Ruby 初級者向けレッスン第 20 回

okkez @ Ruby 関西, モリ@小波ゼミ

2008 年 05 月 17 日

今回の内容

- Ruby の基本的な部分をおさらいする
  - 基本型
  - 制御構造

今回のゴール

- リテラルについて知る
- 制御構造について知る
- それらを用いた簡単なプログラムを書く

基本型

Ruby の基本型には以下のものがあります。

- 数値
- 文字列
- 範囲
- 配列
- ハッシュ
- シンボル
- 正規表現

数値

Ruby は整数と浮動小数点数を組込みでサポートしています。

- Ruby の整数は Fixnum か Bignum クラスのインスタンス
- Fixnum のサイズは OS 依存
- Fixnum に収まらない整数は自動的に Bignum に変換されます
- Bignum はメモリが許す限りいくらでも大きな整数を保持できます
- ASCII に対応する数値を得るにはその文字に '?' を付けます
- 小数点や指数の付いている数値リテラルは Float オブジェクトになります

サンプルコード

```
01:
02: 123456                  #=> Fixnum
03: 0d123456                #=> Fixnum
04: 123_456_789             #=> Fixnum アンダースコアは無視
05: -543                    #=> Fixnum 負の数
06: 0xaabb                  #=> Fixnum 16 進数
07: 0666                    #=> Fixnum  8 進数
08: -0b10_1010              #=> Fixnum 負の 2 進数
09: 123_456_789_123_456_789 #=> Bignum
10:
11: ?a        #=> 97
12: ?\n       #=> 10  改行コード (0x1a)
13: ?\C-a   #=> 1   コントロール a <=> ?A & 0x9f = 0x01
14: ?\M-a   #=> 225 メタは 8 ビット目を立てる
15: ?\M-\C-a  #=> 129 メタ コントロール a
16: ?\C-?     #=> 127 削除文字
17:
18: 12.34     #=> 12.34
19: -0.1234e2 #=> -12.34
20: 1234e-2   #=> 12.34
```

文字列

Ruby には文字列リテラルを作成する方法がたくさんあります

- シングルクォート文字列 (' 文字列' or %q!文字列!)
  - \\(二つのバックスラッシュ) が \ (一つのバックスラッシュ) に置換されます
  - \' がシングルクォートに置換されます

- ダブルクォート文字列 (”文字列” or %Q!文字列! or %/文字列/)
  - サンプルコードにあるような置換と式展開が行われます
- ヒアドキュメントも使用できます

サンプルコード

```
01: puts 'hello'                    #=> hello
02: puts 'a backslash \'\\\''       #=> a backslash '\'
03: puts %q/simple string/          #=> simple string
04: puts %q(nesting (really) works) #=> nesting (really) works
05: puts %q no_blanks_here ;        #=> no_blanks_here
06:
07: a = 123
08: puts "\123mile"                     #=> Smile
09: puts "Say \"Hello\""                #=> Say "Hello"
10: puts %Q!"I said 'nuts'," I said!    #=> "I said 'nuts'," I said
11: puts %Q(Try #{a + 1}, not #{a - 1})  #=> Try 124, not 122
12: puts %<Try #{a + 1}, not #{a - 1}>   #=> Try 124, not 122
13: puts "Try #{a + 1}, not #{a - 1}"    #=> Try 124, not 122
14: puts %{ #{a = 1;b = 2; a + b} }      #=>  3
15:
16: print <<HERE
17: Double quoted \
18: here document.
19: It is #{Time.now}
20: HERE
21: print <<-'THERE'
22:       This is single quoted.
23:       The above used #{Time.now}
24:       THERE
25: #=> Double quoted here document.
26: #=> It is Sun May 11 22:58:50 +0900 2008
27: #=>       This is single quoted.
28: #=>       The above used #{Time.now}
29:
30: puts 'Con' "cat" 'en' "ate" #=> Concatenate
```

範囲

- 1 .. 10 は両端の値を含みます
- 1 ... 10 は右端の値を含みません

サンプルコード

```
01: p (1..10).to_a  #=> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
02: p (1...10).to_a #=> [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

配列

Ruby では一つの配列に様々な型のオブジェクトを格納することができます。

- 角括弧 [ ] で囲むことによって生成できます
- %w は式展開をしません
- %W は式展開を行います (ダブルクォート文字列と同じです)

サンプルコード

```
01: p %w( fred wilma barney betty great\ gazoo)
02: #=> ["fred", "wilma", "barney", "betty", "great gazoo"]
03: p %w( Hey!\tIt is now -#{Time.now}- )
04: #=> ["Hey!\\tIt", "is", "now", "-\#{Time.now}-"]
05: p %W( Hey!\tIt is now -#{Time.now}- )
06: #=> ["Hey!\tIt", "is", "now", "-Sun May 11 23:13:06 +0900 2008-"]
```

ハッシュ

- Hash のリテラルは、キーと値のペアをブレースで囲むことによって作成します
- キーと値はカンマまたは => で区切ります
- あるハッシュ内のキーや値の型を揃える必要はありません
- 1.9 では最後の書き方は出来ません

サンプルコード

```
01: colors = {
02:   "red"   => 0xf00,
03:   "green" => 0x0f0,
04:   "blue"  => 0x00f,
05: }
06: hash = {
07:   :aaa  => "aaa",
08:   "abc" => :abc,
09:   123   => [1, 2, 3],
10: }
11: h = { :a, :b, :c, :d, :e, :f,}
```

シンボル

シンボルは任意の文字列と一対一に対応するオブジェクトです。
主な用途は以下のとおりです。

- ハッシュのキー
- アクセサの引数で渡すインスタンス変数名
- メソッド引数で渡すメソッド名
- C の enum 的な使用 (値そのものは無視してよい場合)

サンプルコード

```
01: :Object
02: :my_variable
03: :"Ruby rules"
04: %s!Symbol!
05: a = 'cat'
06: p :'catsup'  #=> :catsup
07: p :"#{a}sup" #=> :catsup
08: p :'#{a}sup' #=> :"\#{a}sup"
```

正規表現

Ruby では、パターンのマッチングと置換を簡単に行えるように、正規表現を組み込みでサポートしています。
正規表現リテラルの表記方法は以下の通りです。

- /パターン/
- /パターン/オプション
- %r!パターン!
- %r!パターン!オプション

他に知っていると便利なのは以下の事柄です。

- パターン内に #{...} で式展開が実行される
  - デフォルトではリテラルが評価される度に式が評価される
  - o オプションを指定すると最初にリテラルが評価された時だけ式が評価されるようになる
- 1.9 から正規表現エンジンが鬼車に変更されました

詳しいオプションやパターンなどについてはリファレンスマニュアルなどを参照してください。

サンプルコード

```
01: p /abc/    #=> /abc/
02: p /abc/i   #=> /abc/i
03: p %r!abc!  #=> /abc/
04: p %r!abc!i #=> /abc/i
```

制御構造

Ruby の制御構造には以下のものがあります。

- 逐次処理
- 条件判断
- 繰り返し

逐次処理

プログラムを書かれた通りに、先頭から順に実行する

```
n = 1
puts  "#{n}番目に実行する"
n += 1
puts  "#{n}番目に実行する"
n += 1
puts  "#{n}番目に実行する"
```

条件判断

ある条件が成り立つ場合は○○を、そうでない場合は××を実行する

```
if 条件 then
  条件が成り立ったときに実行したい処理
else
  条件が成り立たなかったときに実行したい処理
end

param = 10

if param > 10
  puts 'param > 10'
else
  puts 'param <= 10'
end
```

このように then を省略することができます。また以下のように、 else 節を省略することができます。

```
# 引数がいくつかあるコマンドを作成する場合
if ARGV.size == 0
  raise ArgumentError, '引数の数が足りません'
end

puts "引数は #{ARGV.size} 個でした"
```

その他の条件判断

- unless
  - if の逆
  - elsif に対応するものはない
- case
  - 条件分岐がたくさんある場合に用いる
- 三項演算子

引数が一つだけのコマンドを作成する場合

```
unless ARGV.size == 1
  raise ArgumentError
end

puts "引数は #{ARGV.size} 個でした"
```

単純に条件が多い場合

```
case
when '1' == ARGV.first then puts '1'
when /\A--help\z/ =~ ARGV.first then puts 'Help me!'
when 100 < ARGV.first.to_i then puts '100 より大きい数字です'
else puts '条件にあてはまりません'
end
```

よく使われるパターン

```
case ARGV.first
when /\A-+h/ then puts 'Help me'
when /\A-+v/ then puts 'version 1.0'
else puts '条件にあてはまりません'
end

case obj
when String then 'String class の場合の処理'
when Regexp then 'Regexp class の場合の処理'
when Fixnum then 'Fixnum class の場合の処理'
when Bignum then 'Bignum class の場合の処理'
else 'どのクラスでも無い場合の処理'
end
```

三項演算子の例

```
ARGV.size == 0 ? puts '引数は 0 個' : puts '引数は 0 個ではない'
```

繰り返し

ある条件が成り立つ間、○○を繰り返し実行します。

```
while 条件 do
  条件が成り立っている間ずっと実行したい処理
end

while true
  puts 'ずっと実行される'
end
```

このように do を省略することができます。

その他の繰り返し

loop

- 無限ループ
- do を省略できない

until

- while の逆
- unless と間違えると大変なことになります

for

- 糖衣構文 (シンタックスシュガー) のようなものです
- ブロック付きのイテレータとは変数のスコープが異なります

イテレータ

- Integer#times
- Integer#step
- Array#each

などなど

便利な修飾子

if/unless, while/until には短くてきれいなコードを書くのに便利な方法があります。まずは例を示します。

```
[1,2,3,4,5].each do |elem|
  puts elem if elem % 2 == 0 #=> 偶数だけ表示
end
```

このように、式の後ろに修飾子として if 式を記述すると、条件が成り立つ場合のみその直前に書かれた式を実行することができます。

ただし、while/until を修飾子として使用するときは注意が必要です。修飾している文が begin/end ブロックの場合は、条件式の値に関係なく、そのブロック内のコードが常に最低 1 回は実行されます。

```
puts 'Hello' while false
begin
  puts 'Goodbye'
end while false
```

条件の書き方いろいろ

最後に、条件の書き方を説明しておきます。

Ruby では、nil, false 以外は全て真として扱われます。例えば、以下のような感じです。

```
if ARGV.first
  puts 'ARGV.first is not nil'
else
  puts 'ARGV.first is nil'
end
```

この場合は、スクリプト実行時にコマンドライン引数を指定しているかどうかで結果が変化します。コマンドライン引数が指定されていない場合は、 ARGV = [] となるので ARGV.first(ARGV[0]) は nil になります。

```
while line = gets
  # line を処理する
end
```

こちらの例は、ファイルの次の行を返す IO#gets は、ファイルの終端に達すると nil を返すということを利用しています。

また、 Ruby では 0 や空文字列 (””) は偽になりません。

```
if 0
  # 常にに実行される
end

if ''
  # 常に実行される
end
```

これらを、条件に使用したい場合は例えば以下のようにします。

```
param = 1
if param == 0
  # 実行されない
end

value = 'hoge'
if value.empty?
  # 実行されない
end
```

まだまだ色々ありますが、今回はここまでです。

演習

ヒントに書いてあるメソッドについてリファレンスマニュアルで調べたりするとよいと思います。

3 の倍数と 3 の付く数字で… (並)

FizzBuzz 問題の亜種です。

- 1 から 100 までの数値を出力します
- 3 の倍数と 3 の付く数字で何か変な文字列を出力します

- 5 の倍数で動物の鳴き声を文字列化したものを出力します

以下は、一つ目の条件で 'Aho' と出力し、二つ目の条件で 'Bow' と出力した場合の例です。

```
1
2
Aho
4
Bow
Aho
7
8
Aho
Bow
11
Aho
Aho
14
AhoBow
16
17
Aho
19
Bow
...
```

ヒント

- Fixnum#to_s
- Fixnum#%
- String#include?
- Regexp#=~

99 Bottles of Beer on the Wall (並)

"99 本のビールが壁に..." 遠足などでよく歌われる古典的な童謡の歌詞、"99 Bottles of Beer on the Wall" を出力するプログラムを書いてみましょう。答えは一つではないので、何通りも書いてみましょう。

```
# 出力例
99 Bottles of beer on the wall
99 Bottles of beer
```

```
Take one down and pass it around
98 Bottles of beer on the wall

98 Bottles of beer on the wall
98 Bottles of beer
Take one down and pass it around
97 Bottles of beer on the wall

...

3 Bottles of beer on the wall
3 Bottles of beer
Take one down and pass it around
2 Bottles of beer on the wall

2 Bottles of beer on the wall
2 Bottles of beer
Take one down and pass it around
1 Bottle of beer on the wall

1 Bottle of beer on the wall
1 Bottle of beer
Take one down and pass it around
No Bottles of beer on the wall

No bottles of beer on the wall
No bottles of beer
There are no more to pass around
No bottles of beer on the wall
```

石取りゲーム (難)

テイルズシリーズでお馴染みの石取りゲームを Ruby で実装してみましょう。互いにミスをしなければ、最初の石の状態で先手が勝つか後手が勝つか決まります。

ルール

- コンピュータと人間が対戦する
- 人間は先手か後手か選択できる
- 交互に石を取り合う
- 最後の一つを取った方が負け

- 一度に取れる石の個数は、 1 から 3 個

仕様

- 初期の石の個数は、10 から 100 個の間でランダムにする
- コンピュータは負けそうな予感がするときはたまにゴネて順番を選択させてくれません
- 毎ターン、残りの石の個数を表示する
- 勝負がつくと勝ち負けを表示して終了する

ヒント

- Kernel#gets, IO#gets
- String#chomp, String#chomp!
- Kernel#rand
- ゴネないバージョンを作成してからゴネるバージョンを作成してみるといいかもしれません

参考文献

初めてのプログラミング
http://www.oreilly.co.jp/books/4873112923/

プログラミング Ruby 第 2 版 言語編
http://ssl.ohmsha.co.jp/cgi-bin/menu.cgi?ISBN=4-274-06642-8

たのしい Ruby 第 2 版
http://shop.sbcr.jp/bm_detail.asp?sku=4797336617

今後の情報源

公式 Web サイト
http://www.ruby-lang.org/

Ruby リファレンスマニュアル刷新計画
http://doc.loveruby.net/refm/api/

日本 Ruby の会
http://jp.rubyist.net/

Rubyist Magazine
http://jp.rubyist.net/magazine/

okkez weblog
http://typo.okkez.net/