第2章 便利なオブジェクト

第1章では、Rubyで扱う基本的なデータとして「文字列」と「数値」を取りあげましたが、Rubyで扱えるオブジェクトはこれだけではありません。多くのRubyのプログラムでは、もっと複雑なデータを扱うことになるでしょう。

Rubyでアドレス帳を作ることを考えてみます。アドレス帳に必要な項目は、

- 名前
- ふりがな
- 郵便番号
- 住所
- 電話番号
- メールアドレス
- SNSのID
- 登録日

といったところでしょうか。これらはいずれも文字列で表現できそうです。

これらの項目をひとまとめにすることで、1人分の情報になります（図2.1）。さらに、交友関係の人たちの情報が集まって、アドレス帳全体のデータができあがるわけです。

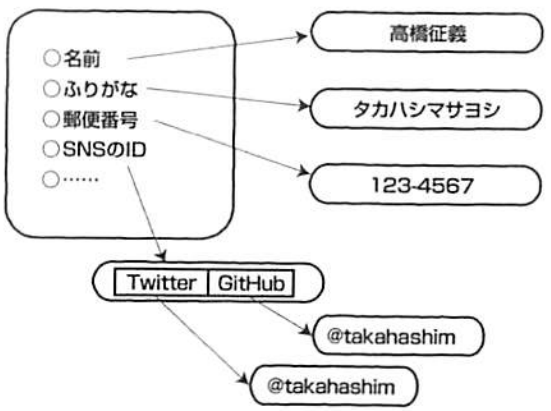

図2.1 項目を結びつけてひとまとめにする

図2.1のようにデータとデータを合わせた1つのデータを表すには、これまでに紹介した「文字列」や「数値」といった単純なオブジェクト以外に、データの集まりを表現するデータ構造が必要になります。

この章では、「配列」と「ハッシュ」というデータ構造を紹介します。また、「正規表現」という、文字列処理に使われるオブジェクトも紹介します。

メモ　配列やハッシュのようにオブジェクトを格納するオブジェクトを、コンテナやコレクションといいます。

配列・ハッシュ・正規表現はさまざまな場面で使われますが、より詳しい説明はあとの章で行うことにして、ここではごくおおまかにイメージをつかむことを目的に解説します。

2.1 配列(Array)

配列は「いくつかのオブジェクトを順序つきで格納したオブジェクト」として、もっとも基本的でよく使われるコンテナです。「配列オブジェクト」「Arrayオブジェクト」などと呼ばれることもあります。

2.1.1 配列を作る

新しい配列を作るには、要素をカンマ区切りで並べて、「[]」で全体を囲みます。まずは簡単な、文字列の配列を作ってみましょう。

```
names = ["小林", "林", "高野", "森岡"]
```

この例では、namesという配列オブジェクトが作られました。各要素として「"小林"」「"林"」「"高野"」「"森岡"」という4つの文字列を格納しています。図示すると図2.2のようになります。

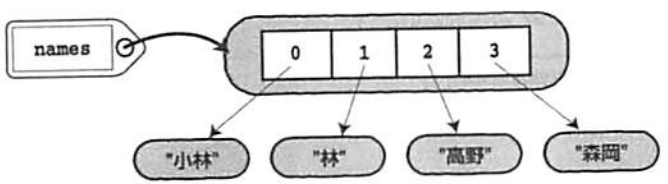

図2.2 配列オブジェクト

2.1.2 配列オブジェクト

配列の要素となるオブジェクトが決まっていない場合には、「[]」とだけ書くと、空の配列オブジェクトができます。

```
names = []
```

これ以外にも配列の作り方はいくつかあります。詳しくは「第13章　配列(Array)クラス」で説明します。

2.1.3 配列からオブジェクトを取り出す

配列に格納されたオブジェクトには、位置を表す番号であるインデックスがつきます。このインデックスを使って、オブジェクトを格納したり、取り出したりできます。

配列の要素を取り出すには、

配列名[インデックス]

という構文を使います。たとえば、namesという名前の配列オブジェクトを次のように作ったとします。

```
names = ["小林", "林", "高野", "森岡"]
```

配列namesの最初の要素である「小林」という文字列を取り出すには、

```
names[0]
```

と書きます。そのため、

```
print "最初の名前は", names[0], "です。\n"
```

という文を実行すると、

```
最初の名前は小林です。
```

と表示されます。同様に、names[1]は"林"、names[2]は"高野"になります。

実行例

```
> irb --simple-prompt
>> names = ["小林", "林", "高野", "森岡"]
=> ["小林", "林", "高野", "森岡"]
>> names[0]
=> "小林"
>> names[1]
=> "林"
>> names[2]
=> "高野"
>> names[3]
=> "森岡"
```

配列のインデックスは0から始まります。1ではありません。ですから、a[1]と書くと、aという配列オブジェクトの先頭の要素ではなく、2番目の要素が返ってきます。慣れるまでは間違いやすいかもしれません（慣れていても間違いやすいところです）。注意してください。

Windowsのコマンドプロンプトで日本語入力モードに切り替えるには、[半角／全角]キーを押します。

2.1.4　配列にオブジェクトを格納する

すでにある配列に、新しいオブジェクトを格納することもできます。

配列の要素の1つを別のオブジェクトと置き換えるには、

配列名[インデックス] = 格納したいオブジェクト

という構文を使います。先ほどの配列namesを使ってみましょう。先頭に"野尻"という文字列を格納するには、

```
names[0] = "野尻"
```

と書きます。たとえば、次のように実行すると、namesの最初の要素が「野尻」になることがわかります。

実行例

```
> irb --simple-prompt
>> names = ["小林", "林", "高野", "森岡"]
=> ["小林", "林", "高野", "森岡"]
>> names[0] = "野尻"
=> "野尻"
>> names
=> ["野尻", "林", "高野", "森岡"]
```

オブジェクトの格納先として、オブジェクトのまだ存在しない位置を指定すると、配列の大きさが変わります。Rubyの配列は、必要に応じて自動的に大きくなります。

実行例

```
> irb --simple-prompt
>> names = ["小林", "林", "高野", "森岡"]
=> ["小林", "林", "高野", "森岡"]
>> names[4] = "野尻"
=> "野尻"
>> names
=> ["小林", "林", "高野", "森岡", "野尻"]
```

2.1.5　配列の中身

配列の中には、どんなオブジェクトも要素として格納できます。たとえば、文字列ではなく数値の配列も作れます。

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

1つの配列の中に、複数の種類のオブジェクトを混ぜることもできます。

```
mixed = [1, "歌", 2, "風", 3]
```

ここでは例を挙げませんが、「時刻」や「ファイル」といったオブジェクトも、配列の要素にできます。

2.1.6　配列と大きさ

配列の大きさを得るには、sizeメソッドを使います。たとえば、配列arrayに対して次のように使います。

```
array.size
```

sizeメソッドを使って、先ほどの配列オブジェクトnamesの大きさを調べてみましょう。

実行例

```
> irb --simple-prompt
>> names = ["小林", "林", "高野", "森岡"]
=> ["小林", "林", "高野", "森岡"]
>> names.size
=> 4
```

このように、配列の大きさが、数値として返ってきます。

2.1.7　配列と繰り返し

「配列の要素をすべて表示したい」とか、「配列の要素のうち、ある条件に当てはまる要素についてはxxメソッドを、当てはまらない要素についてはyyメソッドを適用したい」といったときには、配列の要素すべてにアクセスする方法が必要です。

Rubyには、このためのメソッドとして、eachメソッドが用意されています。eachメソッドは、第1章でも少し触れたように「イテレータ」というメソッドの1つです。

eachメソッドは、次のように使います。

```
配列.each do |変数|
  繰り返したい処理
end
```


eachのすぐ後ろの「do ～ end」で囲まれている部分を**ブロック**といいます。そのため、eachのようなメソッドは、**ブロックつきメソッド**とも呼ばれます。ブロックにはいくつかの処理をまとめて記述することができます。

ブロックの冒頭には「**|変数|**」という部分があります。eachメソッドは、配列から要素を1つずつ取り出して、「**|変数|**」で指定された変数に代入して、ブロックの中のメソッドを繰り返し実行していきます。

実際に使ってみましょう。配列namesにあるすべての要素を順番に表示してみます。

実行例

```
> irb --simple-prompt
>> names = ["小林", "林", "高野", "森岡"]
=> ["小林", "林", "高野", "森岡"]
>> names.each do |n|
?>   puts n
>> end
小林
林
高野
森岡
=> ["小林", "林", "高野", "森岡"]
```

do～endのように複数行にまたがる場合は、endが入力されるまで実行されません

putsメソッドの実行結果

eachメソッドの戻り値

|n|となっている部分の変数nには、繰り返しのたびに配列namesの要素が代入されます(図2.3)。

図2.3　繰り返しによるnの変化

配列にはeachメソッドのほかにもブロックを使うメソッドがたくさん用意されています。配列の要素をまとめて処理する場合によく使います。詳しくは「13.6　配列の主なメソッド」(p.256)で取りあげます。

2.2 ハッシュ(Hash)

ハッシュ（Hash）もよく使われるコンテナです。ハッシュでは文字列やシンボルなどをキーにしてオブジェクトを格納します（図2.4）。

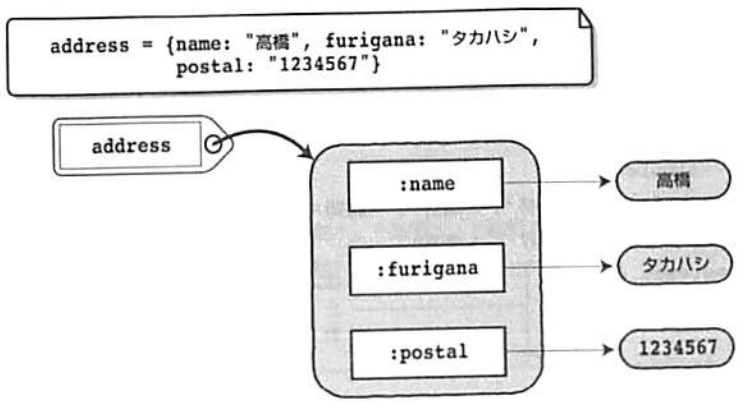

図 2.4 ハッシュ

2.2.1 シンボルとは

シンボル（Symbol）というのは、文字列に似たオブジェクトで、Rubyがメソッドなどの名前を識別するためのラベルをオブジェクトにしたものです。

シンボルは、先頭に「:」をつけて表現します。

```
sym = :foo     # これがシンボル「:foo」を表す
sym2 = :"foo" # 上と同じ意味
```

シンボルと同様のことはたいてい文字列でもできます。ハッシュのキーのように単純に「同じかどうか」を比較するような場合は、文字列よりも効率がよいことが多いので、シンボルがよく使われます。

なお、シンボルと文字列はそれぞれ互いに変換できます。シンボルにto_sメソッドを使えば、対応する文字列を取り出せます。逆に、文字列にto_symメソッドを使えば、対応するシンボルを得られます。

実行例

```
> irb --simple-prompt
>> sym = :foo
=> :foo
>> sym.to_s        # シンボルを文字列に変換
=> "foo"
>> "foo".to_sym    # 文字列をシンボルに変換
=> :foo
```

2.2.2 ハッシュを作る

新しいハッシュの作り方は、配列の作り方にちょっと似ています。配列と違うのは、「[]」の代わりに「{}」で囲むところです。また、ハッシュでは、オブジェクトを取り出すためのキーと、そのキーと対応させるオブジェクトを「**キー => オブジェクト**」という形式で指定します。キーにはシンボル、文字列、数値がよく使われます。

```
song = { :title => "Paranoid Android", :artist => "Radiohead" }
person = { "名前" => "高橋", "仮名" => "タカハシ" }
mark = { 11 => "Jack", 12 => "Queen", 13 => "King" }
```

とりわけシンボルがよく用いられるため、専用の短い書き方が用意されています。次の2つは同じ意味です。

```
person1 = { :name => "後藤", :kana => "ゴトウ" }
person2 = { name: "後藤", kana: "ゴトウ" }
```

2.2.3　ハッシュの操作

ハッシュからオブジェクトを取り出したり、オブジェクトを格納したりする方法も、配列にそっくりです。ハッシュに格納されたオブジェクトを取り出すには、次の構文を使います。

```
ハッシュ名[キー]
```

また、オブジェクトを格納するには次の構文を使います。

```
ハッシュ名[キー] = 格納したいオブジェクト
```

配列と違って、キーには数値以外のオブジェクトも使えます。シンボルをキーにしたハッシュを操作してみましょう。

実行例

```
> irb --simple-prompt
>> address = {name: "高橋", furigana: "タカハシ"}
=> {:name=>"高橋", :furigana=>"タカハシ"}
>> address[:name]
=> "高橋"
>> address[:furigana]
=> "タカハシ"
>> address[:tel] = "000-1234-5678"
=> "000-1234-5678"
>> address
=> {:name=>"高橋", :furigana=>"タカハシ", :tel=>"000-1234-5678"}
```

2.2.4　ハッシュの繰り返し

eachメソッドを使って、ハッシュのキーと値を1つずつ取り出し、すべての要素を処理することができます。配列の場合はインデックスの順に要素を取り出しましたが、ハッシュの場合は「キー」と「値」の組を取り出すことになります。

ハッシュ用のeachは次のように書きます。


```
ハッシュ.each do |キーの変数, 値の変数|
  繰り返したい処理
end
```

さっそく使ってみましょう。

実行例

```
> irb --simple-prompt
>> address = {name: "高橋", furigana: "タカハシ"}
=> {:name=>"高橋", :furigana=>"タカハシ"}
>> address.each do |key, value|
?>   puts "#{key}: #{value}"
>> end
name: 高橋
furigana: タカハシ
=> {:name=>"高橋", :furigana=>"タカハシ"}
```

eachメソッドによって、ハッシュaddressが持っている項目名とその値を表示するputsメソッドが繰り返し実行されるのがわかります。

2.3　正規表現

Rubyで文字列を処理するときには、**正規表現**（Regular Expression）というものがよく使われます。正規表現を使うと、

- 文字列とパターンの一致（マッチング）
- パターンを使った文字列の切り出し

などを手軽に行えます。

正規表現は、PerlやPythonなど、Rubyの先輩格にあたるスクリプト言語でつちかわれてきた機能です。Rubyもその流れを受け継いでいて、言語に組み込みの機能として、手軽に正規表現を扱えます。文字列処理はRubyの得意分野ですが、それはこの正規表現のおかげでもあります。

2.3.1　パターンとマッチング

「○○という文字列を含んだ行を表示したい」とか、「○○と××の間に書かれた文字列を抜き出したい」などといった、特定の文字列のパターンに対する処理を行いたい場合があります。文字列がパターンに当てはまるかどうかを調べることをマッチングといい、パターンに当てはまることを「マッチする」といいます。

このような文字列のパターンをプログラミング言語で表現するために使われるのが、正規表現です（図2.5）。

図 2.5　マッチングの例

「正規表現」という言葉から、何やら難しげな雰囲気が漂う、硬そうな印象を持たれるかもしれません。実際のところ正規表現の世界は何かと奥が深いのですが、単純なマッチングに使う分にはあまり身構える必要はありません。まずは、そういうものがあるということを覚えておいてください。

正規表現オブジェクトを作るための構文は、次の通りです。

/パターン/

たとえば「Ruby」という文字列にマッチする正規表現は、

```
/Ruby/
```

と書きます。そのままですね。アルファベットと数字からなる文字列に一致するパターンを書く分には、「そのまま」で大丈夫です。

正規表現と文字列のマッチングを行うためには、「=~」演算子を使います。同じオブジェクト同士が等しいかどうかを調べる「==」に似ています。

正規表現と文字列のマッチングを行うには、

/パターン/ =~ マッチングしたい文字列

と書きます。英数字や漢字だけのパターンを使った場合は、パターンの文字列を含んでいればマッチし、含んでいなければマッチしません。マッチング

が成功したときは、マッチ部分の位置を返します。文字の位置は、配列のインデックスと同様に、0から数えます。つまり、先頭文字の位置は0と表されます。一方、マッチングが失敗だとnil（ニル）を返します。

実行例

```
> irb --simple-prompt
>> /Ruby/ =~ "Yet Another Ruby Hacker,"
=> 12
>> /Ruby/ =~ "Ruby"
=> 0
>> /Ruby/ =~ "Diamond"
=> nil
```

正規表現の右側の「/」に続けて「i」と書いた場合には、英字の大文字・小文字を区別せずにマッチングを行うようになります。

実行例

```
> irb --simple-prompt
>> /Ruby/ =~ "ruby"
=> nil
>> /Ruby/ =~ "RUBY"
=> nil
>> /Ruby/i =~ "ruby"
=> 0
>> /Ruby/i =~ "RUBY"
=> 0
>> /Ruby/i =~ "rUbY"
=> 0
```

これ以外にも、正規表現には数々の書き方や使い方があります。詳しくは「第16章　正規表現（Regexp）クラス」で説明します。

Column

nilとは？

nilはオブジェクトが存在しないことを表す特別な値です。正規表現によるマッチングの際に、どこにもマッチしなかったことを表す場合のように、メソッドが意味のある値を返すことができないときにはnilが返されます。また、配列やハッシュからデータを取り出す場合に、まだ存在していないインデックスやキーを指定すると次のようにnilが得られます。

実行例

```
> irb --simple-prompt
>> item = {"name"=>"ブレンド", "price"=>610}
=> {"name"=>"ブレンド", "price"=>610}
>> item["tax"]
=> nil
```

if文やwhile文は、条件を判定するときにfalseとnilを「偽」の値として扱い、それ以外のすべての値を「真」として扱います。したがって、trueかfalseのどちらかを返すメソッドだけではなく、「何らかの値」もしくは「nil」を返すメソッドも、条件として使うことができます。

次の例は配列の中の「林」という文字を含む文字列だけを出力します。

List print_hayasi.rb

```
names = ["小林", "林", "高野", "森岡"]
names.each do |name|
  if /林/ =~ name
    puts name
  end
end
```

実行例

```
> ruby print_hayasi.rb
小林
林
```


第3章 コマンドを作ろう

この章では、コマンドラインからデータを受け取り、処理を行う方法を紹介します。また、第1部のまとめとして、Unixのgrep（グレップ）コマンドもどきを作成しましょう。Rubyプログラミングのおおまかな流れをつかんでください。

3.1 コマンドラインからのデータの入力

今まで行ってきたことは、データを画面に出力することでした。「出力」があればその反対、「入力」も試してみたくなります。そもそも、普通に使えるコマンドを作るにはプログラムに動作を指示する方法を知らなければいけません。そこで、Rubyのプログラムにデータを入力してみましょう。

プログラムにデータを与えるには、コマンドラインを利用する方法が一番簡単です。コマンドラインの情報をデータとして受け取るには「ARGV（アーグヴィ）」という配列オブジェクトを使います。このARGVという配列は、コマンドラインからスクリプトの引数として与えられた文字列を要素として持っています。

List 3.1で確認してみましょう。コマンドラインでスクリプトに引数を指定するときは、1つずつ空白で区切って入力してください。

List 3.1 print_argv.rb

```
puts "最初の引数: #{ARGV[0]}"
puts "2番目の引数: #{ARGV[1]}"
puts "3番目の引数: #{ARGV[2]}"
puts "4番目の引数: #{ARGV[3]}"
puts "5番目の引数: #{ARGV[4]}"
```

実行例

```
> ruby print_argv.rb 1st 2nd 3rd 4th 5th
最初の引数：1st
2番目の引数：2nd
3番目の引数：3rd
4番目の引数：4th
5番目の引数：5th
```

配列ARGVを使えば、データをプログラムの中にすべて書いておく必要はなくなります。配列なので、要素を取り出して変数に代入することもできます(List 3.2)。

List 3.2 happy_birth.rb

```
name = ARGV[0]
print "Happy Birthday, ", name, "!\n"
```

実行例

```
> ruby happy_birth.rb Ruby
Happy Birthday, Ruby!
```

引数から取得したデータは文字列になっているので、これを計算に使うときは数値に変換する必要があります。文字列を整数にするには、to_iメソッドを使います(List 3.3)。

List 3.3 arg_arith.rb

```
num0 = ARGV[0].to_i
num1 = ARGV[1].to_i
puts "#{num0} + #{num1} = #{num0 + num1}"
puts "#{num0} - #{num1} = #{num0 - num1}"
puts "#{num0} * #{num1} = #{num0 * num1}"
puts "#{num0} / #{num1} = #{num0 / num1}"
```


実行例

```
> ruby arg_arith.rb 5 3
5 + 3 = 8
5 - 3 = 2
5 * 3 = 15
5 / 3 = 1
```


3.2 ファイルからの読み込み

Rubyのスクリプトが入力として受け取れるデータは、コマンドライン引数だけではありません。ファイルからデータを読み込むこともできます。

Rubyのソースコードには、「ChangeLog」というテキストファイルがついています。これには、誰がどのようにRubyを変更したかが記してあります。

このファイルの中身は、次のようになっています。

```
                                        ⋮
Mon Dec 29 19:38:01 2014  Yukihiro Matsumoto  <matz@ruby-lang.org>

        * version.h (RUBY_VERSION): 2.3.0 development has started.

                                        ⋮
```

メモ Rubyのソースコードは、Rubyの公式ウェブサイトから入手できます。ChangeLogファイルは、GitHubのRubyレポジトリからも取得可能です。

- Rubyのソースコードダウンロード
  https://www.ruby-lang.org/ja/downloads/
- GitHub上のChangeLogファイル
  https://raw.github.com/ruby/ruby/ruby_2_3/ChangeLog

このファイルを使って、Rubyでのファイル操作の練習をしてみましょう。

3.2.1　ファイルからテキストデータを読み込んで表示する

まず、単純にファイルの中身をすべて表示するプログラムを作ってみましょう。ファイルの中身を表示するプログラムは、次のような流れになります。

①ファイルを開く
②ファイルのテキストデータを読み込む
③読み込んだテキストデータを出力する
④ファイルを閉じる

この流れを、そのままプログラムにしてみましょう（List 3.4）。

List 3.4　read_text.rb

```
1: filename = ARGV[0]
2: file = File.open(filename) # ①
3: text = file.read           # ②
4: print text                 # ③
5: file.close                 # ④
```

今までの例に比べると、ちょっとプログラムらしくなってきました。1行ずつ説明します。

1行目では、filenameという変数にコマンドラインから受け取った最初の引数の値ARGV[0]を代入しています。つまり、変数filenameは読み出したいファイルの名前を示していることになります。

2行目で使っている「File.open(filename)」は、filenameという名前のファイルを開き、そのファイルを読み込むためのオブジェクトを返します。……といわれても、「ファイルを読み込むためのオブジェクト」というのが何を意味しているのかよくわからないという方もいるかもしれません。あまり気にせず、ここではそういうオブジェクトがあるとだけ思ってください。詳しくは「第17章　IOクラス」で説明します。

この「ファイルを読み込むためのオブジェクト」が実際に使われるのは3行目です。ここでは、「read」というメソッドでデータを読み込み、その結果をtextに代入しています。ここでtextに代入されたテキストデータが、4行目で出力されます。printメソッドは今までにも何度も使ってきたので、もうすっかりおなじみのことでしょう。そして、最後に「close」というメソッドを実行します。これは、開いたファイルを閉じるためのメソッドです。

このプログラムを次のように実行すると、指定したファイルの内容をそのまま一気に表示します。

```
> ruby read_text.rb 表示したいファイル名
```

もっとも、ファイルを読み込むだけであれば、File.readメソッドを使うともっと簡単に書けます（List 3.5）。

List 3.5　read_text_simple.rb

```
1: filename = ARGV[0]
2: text = File.read(filename)
3: print text
```

File.readメソッドについても詳しくは「第17章　IOクラス」で説明します。

さらに、変数が不要であれば、1行でも書けます（List 3.6）。

List 3.6　read_text_oneline.rb

```
1: print File.read(ARGV[0])
```

3.2.2　ファイルからテキストデータを1行ずつ読み込んで表示する

ここまでで、まとめて読み込んだテキストデータを表示することができるようになりました。しかし、先ほどの方法では、

- ファイルのデータをまとめて読み込むのに時間がかかる
- 一時的にすべてのデータをメモリにためることになるので、大きなファイルの場合に困ることがある

といった問題があります。

100万行あるようなファイルでも、本当に必要なのは最初の数行だけ、ということもあります。そのような場合、すべてのファイルを読み込むまで何もしない、というのは、時間とメモリを無駄に使ってしまうことになります。

このような問題を解決するには、データをすべて読み込んでから処理を開始するというアプローチをやめる必要があります（図3.1）。

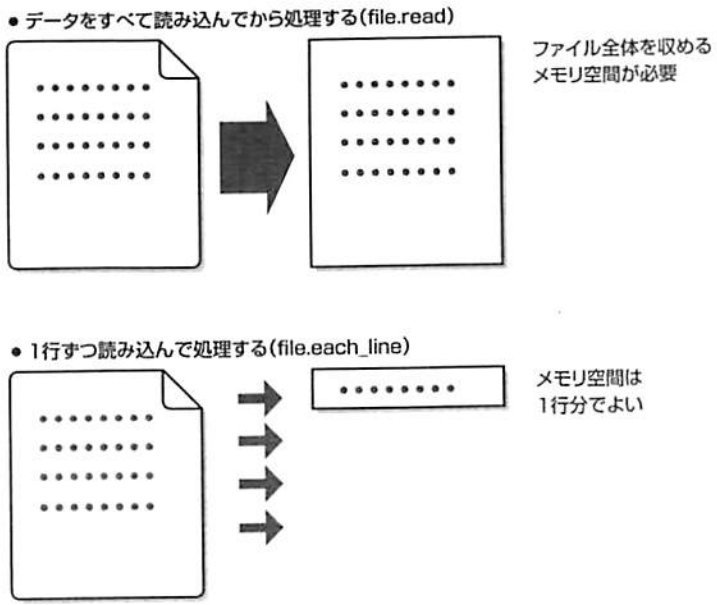

図 3.1　テキストの読み込み方の違い

List 3.4を、1行ずつ読み込むたびに出力するように変更してみましょう（List 3.7）。そうすれば、使われるメモリはその行の分だけで済むようになります。

List 3.7　read_line.rb

```
1: filename = ARGV[0]
2: file = File.open(filename)
3: file.each_line do |line|
4:   print line
5: end
6: file.close
```

1、2行目は、List 3.4と同じです。3行目以降がちょっと変わっています。3行目から5行目はeach_lineメソッドを使っています。

each_lineメソッドは、第2章で紹介したeachメソッドに似たメソッドです。eachメソッドは配列の各要素をそれぞれ処理するメソッドでしたが、each_lineメソッドはファイルの各行をそれぞれ処理するメソッドです。ここではファイルを1行ずつ読み込み、その行の文字列lineをprintメソッドで出力することで、最終的にすべての行が出力されています。

3.2.3　ファイルの中から特定のパターンの行のみを選んで出力する

Unixには、grepというコマンドがあります。これは、入力されたテキストデータの中から、正規表現で指定した特定のパターンにマッチする行を出力するコマンドです。これに似たコマンドを作ってみましょう（List 3.8）。

List 3.8　simple_grep.rb

```
1: pattern = Regexp.new(ARGV[0])
2: filename = ARGV[1]
3:
4: file = File.open(filename)
5: file.each_line do |line|
6:   if pattern =~ line
7:     print line
8:   end
9: end
10: file.close
```

List 3.8を実行するには、次のように入力します。

```
> ruby simple_grep.rb パターン ファイル名
```

少し長くなったので、1行ずつ見ていきましょう。

Rubyを実行する際にコマンドラインで与えた引数は、ARGV[0]とARGV[1]に代入されます。1行目では、1つ目の引数ARGV[0]を元に正規表現オブジェクトを作り、変数patternに代入します。「Regexp.new(*str*)」という形で、引数の文字列*str*から正規表現オブジェクトを作ります。そして2行目では、2つ目の引数ARGV[1]をファイル名に使う変数filenameに代入します。

4行目では、ファイルを開き、ファイルオブジェクトを作り、これを変数fileに代入します。

5行目はList 3.7と同じです。1行ずつ読み込んで変数lineに代入し、8行目までを繰り返します。

6行目はif文になっています。ここで、変数lineの値である文字列が変数patternの値である正規表現にマッチするかどうか調べます。マッチした場合、7行目のprintメソッドでその文字列を出力します。このif文にはelse節がないので、マッチしなかった場合は何も起こりません。

すべてのテキストの読み込みが終わったらファイルを閉じて終了します。

たとえば、ファイルChangelogから「matz」という文字列が含まれている行を出力したい場合には、次のように実行します。

```
> ruby simple_grep.rb matz Changelog
```

「matz」とは、まつもとゆきひろ氏のニックネームです。まつもとゆきひろ氏による変更の履歴が出力されます。

3.3 メソッドの作成

今までいくつかのメソッドを使ってきましたが、自分で作ることもできます。メソッドを作成する構文は次のようになります。

```
def メソッド名
  メソッドで実行したい処理
end
```

「Hello, Ruby.」と表示するメソッドを作ってみましょう。

```
def hello
  puts "Hello, Ruby."
end
```

この3行だけを書いたプログラムを実行しても、何も起こりません。helloメソッドが呼び出される前に、プログラムが終わってしまっているからです。そのため、自分で作成したメソッドを実行するコードも必要になります。

List 3.9 hello_ruby2.rb

```
1: def hello
2:   puts "Hello, Ruby."
3: end
4:
5: hello()
```

実行例

```
> ruby hello_ruby2.rb
Hello, Ruby.
```

「hello()」というメソッド呼び出しにより、1～3行目で定義されたhelloメソッドが実行されます。

3.4 別のファイルを取り込む

プログラムの一部を、別の新しいプログラムの中で使い回したいことがあります。たとえば、あるプログラムで使った自作メソッドを、別のプログラムで利用したい、といった場合です。

たいていのプログラミング言語では、別々のファイルに分割されたプログラムを組み合わせて、1つのプログラムとして利用するための機能を持っています。他のプログラムから読み込んで利用するためのプログラムを、**ライブラリ**といいます。

プログラムの中でライブラリを読み込むには、requireメソッドまたはrequire_relativeメソッドを使います。

```
require 使いたいライブラリのファイル名
```

または、

```
require_relative 使いたいライブラリのファイル名
```

使いたいライブラリのファイル名の「.rb」は省略することができます。

requireメソッドを呼ぶと、Rubyは引数に指定されたライブラリを探して、

そのファイルに書かれた内容を読み込みます（図3.2）。ライブラリの読み込みが終わると再び、requireメソッドの次の行から処理を再開します。

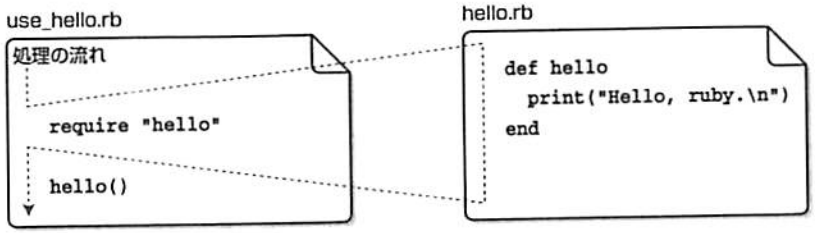

図 3.2 ライブラリとそれを読み込むプログラム

requireメソッドは既存のライブラリを読み込むときに使います。ライブラリ名を指定するだけで、Rubyと一緒にインストールされたライブラリなど、あらかじめ決められた場所から探し出して読み込んでくれます。それに対してrequire_relativeメソッドは、実行するプログラムが置かれたディレクトリ（フォルダ）を基準にしてライブラリを探します。複数のファイルに分けて記述したプログラムを読み込むときに便利です。

実際の例として、先ほどのsimple_grep.rbの検索部分をライブラリとして、他のプログラムから使ってみましょう。ライブラリといっても別に変わった書き方は必要ありません。simple_grepメソッドを定義したファイル（List 3.10）と、それを利用するプログラム（List 3.11）を同じディレクトリに作成します。

List 3.10 grep.rb

```
def simple_grep(pattern, filename)
  file = File.open(filename)
  file.each_line do |line|
    if pattern =~ line
      print line
    end
  end
  file.close
end
```


List 3.11 use_grep.rb

```
require_relative "grep"          # grep.rbの読み込み（「.rb」は不要）

pattern = Regexp.new(ARGV[0])
filename = ARGV[1]
simple_grep(pattern, filename)  # simple_grepメソッドの起動
```

simple_grepメソッドは検索するパターンとファイル名が必要なので、これらをpatternとfilenameという引数で受け取るようになっています。

grep.rbで定義したsimple_grepメソッドを、use_grep.rbで呼び出していることに注目してください。List 3.8の実行例（p.50）と同様に、ファイルChangelogから「matz」という文字列が含まれている行を出力したい場合には、次のように実行します。

```
> ruby use_grep.rb matz Changelog
```

Rubyには、たくさんの便利なライブラリが標準で付属しています。これらを利用する場合にrequireメソッドを使います。

たとえば、dateライブラリを読み込むことで、今日の日付を求めるDate.todayメソッドや特定の日付のオブジェクトを生成するDate.newメソッドなどを利用できるようになります。Rubyの誕生日である1993年2月24日から今日までの日数を求めるプログラムは次のようになります。dateライブラリについては第20章で詳しく説明します。

```
require "date"

days = Date.today - Date.new(1993, 2, 24)
puts(days.to_i)  #=> 8323
```

Column

ppメソッド

pメソッドと同じような目的に使われるメソッドとして、ppメソッドがあります。ppは「Pretty Print」の略です。ppメソッドを利用するには、ppライブラリをrequireメソッドで読み込む必要があります。

List p_and_pp.rb

```
require "pp"

books = [
  { title: "猫街", author: "萩原朔太郎" },
  { title: "猫の事務所", author: "宮沢賢治" },
  { title: "猫語の教科書", author: "ポール・ギャリコ" },
]
p books
pp books
```

実行例

```
> ruby p_and_pp.rb
[{:title=>"猫街", :author=>"萩原朔太郎"}, {:title=>"猫の事務所",
:author=>"宮沢賢治"}, {:title=>"猫語の教科書", :author=>"ポール・
ギャリコ"}]
[{:title=>"猫街", :author=>"萩原朔太郎"},
 {:title=>"猫の事務所", :author=>"宮沢賢治"},
 {:title=>"猫語の教科書", :author=>"ポール・ギャリコ"}]
```

pメソッドとは異なり、ppメソッドはオブジェクトの構造を表示する際に、適当に改行を補って見やすく整形してくれます。ハッシュの配列のように、入れ子になったコンテナを確認する場合に利用するとよいでしょう。

「ようやく　ようやく
これでスタートラインだ
ここからはじまるんだ
ここから
神様との　神様との対話が」
―― 竹本健治「入神」

第2部

基礎を学ぼう

**プログラムの書き方には
いくつかの約束事があります。
Rubyにおけるプログラミングの
ルールを学んでください。**