

Ruby 入門

櫻井彰人



Ruby とは?

・ Ruby:

- 松本ゆきひろ氏による(1993)
- 純粋オブジェクト指向
- スクリプト言語
- Web プログラムで、どんどんポピュラーに
 - ・ “Ruby on Rails”
(<http://www.rubyonrails.org/>)
- なぜか、きわめて Lisp like



松本行弘(Matzen)

Introduction

実行環境

• Windows/Unix/Linux/ etc.

- Interactive Ruby

- `irb`

- ファイルを実行するには **myProgram.rb**

```
ruby myProgram.rb
```



Installation

“Hello world”

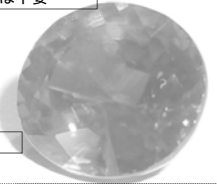
- “Hello world” プログラムは、画面上(標準出力)に “Hello, world!”、又は類似の文言を表示するだけの単純なプログラムである。[Wikipedia]

• Ruby の “Hello world” プログラム

```
puts "Hello World"
```

“,” は不要

画面上への印字



Getting Started

算数の初歩

• 1 + 2

```
1+2 # 3
```

のあとはコメント

• 正式な書き方(何と完璧オブジェクト指向):

```
1.+ (2) # 3
```

- 2 は引数

- + はオブジェクト 1 のメソッド



Getting Started

すべてがオブジェクト

• リテラルもオブジェクト

```
-3180.abs # 3180  
"3180 tutorial".length # 13  
"CSC3180".index("5") # 8  
true.& false # false
```

- Ruby オブジェクトの型を知るには **class** メソッドを使えばよい

```
-3180.class # Fixnum  
3.14159265.class # Float  
"3180 tutorial".class # String  
true.class # TrueClass
```



Getting Started

変数

- Ruby では変数宣言の必要なし:

```
a = 123; b = 456; c = 789; d = "Hello!"
```

- 並行代入が可能:

```
a, b, c, d = 123, 456, 789, "Hello!"
```

- 変数値の入れ替え:

```
a, b = "abcd", 1234
a, b = b, a
a                                     # 1234
b                                     # "abcd"
```

Getting Started

変数

- 変数はオブジェクトへの参照を保持する

```
str1 = "DENDAI"
str2 = str1
str1[5] = "8"
str1                                     # DENDA8
str2                                     # DENDA8
```

- ただし、例外がある: `Fixnum`, `Float`

- 変数はオブジェクト自体を保持する。それらへの参照を保持しているわけではない
- 同じ値の二つの `Fixnum` は値を別々に保持する

Getting Started

配列

- 3要素からなる配列を作る

```
myArray = [ 1, 'hello', 3.14 ]
```

- 最初の要素を取り出す

```
myArray[0]                               # 1
```

- 第二要素を変更する

```
myArray[1]='world'
myArray                                   # [ 1, 'world', 3.14 ]
```

- 要素を追加する

```
myArray << "Oh"
myArray                                   # [ 1, 'world', 3.14, "Oh" ]
```

Getting Started

配列

- Ruby の配列は負のインデックス値も可

```
myArray = [ 1, 'hello', 3.14 ]
```

myArray:	1	'hello'	3.14
index:	0	1	2
	-3	-2	-1

```
myArray[-1]                               # 3.14
myArray[-3]                               # 1
myArray[1000]                             # nil
myArray[-999]                             # nil
myArray[0..1]                             # [1, 'hello']
```

Getting Started

変数と配列

- 複数個の変数に、配列を代入することができる

```
a = [1,"e",3.14]
b, c = a                                   # b=1, c="e"
b, *c = a                                  # b=1, c=["e",3.14]
b, c, d, e = a                             # b=1, c="e"
                                           # d=3.14 e=nil
```

Getting Started

ハッシュ

- キーと値の組を記憶するハッシュ表

```
preference = {
  'c' => 'classical',
  'c++' => 'gothic',
  'java' => 'baroque',
  'prolog' => 'techno'
}

preference['prolog']                       # 'techno'
preference['prolog']='terrific'
preference['prolog']                       # 'terrific'
```

Getting Started

メソッド

- Ruby メソッドの簡単な一例

```
def sayHello(name)
  return "Hello, #{name}"
end

sayHello("DENDAI") # "Hello, DENDAI"
```

文字列のなかに式を埋め込むことができる

- メソッドはクラスの中でもまた外でも定義できる
 - クラスの外側でメソッドを定義すると、それは、トップレベルの Object インスタンスのメソッドとなる

Getting Started

メソッド

- もし return 式がなければ、メソッドは、最後に実行された式の値を返す

- Ruby では、必要でないときはいつでも return 式を省略することができる

- 例: 再帰呼出しを用いた階乗

```
def factorial(n)
  if n==0 then 1 else n*factorial(n-1) end
end
```

```
def factorial(n)
  n==0 ? 1 : n*factorial(n-1)
end
```

Getting Started

メソッド

- Ruby ではメソッドのオーバーローディングはない
- しかし、引数に default 値を与えることができる

```
def sum(a, b, c=2, d=5)
  a + b + c + d
end
```

```
sum(1,9) # 17
sum(1,2,3,4) # 10
sum(1,2,-3) # 5
```

Getting Started

メソッド

- 質問(true/falseが答え)に相当するメソッドには、慣例として、疑問符 '?' を最後につける

```
a = []
a.empty? # true
```

- “危険な” メソッド、すなわち、破壊的なメソッドには慣例として '!' を最後につける

```
a = [1,2,3,4]
a.reverse # [4,3,2,1]
a # [1,2,3,4]
a.reverse! # [4,3,2,1]
a # [4,3,2,1]
```

Arrayクラスには reverse と reverse! とがある

Getting Started

正規表現

- Ruby は文字列のマッチング用に =~ 演算子を用意している

```
if str =~ /Java|Ruby/
  puts "string containing Java or Ruby"
end
```

マッチング用パターン

- str 中の最初の 'Python' を 'Ruby' で置換える

```
str.sub!(/Python/, 'Ruby')
```

- str 中の全部の 'Python' を 'Ruby' で置換

```
str.gsub!(/Python/, 'Ruby')
```

勿論、危険でない sub と gsub もある

Getting Started

true は?

- nil や false 以外は true
 - Ruby では 0, "", [] などはずべて true

- 前のスライドの例

```
if str =~ /Java|Ruby/
  puts "string contains Java or Ruby"
end
```

- この =~ 演算子はマッチングが起こった文字位置 (マッチした場合) または nil (マッチしなかった場合) を値とする

Getting Started

if 式

- Ruby の if 式は他の言語と類似

```
if str =~ /Java/
  puts "Java"
elsif str =~ /Ruby/
  puts "Ruby"
else
  puts "What's that?"
end
```

Getting Started

unless 式

- Ruby には、if 式の否定がある

```
unless gpa > 0
  puts "Absolute Zero"
else
  puts "Positive"
end
```

- "C-style" の ?: 式も用いることができる

```
a = 1
b = 2
max = a > b ? a : b
max
```

2

Getting Started

更に if と unless

- if/unless 式は普通の式の最後に付加することができる

```
a, b = 1, 2

a = b if a > b
a # 1
b # 2

a = b unless a > b
a # 2
b # 2
```

Getting Started

case 式

- C/Java の switch と同じ (ただし break はなし!)

```
case university
when "電大": puts "東京電機大学"
when /東京/: puts "他の東京の大学"
when /京都/, /慶應/, /工科/, /早稲田/
  puts "他の大学"
else
  puts "?"
end
```

改行(new line)があるなら ":"
は不要

Getting Started

while と until ループ

- 単純な while ループ構造

```
a = 1
while a < 100 do # do は省略可
  a *= 2
end
```

- 実は、こんな書き方もできる

```
a = 1
a *= 2 while a < 100
```

- until ループはその反対

```
a = 1
a *= 2 until a >= 100
```

Getting Started

イテレータ

- "Hello World" を3回印字する

```
for i in 1..3 do # do は省略可
  puts "Hello World"
end
```

- こういう風にも書ける

```
3.times do # do end は { } も可
  puts "Hello World"
end
```

- これは次に等価:

```
3.times {puts "Hello World"}
```

Getting Started

イテレータ

- 0 から 9 までの整数を印字

```
for i in 0..9 do # do は省略可
  puts i
end
```

- または...

```
0.upto(9) do |i|
  puts i
end
```

- 等価:

```
0.upto(9){|i| puts i}
```

Getting Started

イテレータ

- [0, 10] 中の偶数をすべて印字する

```
0.step(10,2){|i| puts i}
```

- 配列のイテレータ:

```
for i in ['A', 'B', 'C', 'D', 'E'] do
  puts i
end
```

- これに等価:

```
['A', 'B', 'C', 'D', 'E'].each do |i|
  puts i
end
```

Getting Started

Block 構造

- {..} または do..end でくくったコード

```
{ puts 'This is a block' }
```

```
do
```

```
  puts 'this is another block'
```

```
end
```

- パラメータをブロック内に引渡す ||

```
0.upto(9){|i| puts i}
```

i はブロック内に渡される

これがブロック

Getting Started

Block 構造

- 例: 0-9 内の整数を印字する

```
0.upto(9){|i| puts i}
```

- この 0 に関する upto メソッドは、例えば 1 を生成しブロック中の i に引渡す
- ブロック内のコードが実行される
- 終了すると、制御は upto メソッドに移り、次の値 (2) が生成され、以上が繰り返される

Getting Started

Block 付きのメソッド

- ブロックを 3 回呼出すメソッド

```
def tripleCall
  yield
  yield
  yield
end
```

```
tripleCall{ puts "Hello World" }
```

- この yield が実行されるとブロックが呼ばれる

Getting Started

Block 付きのメソッド

- 0 から num までの整数を生成するメソッド

- さきほどの upto メソッドを手作り

```
def fromZeroTo(num)
  i = 0
  while i <= num
    yield i
    i += 1
  end
end
```

- 0 から 9 までの整数の印字

```
fromZeroTo(9){|i| puts i}
```

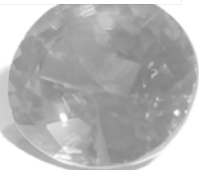
Getting Started

関数型言語なみ

- 関数の関数

```
cube = lambda {|x| x**3} # 関数定義
def double(x, f) # 関数を引数としてもらう関数
  f.call(f.call(x))
end
puts double(2, cube) #=> 512
```

数学で、 $f(x)$ と $f(a)$ とは何が違いますか？
注: 関数 $f(x)$ と関数 $f(x)$ の $x=a$ での値 $f(a)$



Getting Started

関数型言語なみ

- 関数の合成

```
def compose(x, f, g)
  g.call(f.call(x))
end
cube = lambda {|x| x**3}
down = lambda {|x| x-1}
puts compose(4, cube, down) #=>63
```



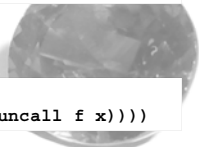
Getting Started

関数型言語なみ

- 関数を値として返す

```
def compose(f, g)
  lambda {|x| g.call(f.call(x))}
end
cube = lambda {|x| x**3}
down = lambda {|x| x-1}
puts compose(cube, down).call(4) #=>63
```

```
(defun compose (f g)
  (lambda (x) (funcall g (funcall f x))))
```



Getting Started

関数型言語なみ

- さらにこんなことも

```
def factorial(n) (1..n).inject{|x,y| x*y} end
def factorial(n) eval( ["*(1..n)"].join("**") ) end
```



Getting Started