# プログラミング入門

瀧川 一学 (たきがわ・いちがく)

**プログラミング【program(m)ing】**立案。プログラム②を作ること。自らの不注意さを再確認する作業。「お前の人生ってなんか―みたいだよねえ」― **-じっしゅう【―実習】**限られた時間の中でプログラミングを実際に体験しながら、それを自らやってみようという動機を得る試みの一。
**プログラム【program(me)】**①目録。番組。予定。計画表。②コンピュータに対して、どのような手順で仕事をすべきかを、機械が解読できるような特別の言語などで指示するもの。またそれを作成すること。③せっかく苦労して作ったため何らかの意味を求めたくなる切ないものの一。

本実習は演習シリーズ「プログラミング実習」の第一回目「プログラミング入門」で以後の実習で必要となる基礎的な知識についての実習となります。本実習を通して利用することになるプログラミング言語rubyを用いて、実際にプログラムを書くための基本的な考え方とそのルール、実際の操作、プログラムの動かし方などを扱います。

この実習は科学技術振興調整費によるバイオインフォマティクス人材養成プログラム「ゲノム情報科学研究教育機構」[1] による演習シリーズ「プログラミング実習（公開コース）」の第一回目にあたります。実習全体の概要は以下の通りです。

概要：バイオインフォマティクスに必須のプログラミング言語についての概要を学び、簡単なプログラミングができるようになるまでの実習を行う。インターネット上のリソースを活用して、生物学・化学のデータベースを利用する実習も行う。

担当教員：5名

片山 俊明（東京大学）　長崎 正朗（東京大学）
伊藤 真純（京都大学）　瀧川 一学（京都大学）
田中 伸也（京都大学）

日程：2006年度(平成18年度) プログラミング実習

| 日時 | | 会場 | 実習名 | 講師 |
|---|---|---|---|---|
| 05/12 | 16:30-18:00 | 京都 | プログラミング入門 | 瀧川・伊藤 |
| 05/19 | | | データリソース活用 I | 片山・伊藤 |
| 05/26 | | | データリソース活用 II | |
| 06/02 | | | データリソース活用 III | |
| 06/09 | | | データリソース活用IV | 片山・伊藤・田中 |
| 10/06 | 16:30-18:00 | 東京 | プログラミング入門 | 長崎・片山 |
| 10/13 | | | データリソース活用 I | 片山・伊藤 |
| 10/20 | | | データリソース活用 II | |
| 10/27 | | | データリソース活用 III | |
| 11/10 | | | データリソース活用IV | 片山・伊藤・田中 |

近年、ハイスループットな生物情報の測定技術の発展と浸透、幅広い生物種についてのゲノム配列決定プロジェクトの終了もしくは進行、核酸・タンパク質・脂質・糖鎖の配列や代謝産物・薬剤等の低分子化合物に関連する立体構造情報や関連する反応情報・相互作用情報・触媒酵素情報など、日々発表される莫大な医薬生化系の文献情報、様々な種類の情報を格納したデータベースの整備とインターネットの普及によるそれらの情報への手軽なアクセスの確立などなどの要因を受けて、それらを例えば紙に印刷して、人の目で一から検証するということは莫大な時間を要するという意味でも、ミスが混入しやすいという意味でも現実的ではない作業となっており、この意味で**自分が望む情報を得るために自分が望むような手順でコンピュータによって様々なデータを処理していくこと**が重要になっている。

そこでコンピュータを利用できる環境にあるひとはこの作業を「プログラミング」というステップを通して**大幅に効率化することができる**ことに思い当たる。 そのために必要な手順としては

1. どのような情報がデータとして与えられ、そこからどんな情報を得たいのか、明確に整理する。
2. 実際にデータをあつめる。
3. あなたのコンピュータにデータをコピーして、整理した手順に従って望む操作を間違いなく行うようにあなたのコンピュータに指示する。

のようなものが典型的である。この演習では主に3.の手順をどうするかを簡単な実習を通して体験することを趣旨としている。また、インターネットを通して得られるデータを使う場合などには2.もある程度プログラミングを通して簡易化することができる。

**結論①：適切に「プログラミング」すると手作業では時間がかかりすぎたり人手には複雑すぎたりして無理そうな作業をあなたのコンピュータにかわりにやらせることができそうである。**

[1] http://www.bic.kyoto-u.ac.jp/egis/index_J.html

## a. 最初の一歩は常に最も難しい。

> **とり-かかり【取掛り】**最初。手はじめ。とっかかり。しばしばつかむのに時間と勇気を必要とする。

結論①はあなたが今行いたいデータの処理手順をプログラミングすることによってあなたのコンピュータがそれをやってくれることを述べている。しかし、コンピュータは別に魔法の箱ということはないので、原理的に手作業でできない操作（どうやるのか手順をあなたが全く知らないこと）をやってくれるわけではない。

たとえば漠然と「これらのデータをまとめたい」という手順ではこのままでは何をやっていいか全く分からない。具体的にはたとえ面倒でも次のように何をやるべきかをすべてコンピュータに指示しなくてはならない。

1. 「data1.txt」と「data2.txt」という名前のファイルがありますのでそれぞれの一行目を抜き出してください。
2. そこには「52.3 24.5 44.8」などと数字が空白文字を区切りに３つ並んでいると思いますのでdata1.txtからとった数字(a, b, cとする)とdata2.txtからとった数字(d, e, fとする)を「a+d b+e c+f」のように足し合わせて下さい。
3. その3つの数字を画面に表示してください。

プログラミングとは詰まるところ、単にこのように**コンピュータが行うべきことの手順を順番に記述して、コンピュータに告げること**である。

## b. 宣告は言語によって行う。

> **いしん-でんしん【以心伝心】**① [仏] 禅家で、言語では表されない真理を師から弟子の心に伝えること。②思うことが言葉によらず、互いの心から心に伝わること。「もう前みたいに―じゃなくなっちゃったよね」

残った（幾分まだ抽象的な）二つの問題点は

**❶ その手順をどのようにしてコンピュータに告げるのか？**

**❷ コンピュータにどこまで指示してやる必要があるのか？**

ということである。もちろん、あなたのパソコンでいつものようにお気に入りのMicrosoft Wordなどを使ってさきほどの1.〜3.の文章を打ち込んでみてもコンピュータがあなたにまったくちっとも何もしてくれないことはそろそろ薄々お気づきのことと思う。

**コンピュータに指示するには専用の言語を使うことが必要である。**言語と言われても、げ、言語ってなに？と思う方もあろうが、たいていアルファベットや数字や記号などといった我々の文字体系をある規則に従って使い、それらの使い方が日本語とかではないというだけである。これらの「言葉遣い」はしばしば「プログラミング言語」などと呼ばれる。言語といっても人工的な取り決めにすぎないので、使うことができる文字の種類などは決められたものだけだし、文法は破格を許さない。

プログラミング言語というのは一種類だけではなく様々な人が様々な取り決めを提案したので、ものすごく多数存在する[2]。あなたは基本的にどれを使っても良いが

**「そのプログラミング言語で書いた手順」を翻訳して書かれた手順をコンピュータに実行してもらうためのプログラムがあなたのコンピュータで動作するもの**

を選ぶのが少なくとも懸命と言えるだろう。人気がある言語はたいていこの条件を満たしている。

では、この節の最初の質問の後者❷を考えよう。この質問の意味は、例えば「〜という名前のファイルの一行目を抜き出して」という命令文は存在するのか、それとも、「〜という名前のファイルを探し出して最初から一文字ずつ読んで、数字と解釈できる場合には数字にして」とか、より細かく言うべきなのか、そもそも「数字と解釈できる場合には」などということをプログラミング言語では取り扱えるのだろうか、などという一連の各論的疑問である。

これの答えは**用いる言語に依存する**、である。より正確に言えば「ファイルを読み書きする」とか「数値xがあるときsin(x)を求める」とか「画面に文字を表示する」とかそういう**よく起こりえて設計者が想定内の操作は既に命令として用意されていることが多い。**これらはしばしば「ライブラリ」などと呼ばれている[3]。ライブラリに用意されているとしたら、どのようにして数値xからsin(x)を導くのか、どのようにしてファイルから文字を読み出せば良いのか、あなたは特に知らなくても良い。どのように書けば良いか、だけ知っていれば良いのだ。 ライブラリはグレートなのだ。と、John Bentleyもそう言った。

したがって、一般には、プログラミングに用いる言語をあらかじめ一つ決めて、その言語における取り決めとライブラリに習熟する必要がある。またそれぞれの言語は設計方針も異なるし、ライブラリにどんなものが用意されているかも異なるため、**一般にそれぞれのプログラミング言語は解ける問題に得手不得手がある。**

**結論②：使いたいプログラミング言語を決め、そのルールとライブラリの使い方に習熟する必要がある。**

---

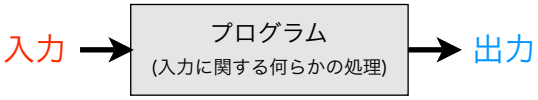
[2] 具体的にどういう言語があるか知りたい方はたとえば http://dir.yahoo.com/Computers_and_Internet/Programming_and_Development/Languages/ などのポータルサイトのカテゴリ名を参照すると良い。

[3] どこまでが「言語の取り決め」でどこからが「ライブラリ」なのかは実は言語の設計方針によってだいぶ違って一般的にはあいまいである（たとえば、 加算命令「数値xと数値yを足す」はどちらなのだろうか？）。

## c. 苦労は教科書中でなく現場で起きる。

けい-けん【経験】人間が外界との相互作用の過程を意識化し自分のものとすること。人間のあらゆる個人的・社会的実践を含む。「一してない奴に何を言っても伝わらないと思う？」

結論①および結論②を考察するに至った経緯を振り返ってみると「プログラミング」によってあなたが今から作るべき「プログラム」というのは、たぶん大まかに以下のような形をしている。




例１）入力＝10個の数字　出力＝その平均値
例２）入力＝ファイル　出力＝内容の画面への表示
例３）入力＝キーボードからの日本語文字の入力
　　出力＝その読みの漢字候補のリストを表示
例４）入力＝ある遺伝子配列　出力＝Cの含有量
例５）入力＝「http://www.genome.jp/」等の文字
　　出力＝そのページの内容の画面への出力
例６）入力＝マウスクリック情報
　　出力＝ポイントされている部分の拡大表示
例７）入力＝いままで書いた日記の全文章
　　出力＝今後どう生きるべきかに関する啓示
例８）入力＝デジカメの写真　出力＝モノクロ調にしたもの

そして、大雑把に言うと、プログラミングを用いた問題解決はだいたい以下のような過程で進行する。

**（１：入出力の確定）**まずプログラムへの「入力」とプログラムからの「出力」を**すべて明確に**確定させる。これはどのような情報からプログラムによってどのような情報を引き出したいのかを熟考する作業となる。

**（２：解法・アルゴリズムの確定）**次に、「入力」からどのような操作をどのような順番で行えば望むべき「出力」が得られるのか、を確定させる必要がある。操作が単純で明らかな場合（例１など）もあるし、操作がたくさんあっていろいろな知識が必要な場合（例５のWWWブラウザや例６など）もあるし、設定が悪くて解けない場合もある（例７など）。最後の場合はコンピュータプログラムによる解決は今のところ望めない。

**（３：コーディング）**問題とそれを解くための手順が確定したら、いよいよそれを（多少張り切って）プログラミング言語を用いて記述する（ひたすら文字を入力するだけなのであるが）。

**（４：実行）**記述しおわったら、翻訳プログラムを実行して実際に作った自分のプログラムを動かす。

**（５：デバッギング）**思った通り動かなかったり、翻訳プログラムがエラーを出すような場合は、もう一度、プログラムの記述が正しいかを念入りに見直す。通常、この作業はプログラムが大規模になればなるほど猛烈に時間がかかるため、様々な支援ソフトウェアも存在する。

**（６：テスト）**思ったように動くようになったら、予め「正しい出力」が分かっているような「入力」をいくつか与えてみて、望む動作をしているかどうかを丁寧に検証する。この作業が作ったプログラムの質を決定する。

**（７：運用・保守）**実際にそのプログラムが活躍すべきシチュエーションにて実用的に使用する。また必要に応じて保守を行う。

段階１と２は「問題の策定」と「問題解決法の設計」（仕様決定・システム設計）を決めるところであり、通常は最も重要であり最も難しい[4]。３と４は狭義の「プログラミング」であって、実際にコンピュータに向かい合って行う。ここにはコンピュータ言語や関連する技術の知識などが必要となる。

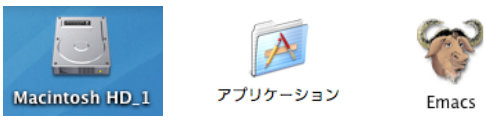
５と６は自分の作ったプログラムが本当に望んだ通り動いているのか確認する大切な手順である。しばしば、たまたまうまく動いているように見えただけだと後で気づくこともあり、応用目的の重要性が高いほど、丁寧さが要求される。例えば、旅客機の軌道制御プログラムがテストコース以外を飛行したときに正常な動作が保証できていなければ大惨事を招きかねない。どのようなテストが必要になるだろうか？

## d. 前置きのあとに準備を始める。

じゅん-び【準備】ある事をするのに必要な物や態勢を前もってととのえること。用意。支度。「まだ心の一ができてないから」

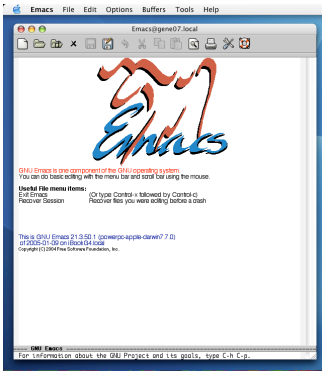
この演習ではプログラミングのうち最も基礎的な３と４のステップを扱う。それでは以下に、実際に演習室のパソコン（Mac OS）でそれを行う方法を説明する。

今回扱うプログラム言語rubyだけに限らずプログラムは通常**アルファベットや数字・記号などの文字データだけを用いて記述する**。Microsoft Word等のワープロソフトは文字データだけではなく、フォントなどの文字の修飾情報や、画像、表、グラフといったいろいろの情報を付与でき、それらのデータ情報を同時に保存するため、最初のプログラミングには不向きである（できないわけではない）。ここでは「Emacs」というテキストエディタ（文字データを編集するソフトウェア）を用いる。

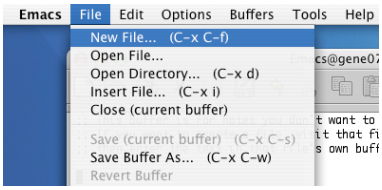



[4] ソフトウェア工学ではこれらシステム開発の行程について様々な理論が議論されてきた。大規模な場合は仕様設計の段階でより細かい作業単位へと分割を行い、それぞれの作業ごとに中間的な入出力が発生する。

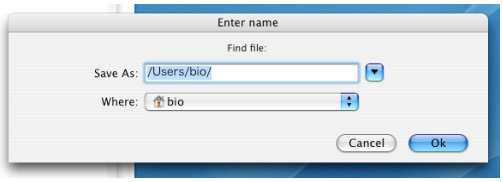
デスクトップ上からMacintosh HD_1→アプリケーション→Emacsの順でアイコンをダブルクリックしてEmacsを起動すると以下のようなウィンドウが開く。



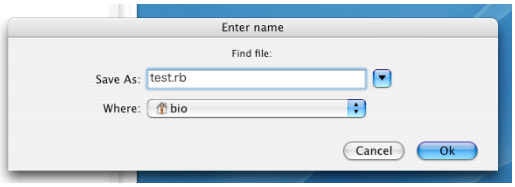
次に最初のテストで用いるプログラムを「test.rb」というファイル名で新規作成する。新規ファイルのアイコンをクリックするか、画面上部のメニューバーからFile→New File..




を選択すると、保存する場所と名前を訪ねるウィンドウが開く。Save Asの右横の三角をクリックすると保存するフォルダを選択することができるが



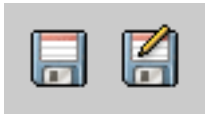

今回は単にSave Asのあとに出ている文字を消して「test.rb」と入力する。




すると何も無い空白の画面が出てくるので、ここにキーボードから文字を入力していく。ここでは最初のプログラムは次のようなものを入力してみよう。

```
greeting1 = "Hello, world!"
greeting2 = "but, goodbye."

puts greeting1
puts greeting2

puts greeting1 + " ... " + greeting2

puts Time.now
puts Time.now.year, Time.now.month, Time.now.day

puts rand(6), rand(6), rand(6), rand(6)
```

入力し終わったら、保存のアイコンかメニューからFile→Saveを選択してファイルを保存しておく。

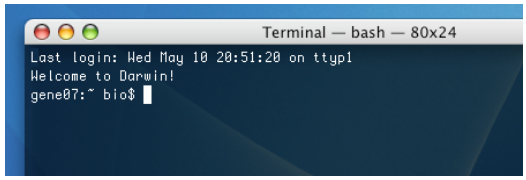


これでruby言語でプログラムを記述できた。今は何をやっているか分からないと思うが**プログラムの記述とはこのように文字を入力してそのファイルに名前をつけ、保存するだけ**である。

では、次に今記述したプログラムをruby言語の翻訳プログラムを動かして翻訳させ、あなたのコンピュータにこれらの手順を伝えて、行ってもらうことにしよう。

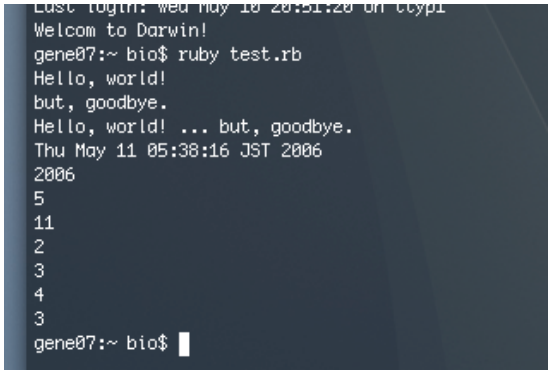
コンピュータ自体にいろいろの指令を伝えるにはここでは「ターミナル」というものを使って行う。まず、画面下部にあるアイコン一覧から「ターミナル」のアイコンをクリックする。


すると以下の命令（コマンド）入力のためのウィンドウが見える。この▉の所へこのコンピュータ（正確にはこのコンピュータのオペレーティング・システム）へのコマンドを書くのである。




**ruby言語のプログラムを翻訳して、そこに書かれた手順を実行させるコマンドは「ruby ファイル名」**である。このような命令を「ターミナル」に入力することによって機械に（ここではrubyプログラムを翻訳して実行しなさいという）指令を出すのである。

先ほど記述したruby言語によるプログラムは「test.rb」という名前で保存してあるので、これを翻訳してコンピュータにこの手続きを実行してもらうには「ruby test.rb」と入力すれば良い。




すると、上のように、そのプログラムが出力した文字がそれ以降の画面にずらずらと表示されるだろう。細かい数字の違いはあっても、だいたいこのようにこの画面にいろいろ表示されていればOKである。

このようにターミナルではプログラミング言語とは別にコンピュータ自身に対して直接コマンドを記述する必要がある。演習室のコンピュータはMac OSというものなのでそれに従った命令を書く必要がある。Mac OSの「ターミナル」ではUNIXというオペレーティングシステムとだいたい同じコマンドを受け付ける。

## e. 答えはきっと一つではない。

> **せん-たく【洗濯】**（センダクとも）①衣類などの汚れを洗ってきれいにすること。洗いすすぎ。②心にたまったつらい思いを忘れ去り、さっぱりした気持になること。浮世床（初）「命の―」

今述べたように、プログラムにコンピュータが行うべき手順をruby言語で記述して、名前をつけて保存して、それを「ターミナル」からrubyコマンドを使うことによって、コンピュータに翻訳させ、その手順を行ってもらう、という一連の流れがプログラミング作業の基本的な部分である。

しかし、ここではまずruby言語とはどんなものなのか？どのように書けば良いのか？このように書いてみたら結果はどうなるのか？という基本的なことを学ぶわけで、その文字を並べる規則を学習しなくてはならない。

さきほどのプログラムとその結果を眺めるに少し推測できることには

プログラムは上から順番に実行される

ということである。そして、一行一行に意味がある。そこで例を見ながら**それぞれの行がどんな意味を持ち、どんなことをやっているのか、を理解していく**のが、やはり近道ではなかろうかと考える。そこでここでは「rubyファイル名」というやり方以外のものとして、ファイルに保存しないでrubyの命令を直接コンピュータに伝えるコマンド「irb」(interactive ruby)について紹介する。

まず、次のように「puts "なになに"」と言うruby言語の一文について推測を働かせてみよう。ファイルに

```
puts "This is the first line."
puts "This is the second line."
puts "This is the third line."
```

のように書いたとしたら、rubyコマンドによる出力は

```
This is the first line.
This is the second line.
This is the third line.
```

のようになる（疑うなら手順を覚えるためにも先ほどの「test.rb」の内容を書き換えて「ruby test.rb」とrubyコマンドをターミナルで実行してみよ）。ここから、どうやら「puts "なになに"」と書いて改行しておくと「なになに」という文字が画面に表示されるということが分かる。そして、上から順番に指示がコンピュータに伝えられて、実際にその手順が行われている。

これを今度はその「irb」コマンドを使って確認してみよう。まず、ターミナルで「irb」と入力すると下図のように「`irb(main):001:0>` 」などという文字が出てruby言語の命令を受け付けるようになる。例の場合は続けて、「puts "なになに"」を試してみている。

```
gene07:~ bio$ irb
irb(main):001:0> puts "Are you OK?"
Are you OK?
=> nil
irb(main):002:0> puts "I'm OK"
I'm OK
=> nil
irb(main):003:0> exit
gene07:~ bio$
```

そのあとに勝手に表示される「`=> nil`」などというのはジャマだがひとまずおいておこう。このようにして短い命令文がどういう命令としてコンピュータに伝わるのか確認することができる。この状況を終了するには「`exit`」と書くか「`quit`」と書くか「Ctrlキーとdを同時に押す」かすれば良い。

この演習では「`irb(main):001:0>` 」という箇所に現れる情報はひとまず利用しないので、単に「irb」と入力するのではなく

```
irb --simple-prompt -r irb/completion
```

と入力して使うこととする。まず「--simple-prompt」というのを「irb」に続けて付け加えると、さきほどの「`irb(main):001:0>`」という箇所が、シンプルに

「>>」となり、**自分が何を入力して、コンピュータが何を出力したのか、だいぶ見やすくなるため、初心者には良い**だろう。気にならないし、コマンドが「irb」と短いほうが良いじゃなーい、という場合はそれで全く問題はない。

```
gene07:~ bio$ irb --simple-prompt -r irb/completion
>> puts "Is this a pen?"
Is this a pen?
=> nil
>> puts "No, this is an orange."
No, this is an orange.
=> nil
>> exit
gene07:~ bio$
```

また「-r irb/completion」は「Tab」キーを押すといろいろ勝手に補完してくれる機能を使うよ、という意味でここではあまり詳細には触れない。

## f. 忘却の上に豊かさが積もる。

> **せいり【整理】**①乱れた状態にあるものをととのえ、秩序正しくすること。「論点を一する」「いいから落ち着け、話を一しようじゃないか」②不必要なものを取り除くこと。「無駄な枝を一する」③[法]株式会社が支払不能または債務超過に陥るおそれのあるときなどに、再建のために、商法に従い裁判所の監督の下に行われる手続。

いろいろな事を述べたので、実際にruby言語がどういう規則になっているかを体験していくまえに、さしあたって、そのための前提と手順をまとめておこう。

### プログラミング

ある問題をプログラミングを通して解決したいとする。そのためには「その問題をどういう手順で解けば良いか」を事前に良く理解しておく必要がある。何がデータとして与えられ、何を求めたいのか、その入出力をつなぐために、どういう手順が必要か、などすべてである。プログラミングとは、解き方の手順を知らない問題をコンピュータが勝手に解いてくれる作業ではなく、ただあなたが理解している手順をコンピュータに代わりに行ってもらうように指令を逐一記述するという作業である。

### ライブラリ

ただし、「ライブラリ」としてある一式の手順が予め用意されている場合には、その内容を知る必要はなく、どのように書けばそれが利用できるか、を知っていれば良い。たとえば、どういう仕組みでインターネット上のファイルが読み書きされるのか知らなくても、ライブラリが既に用意されており、その使い方がたとえば

```
open("http://www.ruby-lang.org/ja/LICENSE.txt")
```

と書けばそれを行ってくれる場合、特に詳細を知らなくてもプログラム中でその作業を行うことができる（もちろん、知っておくにこしたことはないであろうが）。

従って、ライブラリにどのような手順が用意されているかを知ること（あるいは、調べること）はしばしば必要な知識を大幅に軽減してくれるし、作業を大幅に効率化してくれる。

### プログラミングの実際

問題が理解されていれば、狭義には

（1）コーディング

（2）実行とデバッグ

（3）テスト

という流れになる。

### コーディング

ruby言語の規則にしたがって、文字を入力していき、それに名前をつけて、ファイルをして保存するだけの作業である。今回は「Emacs」を起動し、ファイルを新規作成し、名前をつけ、そこのウィンドウに文字を入力し、保存するという作業になる。

### 実行

ファイルに保存したプログラムに書かれた手順を実際に翻訳してコンピュータにそれを行ってもらうには、ruby言語の場合、「ターミナル」を起動し、そこへ「ruby ファイル名」というコマンドを入力することで、それを行うことができる。プログラムが画面に文字を表示するような場合は、そのコマンド入力のあとに一連の結果が出力される。

### デバッグ

実行してみたところ予想外の出力が得られたとか、エラーが出た場合、プログラムの記述か問題の解法が間違っている。丹念に調べてそれを取り除くこと。プログラムの規模が大きくなったり、操作が複雑な場合、この作業には大変時間と手間がかかる。

### テスト

いろいろなデータでプログラムが本当に望んだ通りに作動しているのかをよくチェックすること。

### インタラクティブ実行

上記の実行方法以外に「irb」コマンドを使うと、一行一行のrubyの命令文を翻訳してコンピュータに伝えることができ、命令文ごとにどんなふうに動作するかを確認することができる。（詳細は後述）

※ここまでの文中に現れる辞書風の記述は岩波書店「広辞苑」第四版および第五版の記述、もしくはそれに加筆したりそれに似せた創作です。

## 0. 簡単な確認をする。

ruby言語の文を翻訳して実行してもらうには二通りの方法があるのだった。この違いを見るためにまず次のような計算を考えよう。最初に打ち明けておくとrubyはこのような足し算引き算の文を命令として用意している。

```
20 + 30 + 40 - 50 + 120
550 + 220 + 110
```

このrubyプログラム文の手順を実際にコンピュータに行わせるためには、その旨をコマンドによって（オペレーティングシステムに）指令する必要があるが、

（1）ファイルに入力して名前をつけて保存して、「ターミナル」を起動し「ruby ファイル名」を入力

（2）「ターミナル」を起動し「irb」を入力し、そのあとに上の命令文を一つずつ直接入力

の二つの方法があるのであった。（1）は通常実行、（2）はインタラクティブ実行である。

最初に（1）を試してみよう。「test2.rb」という名前で保存し、実行してみよう。

```
gene07:~ bio$ ruby test2.rb
gene07:~ bio$ ▮
```

ありゃ？　何も表示されない…。上のプログラムは本当に足し算をしてくれたのだろうか。

次に（2）を試してみよう。「irb」コマンドだ。

```
gene07:~ bio$ irb --simple-prompt
>> 20 + 30 + 40 - 50 + 120
=> 160
>> 550 + 220 + 110
=> 880
>> exit
gene07:~ bio$
```

おお。今度は足し算の結果が「=>」のあとに計算されている気がする。

実は先ほどのプログラムは足し算は確かに計算したが、画面にその結果を表示してください、という手順を書いていなかったため、コンピュータはそれを表示しなかったのである。従って、通常実行の（1）では見た目は何もしていないのと同じである。コンピュータは書かれたことしかしないこと、プログラムには最終的には出力が必要なことが少し認識されただろうか。

しかし、（2）のインタラクティブ実行では、各行の命令文（ここでは足し算）がどんなふうな結果をもたらしたのか、「=>」の後に出力している。これが（1）と（2）の実行方式のひとつの違いである。

「画面に〜を表示してください」はruby言語では「**puts 〜**」と記述すれば良い。従って

```
puts 20 + 30 + 40 - 50 + 120
puts 550 + 220 + 110
```

ならば、通常実行でも

```
gene07:~ bio$ ruby test2.rb
160
880
gene07:~ bio$ ▮
```

と結果を表示してくれるのだ。「irb」では

```
gene07:~ bio$ irb --simple-prompt
>> puts 20 + 30 + 40 - 50 + 120
160
=> nil
>> puts 550 + 220 + 110
880
=> nil
>> exit
gene07:~ bio$
```

となる。文の結果が「nil」に変わった。これは「puts 〜」という命令文自体は値を出してしまって、もう何も値を持っていないから「何もなし」という意味である。

## 1. ビルディングブロックを考える。

プログラムはアルファベットとか記号とか数字とかキーボードから入力される「文字データ」を並べて記述される。まず、ruby言語で**種々のデータがどのようにプログラム中で文字データとして表現できるか**を考える。

〔数字〕数字を表す文字はたいていそのまま入力すれば望むように解釈される。

```
>> 109
=> 109
>> 109 + 1
=> 110
>> 109.12 + 1
=> 110.12
>> 12020 - 12000
=> 20
>> 3 + 4
=> 7
>> 3 * 4 + 1
=> 13
>> 3 * (4 + 1)
=> 15
```

```
>> 100.0 / 6.0
=> 16.6666666666667
>> 100 / 6
=> 16
>> 100 % 6
=> 4
>> 16 * 6 + 4
=> 100
>> 2 * 2 * 2 * 2
=> 16
>> 2 ** 4
=> 16
>> -200
=> -200
```

かけ算の記号は「*」、割り算は「/」、べき乗は「**」、剰余は「%」などの記号を用いる。括弧づけなどによる計算の順番付けや、マイナスの数なども入力できる。ここで注意すべきことは「100 / 6」が「16」となっていることであるが、これは**ruby言語では整数同士の計算結果は整数と決まっている**からである。

〔文字〕数字のときと違って、文字はそのまま書いてしまうと望む動作をしない。

```
>> Sunday
NameError: uninitialized constant Sunday
        from (irb):15
        from :0
>> faster than light
(irb):16: warning: parenthesize argument(s) for futur
NameError: undefined local variable or method `light'
        from (irb):16
        from :0
>>
```

おそらく何が起こっているか分からないとしても、なにやら厄介なことになったようだと察することができる。Errorなどという文字もちらっと出ているようだ。

これは先ほどの「puts」のように命令文にもまたアルファベット文字を用いるので、文字データの並びである「puts」が文字そのものを表したかったのか、命令を表したかったのか、コンピュータには分からなくなってしまうからである。

一番最初の例で見たように文字列はダブルクォーテーションやシングルクォーテーションでくくって表現する。これでたとえば「110」という文字自体を表すには"110"などと入力すれば良いことが分かる。

```
>> "faster than light"
=> "faster than light"
>> "puts"
=> "puts"
>> "110 - 250"
=> "110 - 250"
>> 110 - 250
=> -140
>> "abc" + "def"
=> "abcdef"
>> "ping!" * 3
=> "ping!ping!ping!"
```

また文字にも文字同士の結合「+」および文字の繰り返し「*」が定められている。ちなみに**文字が並んだもの**は「文字列 (string)」という言葉で呼ばれる。

こうして今、文字や数字とそのちょっとした操作がかけるようになった。この6と書いたら整数「6」を表すし、helpとかいたらダメだったけれど "help" と書いたら文字列「help」を表せば、コンピュータに数字や文字について伝えることができる。

この 6 やら "help" やら、**文字の並びを用いて、何らかの値を直接表現したもの**は「リテラル」などと呼ばれる。例えば、「6」という文字の並びは整数の6を表す「整数リテラル」であるし、「"help"」という文字の並びは文字列helpを表す「文字列リテラル」である。

## 2. 名前をつける。

ここまででひとまず数値と文字列という二つの種類のデータの「値」をどう書くか、それらをどう画面に表示させるか（putsする）、について述べた。

たとえば "this is a pen."という文字列と、 "but this is an orange." という文字列を交互に２回表示したいとすると、これらを文字列リテラルで書くと、

```
puts "this is a pen."
puts "but this is an orange."
puts "this is a pen."
puts "but this is an orange."
```

となる。

しかし、プログラミングは手作業を簡易化したいのであったのだから、このようにプログラム中で何度も同じ文を入力していては全く意味がない。実際には、これらの値を記録しておく「変数」という入れ物を用意して、そこに名前をつけて、計算した値を記録して取っておいたり、プログラム中で再びそれを利用したりする。

"this is a pen." という文字列をaとして、 "but this is an orange." という文字列をbとしてコンピュータのメモリ上に名前をつけて最初に記録しておけば、

```
a = "this is a pen."
b = "but this is an orange."
puts a
puts b
puts a
puts b
```

のようにaやbとして使える。このaやbは「変数」の名前であってリテラルではない。このように「=」記号の右側の値を計算して左側の名前として記録する。この操作を

変数 a へ値 "this is a pen." を代入する

などと言う。右側では計算なども含むことができるが、左側は変数の名前なので、左側で何かを操作することはできない。例えば、

```
a = 1+2+3+4
puts a
a = a + a + a
puts a
a = 5+6
puts a
a = "this is a pen."
puts a
```

などは可能であるが（「irb」で試してみよ）、「=」は数学の記号とは意味が異なり「左右が等しい」ということではなく、「右を左の名前で記録」という意味で、

```
a + b = 3
```

や「1 + 2 = a」などと書くことは**できない**。また、

```
a = 10
a = a + a
puts a
```

の場合は、10という値を名前aとして記録して、a+aを計算してその結果を再び名前aとして（上書き）記録する、という意味である。**プログラムは上から実行される**ため、一番最後に割り当てられた値がputs aで出力される。いくらになるか「irb」で確認してみること。

代入できるのは10や”this is a pen”などの各種リテラルだけではなく、予め代入された変数による計算結果(a+aなど)も代入できるのである。

## 3. 値を変換しあう。

ここまでで文字と数字の表現の仕方と変数への代入の仕方について述べた。ここでは、文字と数字をお互いに変換しあう操作について述べる。というのも、例えば

```
year   = 365
day    = 24
hour   = 60
minite = 60
puts "a year = " + (year * day) + " hours"
```

などと（上は動かない）計算結果を表示したいのだが、文字と数字を「＋」するという操作はruby言語には用意されていないので、うまくいかない。文字同士を「＋」(結合)するか、数字同士を「＋」(加算)するかどちらかにしなくてはならないからだ。

```
>> 890 + 1
=> 891
>> "890" + "1"
=> "8901"
>> "890" + 1
TypeError: can't convert Fixnum into String
        from (irb):26:in `+'
        from (irb):26
        from :0
```

そこで year×day という数字を「その数字を意味する文字列」に変換すれば良い。ruby言語では、変数あるいはリテラルの後に「.to_s」(to string)をつけると文字列へ変換されたものを意味し、「.to_i」(to integer) とすれば整数値へ変換されたもの、また「.to_f」(to float) とすれば実数値へ変換されたものとして扱われる。

```
>> "890".to_i + 1
=> 891
>> "890".to_f + 1
=> 891.0
>> "890" + 1.to_s
=> "8901"
```

これらを使うと先ほどの 1 年が何時間か何分か何秒な

```
year   = 365
day    = 24
hour   = 60
minite = 60

x = year * day
puts "a year = " + x.to_s + " hours"

x = x * hour
puts "a year = " + x.to_s + " minites"

x = x * minite
puts "a year = " + x.to_s + " seconds"
```

のかを計算して表示するプログラムは、.to_s を用いて

```
gene07:~ bio$ ruby test.rb
a year = 8760 hours
a year = 525600 minites
a year = 31536000 seconds
gene07:~ bio$
```

と書ける。上をtest.rbに保存して実行してみること。

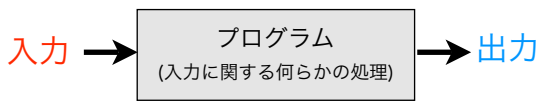
ここで次のようなことに気づく人がいるかもしれない。「puts」が画面に出力しているものは文字列だが、命令文を書くときは、数字でも文字列でも同じように扱えたということである。

```
>> puts 2006
2006
=> nil
>> puts 2006.to_s
2006
=> nil
>> puts "2006"
2006
=> nil
```

は全部同じ「2006」として表示されている。これはputsとはまず.to_sで文字列に変換してから画面に表示しているからである。putsは「**put string**」の意味である。文字自身はもちろん、数字や様々な種類のデータがこの「puts」を通して、文字列に変換されて確認できる。

## 4 . I/O (アイオー)を考える。

ここまでで数字や文字列を扱い、それを計算したり、変換したり、画面に表示したりが少しできるようになった。それでは、最初のプログラムの図を思い出そう。



さきほどの一年が何秒かという計算ではプログラムからの出力は画面の文字として得られたが、入力はプログラム中に含まれた整数リテラル（365, 24, 60など）として直接プログラム中に与えられた。では、プログラム中で（リテラルとして）表現された値によって変数に値をセットするのではなく、プログラムの外部から入力として値（数値や文字列）を与える場合はどうすればよいのだろうか。というか、そもそも、このような可変性がないならプログラムを作るメリットが少ないので、やはり大抵のある程度実用的なプログラムにおいては

1. ファイルから文字や数字を読む
2. キーボードやマウスから直接入力する
3. インターネット上のファイルから文字や数字を読む

など、外部からの入力が想定される。そこでプログラムの「**入出力(Input/Output, I/O)**」について簡単なものとして、キーボードからの入力をプログラムが受け取る場合について考える。これを我々の最初の例題としよう。

例題１：入力としてキーボードから名前（たとえばtakigawa）と年齢（たとえば39）を受け取って、この世知辛い現世に今まで何時間生き長らえたのかと、還暦まで何時間の余地があるのか、を画面に

```
takigawa san ( 341640 hours life )
kanreki made :
    nokori  183960 hours
```

と計算して表示するプログラムを作ってみよう。
（1年=365日、1日=24時間、還暦60才として概算）

出力はputsを使ってできるようになったので、まず入力を受け取る部分について述べる。名前を受け取って表示するには「gets」を使う。「gets」は**キーボードから入力された文字列の入った変数のよう**に扱える。

```
>> name = gets
takigawa
=> "takigawa\n"
>> puts name
takigawa
=> nil
>> puts name + " san"
takigawa
 san
=> nil
```

「takigawa」というのはキーボードからの入力である。ここで一つ問題がある。というのは"\n"などという文字が入っていることである。これは「Enter」キーを押して改行したことを意味する。従って、`name + " san"` は「takigawa san」ではなく、改行されてしまっている。このような改行を取り除くため、入力したのち、文字列から改行等を削除する必要がある。これには文字列変数やリテラルに.chompをつければ良い。

chompは最後に改行がなければ何もしないし、改行がたくさんあってもひとつしか取り除かない。

```
>> "retrospection\n".chomp
=> "retrospection"
>> "retrospection".chomp
=> "retrospection"
>> "retrospection\n\n".chomp
=> "retrospection\n"
```

今0才なら還暦まであと60年と見なすことにすると、以上の知識から例題の解答は例えば以下のようなものになるだろう。

```
puts "Input your name."
name = gets.chomp

puts "Input your age."
age  = gets.to_i

passed = (age * 365 * 24).to_s
left   = ((60-age) * 365 * 24).to_s

puts
puts name + " san ( " + passed + " hours life )"
puts "kanreki made:"
puts "   nokori " + left + " hours"
```

これを「kanreki.rb」に保存して実行してみる。

```
gene07:~ bio$ ruby kanreki.rb
Input your name.
takigawa
Input your age.
39

takigawa san ( 341640 hours life )
kanreki made:
   nokori 183960 hours
gene07:~ bio$
```

**[蛇足]** ちなみに名前が「myname.txt」というファイルの一行目に「Toshiaki Katayama」と改行つきで書いてあるとすると、上のgets.chompの前にこのファイルを開いてそこを入力とするように付け足すだけである。実際、もしファイルが存在すれば、キーボードからの入力を処理するのも、ファイルからの入力を処理するのも、ネットワーク上の入力を処理するのも、あまり違いはない。

```
>> name = gets.chomp
Masumi Itoh
=> "Masumi Itoh"

>> name = File.open("myname.txt").gets.chomp
=> "Toshiaki Katayama"

>> require "open-uri"
=> true
>> name = open("http://web.kuicr.kyoto-u.ac.jp/
~takigawa/myname.txt").gets.chomp
=> "Ichigaku Takigawa"
```

## 5. メソッドを使う。

ところで今までの例題で以下のようなルールがありそうなことにお気づきだろうか。

**リテラルや変数名のうしろに「.なんとか」をつけるとどうも「なんとか」に相当する処理が行われる**

chompとかto_sなどである。rubyにおいてはリテラルや変数に格納された値はすべてこのような操作命令を受け付ける。可能な命令はその値の種類によって異なる。この命令のことを「メソッド」という。整数のメソッドと、文字列のメソッドは異なる。

irbを用いて整数や実数や文字列にどのような命令を送る事ができるのか、もっと見てみることにしよう。実は、 ベタなことに(?)どんなメソッドがあるのかをみるメソッド「.methods」がある。

```
gene07:~ bio$ irb --simple-prompt -r irb/completion
>> 12.methods
=> ["to_a", "%", "<<", "respond_to?", ">>", "divmod",
"&", "type", "integer?", "chr", "protected_methods",
"eql?", "to_sym", "*", "instance_variable_set", "+",
"is_a?", "truncate", "hash", "send", "to_s", "-",
   :   省略
 "div", "times", "object_id", "=~", "singleton_meth-
ods", "equal?", "succ", "taint", "id2name", "[]",
"frozen?", "instance_variable_get", "kind_of?",
"round"]
>>
```

これはrubyのどんな対象についても使えるはずである。今、上の例ではリテラル 12 に対して利用できるメソッドの名前がなにやらたくさん表示されたことと思う。これのそれぞれがどんな命令なのかはヘルプ等を参照すれば良い。変数に対しても同様である。

```
>> a = -199.7
=> -199.7
>> a.methods
=> ["%", "to_a", "respond_to?", "divmod", "nan?",
   : 省略
"=~", "singleton_methods", "equal?", "taint", "fro-
zen?", "instance_variable_get", "kind_of?", "round"]
>> a.to_s
=> "-199.7"
>> a.abs
=> 199.7
>> a.round
=> -200
>> a.integer?
=> false
```

「どんなメソッドがあるのか」はその値自身がどんな種類の対象なのか（実数なのか文字列なのか何なのか）によってあらかじめ決まっている。この種類のことを「クラス」と言う。その値がどんなクラスのものかを確認するためのメソッド「.class」も用意されている。

```
>> "Hello".class
=> String
>> 99.9.class
=> Float
>> 99.class
=> Fixnum
>> -98.class
=> Fixnum
>> (1+2+5.5).class
=> Float
```

それぞれのクラスにどんなメソッドが決められているかは、rubyのドキュメントやヘルプ、例えば、

**http://www.ruby-doc.org/core/**

の「**Classes**」というところで該当するクラスの名前をクリックしてみると良い。たとえば「Fixnum」に用意されているメソッド「.zero?」の記述は

**fix.zero? => true or false**

Returns `true` if *fix* is zero.

となっていて、「trueかfalseかという値が得られること」と、fix.zero?と書いたときに「fixがゼロだったらtrueになること」が書かれている。

このヘルプにも（分かりにくいが）書いてあることには「+」（加算）などもまたメソッドであり、これは今までのものとちがって、さらに一つ数字を必要とする。このように、さらにいくつか情報を与えて機能するようなタイプのメソッドもある。**括弧はわりと省略できる**。

```
>> a = 99
=> 99
>> a.+ 1
=> 100
>> a + 1
=> 100
>> a.+(190)
=> 289
```

しかし、普通は単に「a+1」と書いておく方が良い。ちなみに「irb」に「-r irb/completion」をつけてきたが、これはメソッドの名前を「Tabキー」で補完するためのものである。上の例で「a.」の後「Tabキー」を押して候補が補完されるのを見てみること。

またカッコ付けによる順番付けももちろんできるが、基本的には**メソッドは左から順番に解釈される**。

```
>> "hello".reverse.capitalize
=> "Olleh"
>> "hello".class.class
=> Class
>> (puts "hello").class
hello
=> NilClass
>> nil.class.methods.length
=> 72
```

**【蛇足】**この短時間でまさかとは思うが「.methods」や「.class」というメソッドがさっきのヘルプらしきところに見当たらなかったことに気づいてしまったひとはいないだろうか。たとえばFixnumのページのMethodsというところにそれがないことに。

実はクラス（どんなメソッドを受け付けるデータ種なのか）というのは排他的にわかれているわけではなくて階層的なのである。

万物 ⊃ 生き物 ⊃ 人類 ⊃ 日本人 ⊃ ぼく

みたいな具合に

万物 ⊃ 数字 ⊃ 整数

だとか

万物 ⊃ 数字 ⊃ 実数

だとか概念は階層的になっていて、「数字」レベルで通じるメソッド（使える命令）はそれより下位のレベルの「整数」でも「実数」でも使えるという具合である。

Fixnumの場合、さきほどのページでParentというところを見るとIntegerとなっていて、IntegerのParentはNumericになっていて、NumericのParentを見るとObjectになっていて、ObjectにはParentの記述がない。

つまりこのヘルプのようなものによれば「Object ⊃ Numeric ⊃ Integer ⊃ Fixnum」となっている上にObjectなるクラスには上位のクラスはなさそうだ。「.class」や「.methods」はObjectクラスに用意されているメソッドである。

それぞれのデータ種がどんなメソッドを受け付けるかをこのような概念の階層で整理する考え方はしばしば「**オブジェクト指向**」と呼ばれている。

このような概念の抽象化・一般化とはひとつとして同じものはないように見えるいろいろな物を、どこか共通する性質を見いだして分類していくという作業によって、細かく詳細で無尽蔵の豊かなバリエーションを無視してその情報を捨て去ることである。

それでは、すこしヤヤコしい事柄を説明したりして頭がとんでしまうとアレなので、まずここまでで得たことを整理しておこう。

0節：プログラムが上から順番に翻訳され、コンピュータに伝えられること、irbによって一ステップずつ試してみること、を学んだ。

1節：「値」をどうやって文字データを用いて書き表すのかについて学んだ。それはリテラルと呼ばれた。

2節：「値」を記録してとっておくために「変数」という名前のついた入れ物を使うこと、変数に値を（リテラルで値を表現して）代入すること、を学んだ。

3節：文字列と数字という種類のデータを変換する方法を学んだ。

4節：データを実際に外部から受け取る方法、特にキーボードから入力された文字を文字列として取得する方法について学んだ。

5節：それぞれの「値」が、決められたメソッドを持っていること、それを使って「値」に命令をおくれることを学んだ。

## 6. 複数のものをまとめて扱う。

この節では、**複数の値をまとめて扱う方法**について述べる。５人の仲間のうちで、ひとりだけがパーティーの食材買い出しに行かねばならず、それを公平に決めるため、以下の具体例から考えることにしよう。

例題２：入力としてキーボードから人名を５個受け取って、そのうちどれかひとつだけをランダムに選んで画面に表示する。

この問題においてはひとまず５個の人名を一旦すべてどこかに記録しておかねばならない。このような目的のためには「配列 (array)」というものを用いるのが便利である。配列は値を複数並べた集まりである。

```
>> people = ["Takigawa", "Katayama", "Itoh", "Na-
gasaki", "Tanaka"]
=> ["Takigawa", "Katayama", "Itoh", "Nagasaki",
"Tanaka"]
>> people[0]
=> "Takigawa"
>> people[1]
=> "Katayama"
>> people[2]
=> "Itoh"
```

ここでは５つの文字列をカッコ「[ ]」とカンマ「,」で区切ってひとまとまりのリテラル

```
["Takigawa", "Katayama", "Itoh", "Nagasaki", "Tanaka"]
```

として、変数peopleに代入している。それぞれの要素は

```
people[0], people[1], people[2], people[3], people[4]
```

として取り出すことができる。また、people[0]はこういう名前の変数として通常の変数と同じように扱うことができる。また、**1から5ではなくて0から4の数字で呼び出す**のは間違いではなくて仕様である。

ここでは文字列リテラルだけ並べてみたが、数字が混入していても、配列の要素に配列があっても良い。

```
>> a = [1, "22", 9.0, "Takigawa", [1, 3]]
=> [1, "22", 9.0, "Takigawa", [1, 3]]
>> a.class
=> Array
>> a[0]
=> 1
>> a[4]
=> [1, 3]
>> a[4][0]
=> 1
```

以下に配列データで決められているメソッドの例をいくつか並べるので、挙動と名前から何が起きているかを類推したりしてみるとよいかもしれない。

```
>> a = [1,5,-9,2,3]
=> [1, 5, -9, 2, 3]
>> a.length
=> 5
>> a.empty?
=> false
>> a + a
=> [1, 5, -9, 2, 3, 1, 5, -9, 2, 3]
>> a.include? -9
=> true
>> a.first
=> 1
>> a.reverse
=> [3, 2, -9, 5, 1]
>> a[2,5]
=> [-9, 2, 3]
>> a[2..5]
=> [-9, 2, 3]
>> a.last
=> 3
>> a.sort
=> [-9, 1, 2, 3, 5]
>> a.push 5
=> [1, 5, -9, 2, 3, 5]
>> a
=> [1, 5, -9, 2, 3, 5]
>> a.uniq
=> [1, 5, -9, 2, 3]
>> a.pop
=> 5
>> a
=> [1, 5, -9, 2, 3]
>> a.clear
=> []
```

空の配列データを作るにはArray.newを用いる。またそこへ要素を足していくには.push("x")を用いる。逆にとりだすには.popを使う。

```
>> a = Array.new
=> []
>> a.push 1
=> [1]
>> a.push 33
=> [1, 33]
>> a.push 45
=> [1, 33, 45]
```

ここでもとの問題に戻ると、結局入力された名前を配列に格納したあと、0から4の数字をどれかランダムに選んで変数 i に代入して、people[i]を返せば良い。

```
people = Array.new

people.push "takigawa"
people.push "katayama"
people.push "itoh"
people.push "nagasaki"
people.push "tanaka"

i = rand(people.length)

puts people[i]
```

ここで、rand(n)は「0からn−1までの整数をランダムに選ぶ」命令である。

例題では直接リテラルをpeople.pushしているが、ここをgetsにすることでキーボードからこれらの人名を得ることができる。しかし、それだと、

```
people = Array.new

people.push gets.chomp
people.push gets.chomp
people.push gets.chomp
people.push gets.chomp
people.push gets.chomp

i = rand(people.length)

puts people[i]
```

と同じ文が５回もある。これはいかがなものだろうか。手作業を簡易化するのがプログラミングのメリットだったはずなのに同じ文を５回も入力するのか！まさか！

そこで次のテーマに移らねばならないわけだ。

## 7. 繰り返す。

我々はしばしばある手順を繰り返す必要に迫られてきた。今の場合命令「people.push gets.chomp」を5回繰り返したいのだった。このような場合にはrubyでは以下のように書けば良い。

```
5.times do
   people.push gets.chomp
end
```

あるいは

```
5.times { people.push gets.chomp }
```

である。**doとendにはさまった部分**、あるいは**{ }に挟まった部分**が5回繰り返される。従って、例題の最終的な答えとしては以下のようなシンプルなものになる。

```
people = Array.new
5.times { people.push gets.chomp }
puts people[rand(people.length)]
```

何をやっているか把握できたら実際に実行してみよう。

```
>> people = Array.new
>> 5.times { people.push gets.chomp }
takigawa
katayama
itoh
nagasaki
tanaka
>> puts people[rand(people.length)]
tanaka
```

次に、似たような次の問題を考えてみよう。

例題３：入力としてキーボードから数字を５個受け取って、小さい順番に画面に表示する。

まず配列に格納された数字を小さい順番並び替えるには.sortというメソッドを使う。小さい順に並べ替えた配列を作るところまでやってみよう。

```
>> a = Array.new
=> []
>> 5.times { a.push gets.to_f }
99.7
-3
2.4
687
77.0
=> 5
>> a.sort
=> [-3.0, 2.4, 77.0, 99.7, 687.0]
>> a
=> [99.7, -3.0, 2.4, 687.0, 77.0]
>> a = a.sort
=> [-3.0, 2.4, 77.0, 99.7, 687.0]
>>
```

ここでは a = [99.7, -3.0, 2.4, 687.0, 77.0] という配列を作って、それを小さい順番にならべかえたもの(a.sort)をふたたび変数aに代入している。

「a = a.sort」は「a.sort!」とも書く事ができる。

変数aをそのままputsしてももちろん良いのだが、配列の中の数字を一つずつ順番にputsしたいとしよう。このときは、aの要素ひとつひとつについて前から順番にputsしなくてはいけない。これも一つの繰り返しである。

まず、今さっき習った.timesを使って、iを0,1,2,3,4と順番に変えてputs a[i]する方法を考えてみよう。

```
a = [-3.0, 2.4, 77.0, 99.7, 687.0]
i = 0
a.length.times do
   puts a[i]
   i = i + 1
end
```

これで配列aの要素を順番に画面に表示することができる。しかし、実は.timesではそのあとの処理で内部に0から一つずつ増える値を使うこともできるので、以下のようにもうすこし簡単にすることもできる。

```
a = [-3.0, 2.4, 77.0, 99.7, 687.0]

a.length.times do |i|
   puts a[i]
end
```

このようにある変数 i を 0 から a.length−1 まで+1ずつしながらの繰り返しは次のようにも書く事ができる。

```
a = [-3.0, 2.4, 77.0, 99.7, 687.0]
n = a.length
for i in 0..n-1 do
   puts a[i]
end
```

また、i番目の要素 a[i] として出さず、要素自体を各々変数xに代入し、puts xするという方法も考えられる。

```
a = [-3.0, 2.4, 77.0, 99.7, 687.0]

for x in a do
   puts x
end
```

だいぶシンプルで読みやすくなったようだ。しかし、このようなケースでは大半のrubyユーザは.eachというメソッドを使って以下のように書くのを好むようである。

```
a = [-3.0, 2.4, 77.0, 99.7, 687.0]
a.each { |x| puts x }
```

もしくは

```
a = [-3.0, 2.4, 77.0, 99.7, 687.0]

a.each do |x|
   puts x
end
```

である。まとめると、例題に対する答えとしては次のようなものが考えられるだろう。

```
a = Array.new
5.times { a.push gets.to_f }
a.sort!
a.each { |x| puts x }
```

さて、ここでの問題では

(1) 空の配列を作り変数aに代入。　a = Array.new
(2) 必要なデータを順番にそれに押し込む。 a.push x
(3) a.each { |x| xを何かする } として、配列aの要素xそれぞれについて何か行う。

という流れであった。ここで「xを何かする」場合には「x = 1」とか代入することはできないことに注意が必要である。以下のような記述は**うまく動かない**。

```
a = [ 1, 2, 3, 4 ]
a.each { |x|  x = 100 }
```

配列に用意されているメソッドを使うと、aの要素を書き換えることもできるが、それはeachではない。これを次の例題で見よう。

例題4：整数nをキーボードから受け取って配列
a = [ $1^2$, $2^2$, $3^2$, $4^2$, … , $n^2$ ] を作る。

これにもいろいろな方法が考えられるが以下のどちらかが直接的であろう。Array.new(n)は大きさがnで要素が全部nilの配列を作る（ちなみに以下の例だとrubyではArray.newのままでも動くが）。

```
a = Array.new
n = gets.to_i

for i in 1..n do
   a.push i**2
end
```

```
n = gets.to_i
a = Array.new n

for i in 0..n-1 do
   a[i] = (i+1)**2
end
```

これは.mapもしくはmap!というメソッドを用いると次のようにも書く事ができる。例のmapでは**配列のある要素の値が i とするとその値を i**2 に置き換えている**。mapメソッドは**それぞれの要素への作業の結果を記録しておいてそれらをまとめて配列にしたものを作る**。

```
n = gets.to_i
a = (1..n).map { |i| i**2 }
```

※ 厳密にいうと(1..n)は配列ではない。従ってここまでの説明でいうと.to_aメソッド(to array)を使って

```
n = gets.to_i
a = (1..n).to_a
a.map! { |i| i**2 }
```

などと書く方が適当かもしれない。`a.map! { |i| i**2 }`の部分は `a = a.map { |i| i**2 }` と同じである。

## 8. 場合分けを行う。

例題５：「5 6.2 3.3 -1.7 9」のような文字列をキーボードから受け取って、それらの数字の平均値より大きい数字だけからなる配列を作る。

今までは数字を５つそれぞれgetsで受け取ってそれを配列に記録してきた。ここでは**いくつの数字が入力されるかわからないが**それは空白文字を区切りとしていっぺんに入力されるものとしよう。データファイルなどは典型的にはこんなようなかたちで保存されていることも多いのではないだろうか。

まず文字列からひとつひとつの数字を切り出して、配列のかたちにかえてやる必要がある。文字列は.splitというメソッドを使うことで配列に分解することができる。

```
>> line = "5 6.2 3.3 -1.7 9"
=> "5 6.2 3.3 -1.7 9"
>> line.split
=> ["5", "6.2", "3.3", "-1.7", "9"]
>> line.split "."
=> ["5 6", "2 3", "3 -1", "7 9"]
>> line.split "3"
=> ["5 6.2 ", ".", " -1.7 9"]
>> line.split "-"
=> ["5 6.2 3.3 ", "1.7 9"]
>> "ab:cd:ef:gh:".split ":"
=> ["ab", "cd", "ef", "gh"]
```

splitはその後に何も追加情報の文字がない場合はスペースなどの空白文字で分割を行うが、.split("m")として追加の文字が与えられた場合、その文字 "m" を「区切り文字」として文字列配列への分解を行う。

この新しい知識といままでの知識をあわせれば数字の配列を作るところまでは書く事ができるだろう。

```
a = gets.split.map { |x| x.to_f }
```

次にこの配列 a の要素の平均値を求める必要がある。すなわち「要素の総和/要素の数」を計算して変数avgに記録することにする。なお、このように**割り算が入るときは整数同士の計算にならないように注意する**こと（必要なら.to_fする）。以下が直接的ではないだろうか。

```
sum = 0.0
a.each { |x| sum = sum + x }
avg = sum / a.length
```

ちなみにinjectというメソッドの利用も可能である。

```
avg = a.inject(0) { |sum, x| sum + x } / a.length
```

あとは配列 a の要素のうちこの値avgよりも大きい数字だけ残した配列を作れば良い。これは

```
if 条件 then
   操作１
else
   操作２
end
```

という形で、条件が満たされるとき(trueのとき)操作１を行い、そうでないとき操作２を行うという「条件分岐」の構文で記述するのが直接的である。

これを用いれば次のようになるだろう。

```
result = Array.new
a.each do |x|
   if x > avg then
      result.push x
   end
end
```

しかし、配列 a の中で「条件」を満たすものだけをとってくる操作は.selectとして、「条件」を満たすものだけを消し去る操作は.delete_ifとして用意されている。

```
result = a.select { |x| x > avg }
result = a.delete_if { |x| not (x > avg) }
```

以上をまとめると以下のようなものになる。

```
a = gets.split.map { |x| x.to_f }

sum = 0.0
a.each { |x| sum = sum + x }
result = a.select { |x| x > sum/a.length }
```

## 9. 対応付けを行う。

例題6：

seq="atgaacgtgctccacgactttgggatccagtcg"

からそれぞれの数を以下のように棒グラフのようにプロットする。

```
a ******* 7
t ******** 8
c ********* 9
g ********* 9
```

直接的には文字列では.count("x")で "x" の個数をカウントできることを使って次のように書く事ができる。

```
seq = "atgaacgtgctccacgactttgggatccagtcg"

for x in "atcg".split('') do
   n = seq.count x
   puts x + " " + ( "*" * n ) + " " + n.to_s
end
```

しかし、上のものはあらかじめ何種類のどのような文字が出てくるか分かっている場合（この場合a,t,c,gの4種類だけと事前に分かっている）の処理である。ここではそれが分からない場合について考える。

それには「ハッシュ (hash)」と呼ばれるものを使うと便利である。ハッシュとは辞書のようなものである値aとある値bを関連づけるものである。しばしば他の言語では辞書型と呼ばれたり、連想配列と呼ばれたりする。

ハッシュをhとすると、「 h[キー] = 値」のように対応付けすれば、キー→値の関係を覚えておいてくれるのである。必要ならば h[キー] で対応する値が何だったか思い出せる。値もキーも数値でなくとも良い。空のハッシュはHash.newで作る。

```
>> h = Hash.new
=> {}
>> h["hello"] = "konnichiwa"
=> "konnichiwa"
>> h["goodbye"] = "sayounara"
=> "sayounara"
>> h["see you"] = "osakini"
=> "osakini"
>> h
=> {"see you"=>"osakini", "hello"=>"konnichiwa",
"goodbye"=>"sayounara"}
>> h.key? "good morning"
=> false
>> h.key? "goodbye"
=> true
>> h.keys
=> ["see you", "hello", "goodbye"]
>> puts h["goodbye"]
sayounara
=> nil
```

ここでHash.newに初期値を与えておくと、キーがないものを調べようとしたとき、その初期値が返される。

```
>> h = Hash.new
=> {}
>> h[55] = -2
=> -2
>> h[44]
=> nil
>> hh = Hash.new 900
=> {}
>> hh[55] = -2
=> -2
>> hh[44]
=> 900
```

このことを利用すれば以下のように書く事ができる。

```
seq = "atgaacgtgctccacgactttgggatccagtcg"

counter = Hash.new 0
seq.split('').each { |x| counter[x] += 1 }

counter.each do |x, n|
   puts x + " " + ("*" * n) + " " + n.to_s
end
```

※ 実は文字列には.scanというメソッドがあるのだが。

## 10. これから。

ここでは網羅的には扱えなかったので自習の際には書籍などを参考にすること。おつきあい本当にありがとう。

## 補　足

　実習では体験を趣旨としているのでどうしても言語の全体について詳細には解説ができなかった。以下に自習するためのウェブ上の様々な情報源を載せておくので、実際にrubyを自分のパソコンでも使ってみようと思った方やさらに詳しく学びたい方は適宜参照のこと。

### ❶ ruby-lang.org

公式ホームページ（日本語）

　http://www.ruby-lang.org/ja/

リファレンスマニュアル（日本語）

　http://www.ruby-lang.org/ja/man/

紹介、リリース情報、ダウンロード、インストールの仕方の解説、FAQ、チュートリアルなど、rubyの情報が集まっている。

### ❷ ruby-doc.org

Rubyに関するドキュメンテーション、案内（英語）

　http://www.ruby-doc.org/

Core API

　http://www.ruby-doc.org/core/

標準ライブラリドキュメンテーション

　http://www.ruby-doc.org/stdlib/

### ❸ Programming Ruby

ピッケル本 (the pickaxe)
「プログラミングRuby—達人プログラマガイド」

　Rubyの網羅的な解説（英語）

http://www.ruby-doc.org/docs/ProgrammingRuby/

http://phrogz.net/ProgrammingRuby/

### ❹ Learn to program

（Rubyを採用している）
とても分かりやすいプログラミング入門（英語）

　http://pine.fm/LearnToProgram/

その邦訳（日本語）
　http://chem.tf.chiba-u.jp/~shin/tutorial/

### ❺ Why's (poignant) guide to Ruby

Rubyに関するとても変わった入門解説（英語）

　http://poignantguide.net/ruby/

### ❻ るびま

日本Rubyの会によるRubyist Magazine（日本語）

　http://jp.rubyist.net/magazine/

Rubyではじめるプログラミング

http://jp.rubyist.net/magazine/?0002-FirstProgramming

## 関　連

### ❶ Refe

日本語マニュアルをコマンドラインで引くためのツール

http://i.loveruby.net/ja/prog/refe.html

### ❷ RAA(Ruby Application Archive)

Rubyの様々なライブラリを集めたサイト

http://raa.ruby-lang.org/

### ❸ RubyForge

多くのRuby関連プロジェクトの開発・配布サイト

http://www.rubyforge.net/

### ❹ Ruby on Rails

David Heinemeier Hansson 氏によるRubyでウェブアプリケーションを作るためのフレームワーク。

http://rubyonrails.com/

## 書　籍

**❶ たのしいRuby—Rubyではじめる気軽なプログラミング**
高橋 征義, 後藤 裕蔵, まつもと ゆきひろ (著)
**出版社:** ソフトバンククリエイティブ
**ISBN:** 4797314087 ; (2002/03)

**❷ プログラミングRuby—達人プログラマーガイド**
David Thomas, Andrew Hunt (原著), 田和 勝 , まつもと ゆきひろ (翻訳)
**出版社:** ピアソンエデュケーション
**ISBN:** 4894714531 ; (2001/09)

**❸ オブジェクト指向スクリプト言語 Ruby**
まつもと ゆきひろ, 石塚 圭樹 (著)
**出版社:** アスキー
**ISBN:** 4756132545 ; (1999/10)

**❹ Rubyレシピブック 268の技**
青木 峰郎, 後藤 裕蔵, 高橋 征義, まつもと ゆきひろ (著)
**出版社:** ソフトバンククリエイティブ
**ISBN:** 4797324295 ; (2004/05)