

- l (list)

  ソースプログラムを表示する。

- disp (display)

  「disp *変数*」などとして，変数の値をステップ実行ごとに常時表示する。 undisp により解除できる。

短いプログラムの場合はデバッグバッファで，s を打ち続けて1行ずつ実行を進めて，適宜変数の値をpで表示して確認する。Return を空打ちすると *直前と同じコマンド* が繰り返し実行される。

長いプログラム，あるいはループで何度も繰り返すプログラムの場合は動きを調べたい場所の直前にブレークポイントを設定してから c で継続実行する。ブレークポイントを設定するにはソースプログラムの該当行番号を調べ、

```
b 行番号
```

と入力する。

練習問題 9

1. 入力した最後の文字を1字だけ出力するプログラム last1c.rb を作成せよ。実行例は以下のとおり。

```
% ./last1c.rb
This is a pen!
!
Ruby
y
[C-d]
```

2. 標準的な端末画面は横80桁である。入力した文字列のうしろに画面の右端いっぱいまで続くハイフン(-)を追加して出力するプログラムhyphen.rbを作成せよ。実行例は以下のとおり。

```
% ./hyphen.rb
This is a pen!
This is a pen!----------------------------------------------------------------
Ruby
Ruby--------------------------------------------------------------------------
(いずれの出力も全部で80文字)
```

3. 変数mに複数の文字列が配列として入っている。

【例】

```
m = ["世界で一番高い山は?", "日本で一番高い山は?",
  "エトセトラ、を表す3文字の略語は?",
  "高速道路で料金を自動的に払うシステムの略称は?"]
```

4つある要素のうち1つをランダムに選んで出力するプログラム randq.rb を作成せよ。

4. quiz.datというファイルに、「問題」、「答」が 1行単位で順番に入っている。つまり、1, 3, 5, 7, 9, ...と奇数行目には問題が、2, 4, 6, 8, ...と偶数行目にはその答がある。

```
1月
January
2月
February
月曜日
Monday
  :
  :
```

起動すると、quiz.datファイルを自動的に openし、問題だけを集めた配列、答だけを集めた配列を作成した上で、問題と答を交互に出力していくプログラムqpod.rbを作成せよ。

5. (0..52).collect とすると、0から52までの整数(53個)を要素に含む配列が生成される。たとえば、

```
x = (0..52).collect
```

とするとxは [0, 1, 2, ····, 52] のような配列を持つ。このようなxに対し、以下の手順を繰り返すと要素の順番がでたらめになる。

i. iを0とする

ii. iの値が53未満ならvまでを繰り返す。

iii. 0〜52の乱数(整数)を生成し、tとする。

iv. x[i]とx[t]とで、値を交換する

v. iの値を1増やす。

上記の手順行ない、xの値を出力するプログラム shuffle.rbを作成せよ。なお、プログラムの始めと終わりは以下のようにするとよい(???の部分を埋める)。

```
x = (0..52).collect
???
???
p x
```

6. 4番で作成したqpod.rbを改良し、問題がランダムな順番で出てくるようにしたプログラムqpod-shuffle.rbを作成せよ。

7. コンピュータとじゃんけんができるプログラムjanken.rb を作成せよ。実行例は以下のとおり。

```
% ./janken.rb
じゃんけんしましょ、
あなたの手は? [グーなら0、チョキなら1、パーなら2]: 0
あなた「グー」 : 「チョキ」わたし
があ〜ん負けちゃったー! 強いのね。
もういちどじゃんけんしましょ、
あなたの手は? [グーなら0、チョキなら1、パーなら2]: 1
あなた「チョキ」 : 「グー」わたし
やったー、勝ったー! よわよわじゃん。
もういちどじゃんけんしましょ、
あなたの手は? [グーなら0、チョキなら1、パーなら2]: 2
あなた「パー」 : 「パー」わたし
おっと引き分けだ。仲良しこよし。
もういちどじゃんけんしましょ、
あなたの手は? [グーなら0、チョキなら1、パーなら2]: [C-d]
あらもうおしまい? さようなら!
```

8. 前問のjanken.rbを改良し、[C-d]で入力を終えたときに、それまでの対戦成績を出力するようなプログラムjankenstat.rbを作成せよ。