

3. Rubyの基礎

いよいよプログラミング言語『Ruby』を使ったプログラミングをはじめます。ここではRubyの基本的な構文について説明します。

プログラムの説明を聞いたたら、「最初のプログラム」で作った『hello.rb』というファイルをエディタで開いて、

- プログラムを入力
- ファイルを保存
- コマンドプロンプトでプログラムを実行

をしながら、プログラムの動きを確かめてみよう。

文字列と数値

表示

putsは文字を表示する命令です。

```
1 | puts "Hello World"
```

putsと似たような命令にprintがあります。

```
1 | print "Hello World"
```

文字列

「」ダブルコーテーションで囲んだ文字を文字列と言います。

```
1 | puts "Hello World"
```

文字をダブルコーテーションで囲まないとエラーになります。

```
1 | puts Hello World
```

数値

ダブルコーテーションで囲んでいない0～9までの文字を数値と言います。

```
1 | puts 1
```

0～9までの文字はダブルコーテーションで囲まなくてもエラーになりません。

```
1 | puts 1
```

四則演算

足し算

数値は足し算ができます。

```
1 | puts 1 + 1
```

引き算

数値は引き算ができます。

```
1 | puts 2 - 1
```

掛け算

数値は掛け算ができます。

```
1 | puts 2 * 3
```

割り算

数値は割り算ができます。

```
1 | puts 4 / 2
```

四則演算の混合

足し算と引き算、掛け算と割り算は左から順番に実行されます。

```
1 | puts 2 + 3 - 1  
2 | puts 3 * 4 / 6
```

掛け算と割り算は、足し算と引き算より先に実行されます。

```
1 | puts 2 + 3 * 4  
2 | puts 5 - 2 / 3
```

()内の演算は、()の外の演算より先に実行されます。

```
1 puts (1 + 2) * 4  
2 puts 5 + (7 - 3)
```

整数と小数の演算

整数のみの四則演算の答えは整数になります。

```
1 puts 1 + 4  
2 puts 8 - 3  
3 puts 6 * 7  
4 puts 5 / 2
```

小数が含まれている四則演算の答えは小数になります。

```
1 puts 1 + 4.0  
2 puts 8.0 - 3  
3 puts 6.0 * 0.7  
4 puts 5.0 / 2
```

便利な演算子

「%」は割り算の余りを返します。

```
1 puts 5 % 2
```

「**」はべき乗の演算ができます。

```
1 puts 3 ** 4
```

変数

変数とは

変数とは、文字列や数値等をしまっておく箱のようなものです。

変数には好きな名前を付ける事ができます。

変数は以下のように使います。これは変数「a」に数値「10」を代入するという意味です。

```
1 | a = 10
```

代入

代入とは、変数に文字列や数値等をしまうことです。

以下のように使います。

変数「b」に数値「7」を代入します。

```
1 | b = 7
```

変数「c」に文字列「hello」を代入します。

```
1 | c = "hello"
```

変数「d」に式「1+2」を代入します。変数「d」に「1+2」の結果「3」が代入されます。

```
1 | d = 1 + 2
```

変数「e」に数値「3」を代入します。変数「f」に「e+2」を代入します。

```
1 | e = 3
```

```
2 | f = e + 2
```

自己代入

以下のように変数「a」へ、変数「a」に値を加算した結果を代入する事を自己代入と言います。

```
1 | a = 1
```

```
2 | a = a + 2
```

「 $a = a + 2$ 」は以下のように短縮して書くこともできます。

足し算($a = a + 1$)

```
1 | a += 1
```

他にも以下のような自己代入ができます。

引き算($a = a - 2$)

```
1 | a -= 2
```

掛け算($a = a * 3$)

```
1 | a *= 3
```

割り算($a = a / 4$)

```
1 | a /= 4
```

式展開

文字列の中に値(変数や数値や式)を埋め込む事を式展開と言います。

式展開は以下のように使います。値を「#{ }」で囲むと、文字列に埋め込まれます。

変数を埋め込む

```
1 | a = 2
2 | puts "変数aは #{a} です。"
```

式を埋め込む

```
1 | puts "1 + 5 = #{1 + 5}"
```

sleep

sleepメソッドを使うと、プログラムの処理を一定時間停止させることができます。以下のようにsleep の後に停止する秒数を指定します。

1秒刊停止する

```
1 | sleep 1
```

停止する秒数の指定を省くと、永久に停止する

```
1 | sleep
```

条件分岐

真偽

Rubyでは真偽を「true」「false」を使って表現します。

真

```
1 | true
```

偽

```
1 | false
```

rubyでは、「false」と「nil」が「偽」、それ以外が「真」になります。

比較演算子

「○○と等しい」「△△よりも大きい」等の比較に使用する演算子を比較演算子と言います。

「a」は「b」と等しい

```
1 | a == b
```

「c」は「d」と等しくない

```
1 | c != d
```

「e」は「f」より大きい

```
1 | e > f
```

「g」は「h」より小さい

```
1 | g < h
```

「i」は「j」以上(「i」は「j」より大きいか等しい)

|

```
1 | i >= j
```

「k」は「l」以下(「k」は「l」より小さいか等しい

```
1 | k <= l
```

比較結果は「true」「false」で表す

```
1 puts 1 == 2
2 puts 1 != 2
3 puts 1 > 2
4 puts 1 < 2
5 puts 1 >= 2
6 puts 1 <= 2
```

if文

if文を使うと、「もしも条件がtrueならば、処理Aを実行する。」というように、プログラムの流れを分岐させることができます。

```
1 if 条件
2   条件がtrueの時実行する処理A
3 end
```

「a」が2と等しければ「aは2と等しい」と表示する

```
1 a = 2
2 if a == 2
3   puts "aは2と等しい"
4 end
```

else

elseを使うと、「もしも条件がtrueならば、処理Aを実行する。そうでなければ、処理Bを実行する」というように、プログラムの流れを分岐させることができます。

```
1 if 条件
2   処理A
3 else
4   処理B
5 end
```

「a」が2と等しければ「aは2と等しい」と表示する。そうでなければ、「aは2と等しくない」と表示する。

```
1 a = 2
2 if a == 2
3   puts "aは2と等しい"
4 else
5   puts "aは2と等しくない"
6 end
```

elsif

elsifを使うと「もしも条件Aがtrueならば、処理Aを実行する。そうでない時、もしも条件Bがtrueならば、処理Bを実行する。そうでなかったら、処理Cを実行する」というように、プログラムの流れを分岐させることができます。

```
1 if 条件A
2   処理A
3 elsif 条件B
4   処理B
5 else
6   処理C
7 end
```

「a」が2と等しければ「aは2と等しい」と表示する。そうでない時、「a」が3と等しければ「aは3と等しい」と表示する。そうでなければ「aは2とも3とも等しくない」と表示する。

```
1 a = 3
2 if a == 2
3   puts "aは2と等しい"
4 elsif a == 3
5   puts "aは3と等しい"
6 else
7   puts "aは2とも3とも等しくない"
8 end
```

unless文

unless文を使うと、「もしも条件がtrueでなければ、処理Aを実行する。」というように、プログラムの流れを分岐させることができます。

```
1 unless 条件
2   条件がfalseの時実行する処理A
3 end
```

「a」が2と等しくなければ「aは2と等しくない」と表示する。

```
1 a = 3
```

```
2 unless a == 2
3   puts "aは2と等しくない"
4 end
```

else

unless文でも、if文と同様にelseを使用できます。elseを使うと、「もしも条件がtrueでなければ、処理Aを実行する。そうでなければ、処理Bを実行する」というように、プログラムの流れを分岐させることができます。

```
1 unless 条件
2   処理A
3 else
4   処理B
5 end
```

「a」が2と等しくなければ「aは2と等しくない」と表示する。そうでなければ、「aは2と等しい」と表示する。

```
1 a = 3
2 unless a == 2
3   puts "aは2と等しくない"
4 else
5   puts "aは2と等しい"
6 end
```

elsif

unless文ではelsifを使う事は出来ません。

if文、unless文のネスト(入れ子)

if文(unless文)をif文(unless文)の中に入れ込むことができます。

```
1 if 条件式A
2   if 条件式B
3     処理B
4   end
5 end
```

論理演算子

if文などで「もしも条件Aがtrue、かつ条件Bがtrueならば、処理Aを実行する」という複雑な条件式を記述する場合は論理演算子を使います。

and

andを使うと「条件Aがtrue、かつ条件Bがtrue」という条件式を書くことができます。条件Aと条件Bが両方ともtrueの場合は真、どちらかがfalseの場合は偽になります。

```
1 | 条件A and 条件B
```

「a > 1」がtrue、かつ「b > 2」がtrue

```
1 | a > 1 and b > 2
```

「a」が1と等しい、かつ「b」が2と等しければ、「a == 1 and b == 2はtrue」と表示する。

```
1 | a = 1
2 | b = 2
3 | if a == 1 and b == 2
4 |   puts "a == 1 and b == 2はtrue"
5 | end
```

or

orを使うと「条件Aがtrue、または条件Bがtrue」という条件式を書くことができます。条件Aと条件Bのいずれかがtrueの場合は真、両方falseの場合は偽になります。

```
1 | 条件A or 条件B
```

「a > 1」がtrue、または「b > 2」がtrue

```
1 | a > 1 or b > 2
```

「a」が1と等しい、または「b」が2と等しければ、「a == 1 or b == 2はtrue」と表示する

```
1 | a = 1
2 | b = 3
3 | if a == 1 or b == 2
4 |   puts "a == 1 or b == 2はtrue"
5 | end
```

&& と ||

「&&」は「and」と同じ意味です。andより優先順位が高くなっています。

```
1 | a > 1 && b > 2
```

「||」は「or」と同じ意味です。orより優先順位が高くなっています。

```
1 | a > 1 || b > 2
```

case文

case文を使うと、「変数Aの値が、値Aならば処理A、値Bならば処理B、値Cならば処理Cを実行する。」というように、プログラムの流れを分岐させることができます。

```
1 case 変数A
2 when 値A
3   処理A
4 when 値B
5   処理B
6 when 値C
7   処理C
8 end
```

case文でもif文と同じように、elseを使う事ができます。

「a」が、1ならば「あいうえお」、2ならば「かきくけこ」、3ならば「さしすせそ」、そうでなければ「ABC」と表示する。

```
1 a = 1
2 case a
3 when 1
4   puts "あいうえお"
5 when 2
6   puts "かきくけこ"
7 when 3
8   puts "さしすせそ"
9 else
10  puts "ABC"
11 end
```

繰り返し

while文

while文を使うと「条件がtrueの間、処理Aを繰り返し実行する」というように繰り返し処理を行う事ができます。

```
1 while 条件式 do  
2   処理A  
3 end
```

「a」が10より小さい間、「○回目の繰り返し」と繰り返し表示する。○回目の「○」には現在の「a」の値を表示させる。

```
1 a = 1  
2 while a < 10 do  
3   puts "#{a}回目の繰り返し"  
4   a += 1  
5 end
```

until文

until文を使うと「条件がfalseの間、処理Aを繰り返し実行する」というように繰り返し処理を行う事ができます。

```
1 until 条件 do  
2   処理A  
3 end
```

「a」が1より小さくない間、「aは1より小さくない」と繰り返し表示する。○回目の「○」には現在の「a」の値を表示させる。

```
1 a = 10  
2 until a < 1 do  
3   puts "aは1より小さくない"  
4   a -= 1  
5 end
```

繰り返し中に使えるメソッド

break

途中で繰り返しを抜けたいときはbreakを使います。

```
1 while true do
2   puts "繰り返しを抜けます。"
3   break
4 end
```

next

次の繰り返しにジャンプしたい時はnextを使います。

```
1 a = 0
2 while a < 10 do
3   a += 1
4   if a == 5
5     puts "次の繰り返しにジャンプします。"
6     next
7   end
8   puts "aは #{a} です。"
9 end
```

redo

現在の繰り返しをやり直したい時はredoを使います。

```
1 a = 0
2 while a < 10 do
3   a += 1
4   puts "aは #{a} です。"
5   if a == 5
6     puts "現在の繰り返しをやり直します。"
7     redo
8   end
9 end
```

乱数

rand

randメソッドは乱数を発生させます。0から値A未満の範囲で乱数を発生させます。

```
1 | rand 値A
```

0から9までの乱数を発生させる。

```
1 | rand 10
```

1から値Aまでの範囲で乱数を発生させる。

```
1 | (rand 値A) + 1
```

1から10までの範囲で乱数を発生させる。

```
1 | (rand 10) + 1
```

gets

getsメソッドを使うと、キーボードからの入力を取得することができます。入力している間はプログラムが停止します。文字を入力したらエンターキーで決定します。

```
1 | gets
```

getsメソッドで取得する値には、改行が含まれています。

```
1 | puts "名前を入力してください"
2 | name = gets
3 | puts "名前は #{name} です。"
```

getsの後に、「chomp」メソッドを付けると、改行を取り除くことができます。

```
1 | puts "名前を入力してください"
2 | name = gets.chomp
3 | puts "名前は #{name} です。"
```

メソッド

同じような処理を何回も使いたい時は、メソッドを使います。メソッドを使うことで、プログラムが見やすくなったり、修正しやすくなったりします。

メソッドは以下のように定義します。

```
1 def メソッド名  
2   処理A  
3   処理B  
4 end
```

次のプログラムは同じ処理が何回も書かれています。

```
1 puts "Hello"  
2 sleep 1  
3 puts "Hello"  
4 sleep 1  
5 puts "Hello"  
6 sleep 1  
7 puts "Hello"  
8 sleep 1
```

メソッドを使うと次のように書くことができます。

```
1 def puts_hello  
2   puts "Hello"  
3   sleep 1  
4 end  
5  
6 puts_hello  
7 puts_hello  
8 puts_hello
```

メソッドを定義する時に引数を付けることができます。

```
1 def メソッド名(引数)  
2   処理A  
3   処理B  
4 end
```

次のプログラムは同じような処理が何回も書かれています。それぞれの処理の違いは、putsで表示する文字列です。

```
1 puts "Hello"  
2 sleep 1  
3 puts "good morning"
```

```
4 sleep 1
5 puts "good evening"
6 sleep 1
7 puts "good night"
8 sleep 1
```

メソッドを使うと次のように書くことができます。

```
1 def puts_greeting(word)
2   puts word
3   sleep 1
4 end
5
6 puts_greeting("good morning")
7 puts_greeting("good evening")
8 puts_greeting("good night")
```

クラスとオブジェクト

オブジェクト

1のような数値や"Hello"のような文字列など、Rubyのプログラムにててくる「もの」を「オブジェクト」と言います。

オブジェクトは、「クラス」から生成されます。

1は数値クラスのオブジェクト、"Hello"は文字列クラスのオブジェクトです。

クラス

クラスには、生成するオブジェクトの性質や、機能を定義します。

クラスの定義

クラスは以下のように定義します。

```
1 class クラス名  
2   クラスの性質、機能など  
3 end
```

オブジェクトの生成

以下のようにしてオブジェクトを生成します。変数に代入することで生成したオブジェクトの操作が簡単になります。

```
1 object = クラス名.new
```

オブジェクトのメソッド

クラスにはメソッドを定義することができます。

```
1 class クラス名  
2   def メソッド名  
3     処理  
4   end  
5 end
```

定義したメソッドは、生成したオブジェクトから呼び出すことができます。

```
1 class Greet  
2   def hello
```

```
3  puts "Hello"  
4  end  
5 end  
6  
7 greet = Greet.new  
8 greet.hello
```