

Rubyプログラミング入門

第1章	Light Weight Language概要	1
1.1	Light Weight Languageとは	2
1.1.1	Light Weight Languageの概要	2
1.1.2	Light Weight Languageの特徴	3
1.1.3	Light Weight Languageによる開発の流れ	4
1.2	代表的なLight Weight Language	5
1.2.1	Perl	5
1.2.2	PHP	6
1.2.3	Python	7
1.2.4	Ruby	8
第2章	Perlの特徴と基本的なプログラム	11
2.1	Perlの概要	12
2.1.1	Perlとは	12
2.1.2	Perlのインストール	13
2.1.3	Perlスクリプトの実行	14
2.2	Perlプログラミング入門	15
2.2.1	文の構造	15
2.2.2	文字列とprint文	16
2.2.3	スカラー	17
2.2.4	配列	18
2.2.5	ハッシュ	19
2.3	プログラムの制御	20
2.3.1	if文	20
2.3.2	unless文	22
2.3.3	for文	24
2.3.4	foreach文	25
2.3.5	サブルーチン	26
2.3.6	変数のスコープ	27
第3章	PHPの特徴と基本的なプログラム	29
3.1	PHPの概要	30
3.1.1	PHPとは	30
3.1.2	PHPのインストール	31
3.1.3	PHPスクリプトの実行準備	32
3.2	PHPプログラミング入門	34
3.2.1	ソースプログラムの構造	34
3.2.2	変数	35
3.2.3	配列	36
3.2.4	連想配列	37
3.2.5	演算子	38

3.3 プログラムの制御	39
3.3.1 if文	39
3.3.2 switch文	40
3.3.3 while文とdo while文	41
3.3.4 for文	42
3.3.5 foreach文	43

第4章 Pythonの特徴と基本的なプログラム **47**

4.1 Pythonの概要	48
4.1.1 Pythonとは	48
4.1.2 Pythonのインストール	49
4.1.3 Pythonスクリプトの実行	50
4.2 Pythonプログラミング入門	51
4.2.1 文の構造	51
4.2.2 文字列とprint文	52
4.2.3 変数	53
4.2.4 リスト	54
4.2.5 タプル	56
4.2.6 辞書型	57
4.3 プログラムの制御	58
4.3.1 if文	58
4.3.2 for文	60
4.3.3 メソッド定義	62

第5章 Rubyの特徴と基本的なプログラム **65**

5.1 Rubyの特徴	66
5.1.1 Rubyとは	66
5.1.2 処理系とバージョン	68
5.1.3 公式サイトとリファレンス	69
5.2 Rubyプログラムの基礎	70
5.2.1 最初のプログラム	70
5.2.2 メソッドとオブジェクト	71
5.2.3 変数と定数	72
5.2.4 定数	74
擬似変数	75
5.2.5 配列とハッシュ	76
5.2.6 コメント	77
5.3 irb	78
5.3.1 irbのインストールと実行	78

第6章 Rubyの基本構文(1)	81
6.1 式と演算子	82
6.1.1 式	82
6.1.2 演算子	83
6.2 条件分岐	85
6.2.1 条件式	85
6.2.3 if~then~end	86
6.2.4 unless	88
6.2.5 if、unlessの後置	89
6.2.6 case	90
第7章 Rubyの基本構文(2)	93
7.1 ループ構造	94
7.1.1 while	94
7.1.2 until	95
7.1.3 for	96
7.1.4 times、each、loop	97
7.1.5 break、next、redo	99
7.2 例外処理	101
7.2.1 begin~rescue~end	101
7.2.2 例外の生成	102
7.3 メソッド	103
7.3.1 メソッドの利用	103
7.3.2 メソッドの定義	104
第8章 オブジェクト指向プログラミング入門	107
8.1 オブジェクト指向の基本	108
8.1.1 オブジェクト指向とは	108
8.2 Rubyにおけるオブジェクト指向	109
8.2.1 オブジェクト	109
8.2.2 クラスとインスタンス	110
8.2.3 継承	112
8.2.4 カプセル化	113
8.2.5 ポリモルフィズム	114
8.2.6 クラスの定義と利用	115

第9章 数値	119
9.1 数値クラス	120
9.1.1 数値 (Numeric) クラスの概要	120
9.1.2 算術演算	121
9.1.3 型変換	123
9.1.4 その他のメソッド	124
9.1.5 ビット演算	125
第10章 文字列、エンコーディング	127
10.1 文字列クラス	128
10.1.1 文字列の生成	128
10.1.2 文字列の長さ	129
10.1.3 文字列の分割と結合	130
10.1.4 文字列の比較	132
10.1.5 文字列の検索	133
10.1.6 主なメソッド	134
10.2 エンコーディング	136
10.2.1 バージョンによる日本語処理の違い	136
10.2.2 エンコーディングの指定	137
10.2.3 エンコーディングの変換 (1.8)	138
10.2.4 エンコーディングの変換 (1.9)	139
10.2.5 ソースコードのエンコーディング	140
第11章 配列	143
11.1 配列	144
11.1.1 配列の概要	144
11.1.2 インデックス	145
11.1.3 配列の操作	146
11.1.4 配列の主なメソッド	147
第12章 ハッシュ	151
12.1 ハッシュ	152
12.1.1 ハッシュの概要	152
12.1.2 値の取り出し	153
12.1.3 ハッシュのメソッド	155

Ruby プログラミング

第1章

Light Weight Language
概要

1.1 Light Weight Languageとは

近年、Light Weight Languageが注目されています。この章では、Light Weight Languageの特徴と、代表的なLight Weight Languageを紹介します。

1.1.1 Light Weight Languageの概要

Light Weight Language (軽量言語) は、C言語、C++言語、Javaといったプログラミング言語と比較して、学習が容易で手軽に扱える言語の総称です。略してLLと呼ばれることもあります。「軽量」というのは、プログラムの負担が軽く、手軽にプログラミングできることから名付けられています。

C言語やJavaによる開発では、ソースコードを作成した後、コンパイルを行って実行ファイル（もしくはクラスファイル）を生成する必要があります。修正が発生すれば、ソースコードを修正した後、再びコンパイルしなければなりません。一方、Webアプリケーションなどで広く使われるようになったスクリプト言語は、インタプリタ型の言語であり、コンパイルを必要としません。ソースコードを書いて、すぐに実行できます。そのため、とりわけ比較的小規模なプログラミングに適しています。

Light Weight Languageの始まりは、1987年に開発がスタートしたPerlとされています。その後、1990年代にPython、Ruby、PHPといった言語が開発され、人気を高めてきました。

※英語圏では、メモリ消費が少ないなど、コンピュータのリソースを消費しないという意味の「軽量」で使われることが多いので注意が必要です。

1.1.2 Light Weight Languageの特徴

Light Weight Languageは、その特徴から、テキスト処理、Webアプリケーションの開発をはじめとして、さまざまな場面での利用が増えています。Perl、Python、PHP、Rubyそれぞれに特徴はありますが、Light Weight Language全般としての特徴は以下のとおりです。

◆スクリプト言語

インタプリタ型の言語であり、実行前にプログラマがコンパイルする必要がありません。そのため、プログラミングを習得するための敷居が低いとされています。

◆動的な型付け

C言語やJavaでは、変数の型を明示的に宣言して利用しますが、Light Weight Languageでは宣言する必要がなく、実行時に型が決まります。

◆正規表現の利用が容易

プログラム内で正規表現を簡単に利用することができることから、テキスト処理に威力を発揮します。

1.1.3 Light Weight Languageによる開発の流れ

一般的な開発の流れは以下のとおりです。

◆開発環境のインストール

言語の実行環境をインストールします。各OS用にバイナリが用意されていますので、インストールは簡単に行えます。必要があれば、GUIを備えた統合開発環境ツールもインストールします。

◆プログラミング

ソースコードを記述します。それぞれの言語が扱える統合開発ツールもありますが、テキストエディタで十分です。多くのテキストエディタでは、プログラミングを支援する機能が搭載されています。

◆実行

コンパイル作業は必要ありません。ソースコードが書かれたファイルを直接実行することができます。したがって、少しずつテストをして動作を確認しながら開発を進めることもできます。実行方法には、コマンドラインから直接実行する、CGIプログラムとして実行する、などの方法があります。

1.2 代表的なLight Weight Language

代表的なLight Weight Languageには、Perl、PHP、Python、Rubyがあります。いずれもオープンソースとして開発されています。

1.2.1 Perl

Perlは、Larry Wall氏によって開発された汎用プログラミング言語で、C言語に似た文法を持っています。WebサイトのCGIプログラミングによく利用されたことで知名度が高まりました。強力な正規表現機能を備えていることから、テキスト処理プログラムを作成するのが容易です。

Perlがインストールされているかどうかは、次のようにして確認できます (CentOS)。

```
$ rpm -q perl
perl-5.8.8-18.el5_3.1
```

Perlについては、第2章で取り上げます。

1.2.2 PHP

PHPは、動的なWebページを作成するためのプログラミング言語で、HTMLファイル内に埋め込む形で利用されます。比較的習得しやすいことや、各種データベースとの連携のしやすさなどから、Webサイトの構築に広く使われています。一般的には、Webサーバにモジュールとして組み込んだ状態で動作させますので、Webプログラミングに特化した言語といえます。

PHPがインストールされているかどうかは、次のようにして確認できます (CentOS)。

```
$ rpm -q php  
php-5.1.6-23.2.el5_3
```

PHPについては、第3章で取り上げます。

1.2.3 Python

Pythonは、Guido van Rossum氏によって開発された汎用プログラミング言語です。日本国内では他のLight Weight Languageと比較して知名度が低いですが、欧米では広く使われています。

Pythonがインストールされているかどうかは、次のようにして確認できます (CentOS)。

```
$ rpm -q python  
python-2.4.3-24.el5
```

Pythonについては、第3章で取り上げます。

1.2.4 Ruby

Rubyは、まつもとゆきひろ氏によって開発された汎用プログラミング言語です。2004年に登場した、RubyによるWebアプリケーションフレームワーク「Ruby on Rails」によって一躍世界的に知られるようになりました。

Rubyがインストールされているかどうかは、次のようにして確認できます (CentOS)。

```
$ rpm -q ruby
ruby-1.8.5-5.el5_2.6
```

UNIX系OSでは、Rubyはほとんどの場合パッケージが用意されているか、デフォルトでインストールされています。Windows用にはいくつかの版がありますが、「ActiveScriptRuby」か「One-Click Ruby Installer for Windows」を利用するのが便利でしょう。

- **ActiveScriptRuby**

URL : <http://www.geocities.co.jp/SiliconValley-PaloAlto/9251/ruby/>

- **One-Click Ruby Installer for Windows**

URL : <http://rubyinstaller.rubyforge.org/wiki/wiki.pl>

UNIX系OSにソースからインストールするには、<http://www.ruby-lang.org/>から最新のアーカイブをダウンロードし、次の手順を実行します。

```
$ tar jxf ruby-1.9.1-p129.tar.bz2
$ cd ruby-1.9.1-p129
$ ./configure
$ make
$ su
# make install
```

詳しくは、アーカイブに添付のドキュメント (README) をご覧ください。

第1章 テスト

問題 1

代表的なLight Weight Languageの名前をいくつか挙げてください。

問題 2

Light Weight Languageの特徴を挙げてください。

Ruby プログラミング

第2章

**Perlの特徴と
基本的なプログラム**

2.1 Perlの概要

2.1.1 Perlとは

Perl (Practical Extraction and Report Language) は、Larry Wall氏によって作られたスクリプト言語です。

の主な特徴は次のとおりです。

- ・オブジェクト指向のスクリプト言語
- ・膨大なライブラリ (CPAN)
- ・動的な型付け
- ・強力なテキスト処理
- ・いろいろなスタイルで書ける許容度の広い構文
- ・GNU GPLおよびArtisticライセンス

Perlは、WebサーバのCGIプログラム、テキスト処理プログラム、システム管理プログラムなどでよく利用されています。

※TMTOWTDI ... There's More Than One Way To Do It (やり方はいくらでもある) と言われます。

2.1.2 Perlのインストール

Perlは、公式サイト (<http://www.perl.org/>) から各種OS用のバイナリやソースを入手できます。多くのLinuxディストリビューションでは、Perlはパッケージとして用意されていますので、パッケージからインストールするのが便利でしょう。CentOSでは次のコマンドを実行します。

```
# yum install perl
```

Debian系ディストリビューションでは、次のようにします。

```
# aptitude install perl
```

Perlがインストールされているかどうかは、次のコマンドで確認できます。

```
$ perl -v

This is perl, v5.8.8 built for i386-linux-
thread-multi

Copyright 1987-2006, Larry Wall

(...)
```

2.1.3 Perlスクリプトの実行

Perlのスクリプトを実行するには、テキストファイルにソースコードを記述し、perlコマンドを実行します。

Perlスクリプトの実行1

```
perl [オプション] <ソースファイル名>
```

オプションとして「-c」を指定すると、文法チェックが行われ、問題がある場合はエラーが表示されます。問題がなければ「syntax OK」と表示されます。また、-wオプションを指定すると、さまざまな警告が表示されます。

Perlスクリプトを引数に指定し、直接実行することもできます。

Perlスクリプトの実行2

```
perl -e 'Perlスクリプト'
```

ソースファイルに実行権限を付けてコマンドのようにプログラムを実行するには、ソースファイルの1行目にインタプリタ指定が必要です。

インタプリタ指定

```
#!/usr/bin/perl
```

Perlの実行形態として多いのは、WebサーバのCGIプログラムとして利用するものでしょう。CGIプログラムとして実行するには、ファイルに実行権限を付け、ドキュメントルート以下に配置します。

WebサーバApacheの場合、設定ファイルhttpd.conf (/etc/httpd/conf/httpd.confなど)で、次のように設定します。この場合、Perlのソースファイルは/var/www/cgi-bin/以下に配置します。

httpd.conf

```
ScriptAlias /cgi-bin/ "/var/www/cgi-bin/"
```

もしくは、次のように設定すると、「~.cgi」ファイルをCGIプログラムとして認識します。

httpd.conf

```
AddHandler cgi-script .cgi
```

2.2 Perlプログラミング入門

2.2.1 文の構造

Perlプログラムは、基本的に1行1文ずつ記述していきます。文末には「;」が必要です。文の区切りとして利用することもできます。

Perlプログラムの例

```
print "Programming Language ";
print "Perl\n";
print "Hello, "; print "World.\n"
```

「#」で始まる行はコメントとみなされます。

コメントの例

```
# この部分はコメント
```

「=pod」～「=cut」の間は、複数行にわたってコメントとみなされます (POD形式)。これはドキュメントを記述するための形式です。

POD形式

```
=POD
コメント
コメント
コメント
=CUT
```

同様に「=catch」から「=cut」までの間は、処理が実行されませんので、複数行のコメントとして使えます。

2.2.2 文字列とprint文

「"」または「'」で囲まれた範囲は文字列として扱われます。文字列はprint文で表示できます。改行が必要な場合は、改行記号「\n」を使います。

hello.pl

```
print "Hello, Perl.\n";  
print 'Hello, Perl.\n';
```

「"」または「'」の違いは、引用符内の特殊文字が解釈されるかどうかです。「'」の場合は、バックスラッシュ記号などもそのまま文字として表示されます。

```
$ perl hello.pl  
Hello, Perl.  
Hello, Perl.\n$
```

なお、「'」の代わりに「q//」、「"」の代わりに「qq//」を使うこともできます。

hello2.pl

```
print qq/Hello, Perl.\n/  
print q/Hello, Perl.\n/;
```

2.2.3 スカラー

Perlの変数には、スカラー、配列、ハッシュがあります。その中で、数値、文字列、バイト列、参照を1つだけ格納できるのがスカラー変数です。初期化は必要なく、値を代入してすぐに利用することができます。スカラー変数名には、頭に「\$」を付けます。

スカラー変数の代入例

```
$str = 'Hello, Perl.';
$val = 100;
$a   = 1;
$a   = 'string';
```

同一の変数に、数値や文字列を代入してもエラーにはなりません。

参照とは、他のスカラー変数や配列、ハッシュへのリファレンスです。参照の例を次に示します。

ref.pl

```
$str = 'Hello,Perl.¥n'
$ref = ¥$str;
print $$ref;
```

なお、「.» 演算子を使って文字列を結合することができます。

str.pl

```
$hello = 'Hello, ';
$lang  = 'Perl';
$str   = $hello.$lang."¥n";
print $str;
```

```
$ perl str.pl
Hello, Perl.
```

※「\$」のような記号のことをシジルといいます。

2.2.4 配列

複数の要素からなるデータのリストが配列です。配列の各要素はスカラーです。配列は「()」で囲み、要素は「,」で区切ります。

配列の例

```
(1, 2, 3, 4, 5)
('Linux', 'Solaris', 'Windows', 'MacOS')
```

配列変数名には頭に「@」を付けます。要素を取り出すインデックスは0から始まります。インデックスに負数を指定すると、末尾からの要素となります（最後の要素が「-1」）。

list.pl

```
@intlist = (1, 2, 3, 4, 5);
@oslist = ('Linux', 'Solaris', 'Windows', 'MacOS');
print $intlist[0], "\n";
print $intlist[-1], "\n";
print $oslist[1], "\n";
```

```
$ perl list.pl
1
5
Solaris
```

2.2.5 ハッシュ

キーと値からなる要素のリストがハッシュです。連想配列ともいいます。キーと値を「=>」で指定しても、キーと値の順に「,」で区切ってもかまいません。また、適度に改行を入れることもできます。

ハッシュの例

```
('one' => 1, 'two' => 2)
('Python',1, 'Ruby',2, 'Perl',3)
```

ハッシュ名は頭に「%」を付けます。値を取り出すときは、キーを{}で囲って指定します。

hash.pl

```
%lang = (
  'Python' => 1,
  'Ruby' => 2,
  'Perl' => 3
);
print %lang, "\n";
print $lang{'Perl'}, "\n";
```

ハッシュの各要素はスカラーです。3行目が「%」ではなく「\$」になっている点に注意してください。

```
$ perl hash.pl
Python1Ruby2Perl3
3
```

2.3 プログラムの制御

2.3.1 if文

条件式の結果が真であるか偽であるかによって処理を分岐させるにはif文を使います。

if文の書式1

```
if (条件式1) {
    実行文1;
} else {
    実行文2;
}
```

if文の書式2

```
if (条件式1) {
    実行文1;
} elsif (条件式2) {
    実行文2;
} else {
    実行文3;
}
```

if文の例を次に示します。\$ARGV[0]は1つめの引数を表します。

iftest1.pl

```
$x = $ARGV[0];

if ($x < 0) {
    print "0以下ですね。 \n";
} elsif ($x == 0) {
    print "0ですね。 \n";
} else {
    print "正の数ですね。 \n";
}
```

```
$ perl iftest1.pl 3
正の数ですね。
$ perl iftest1.pl 0
0ですね。
$ perl iftest1.pl -3
0以下ですね。
```

if文は後置できます。

if文の書式3

```
実行文 if 条件式;
```

以下の2つのif文は同じです。

ifの後置の例

```
print "true\n" if ($x == 0);

if ($x == 0) {
    print "true\n";
}
```

条件式で使える演算子を次表に示します。

[表2-1] 比較演算子

演算子	説明
<code>a < b</code>	aがb未満なら真
<code>a > b</code>	aがbよりも大きければ真
<code>a <= b</code>	aがbと同じか小さければ真
<code>a >= b</code>	aがbと同じか大きければ真
<code>a == b</code>	aとbが等しければ真
<code>a != b</code>	aとbが等しくなければ真
<code>a <=> b</code>	a>bなら1、a==bなら0、a<bなら-1
<code>a eq b</code>	文字列aと文字列bが等しければ真
<code>a ne b</code>	文字列aと文字列bが等しくなければ真

2.3.2 unless文

if文の反対がunless文です。if文と同様、後置もできます。

unless.pl

```
$x = 1;

print "x = 1\n" if ($x == 1);
print "x != 1\n" unless ($x == 1);
```

◆while文とuntil文

while文は、条件が真である間、ループ内の処理を繰り返します。

while文の書式

```
while (条件式){
    実行文;
}
```

while.pl

```
$i = 1;
while ($i < 5){
    print "$i¥n"; $i++;
}
```

```
$ perl while.pl
1
2
3
4
```

一方、until文は、条件が真になるまでループ内の処理を繰り返します。

until文の書式

```
until (条件式) {  
    実行文;  
}
```

until文の書式

```
$i = 1;  
until ($i > 5) {  
    print "$i\n"; $i++;  
}
```

```
$ perl until.pl  
1  
2  
3  
4  
5
```

2.3.3 for文

Perlでは、C言語と同様のfor文でループ処理が行えます。

for文の書式1

```
for (初期化; 条件式; 更新式) {  
    実行文;  
}
```

for1.pl

```
for ($i = 1; $i < 5; $i++) {  
    print "i = $i\n";  
}
```

```
$ perl for1.pl  
i = 1  
i = 2  
i = 3  
i = 4
```

また、次のような使い方もできます。

for2.pl

```
@x = (10, 20, 30, 40, 50);  
for ($i = 0; $i < @x; $i++) {  
    print "$x[$i]\n";  
}
```

```
$ perl for2.pl  
10  
20  
30  
40  
50
```

このプログラムは、次のように記述することもできます。\$_は特殊な変数で、配列やハッシュなどのリストから取り出された要素が一つずつ格納されます。

for3.pl

```
@x = (10, 20, 30, 40, 50);  
for (@x) {  
    print "$_\n";  
}
```

2.3.4 foreach文

foreach文は、リストから要素を抽出しながら、要素の数だけ処理を繰り返します。

foreach文の書式

```
foreach 変数 (リスト){
    実行文;
}
```

foreach.pl

```
@x = (10, 20, 30, 40, 50);
foreach $i (@x){
    print "$i\n";
}
```

```
$ perl foreach.pl
10
20
30
40
50
```

2.3.5 サブルーチン

サブルーチンは、処理をまとめたもので、関数と同等です。return文で指定された値が戻り値として返されます。

サブルーチンの定義

```
sub サブルーチン名 {  
    処理;  
    return 戻り値;  
}
```

サブルーチンを呼び出すには、「&サブルーチン名」とします。呼び出し時に指定した引数は、特殊な配列「@_」に格納されます。

subtest.pl

```
sub add {  
    $a = $_[0];  
    $b = $_[1];  
    return ($a + $b);  
}  
  
print "2 + 3 = ", &add(2, 3), "\n";
```

```
$ perl subtest.pl  
2 + 3 = 5
```

2.3.6 変数のスコープ

変数には有効範囲（スコープ）があります。

◆プライベート変数

サブルーチン内だけで有効です。変数宣言として「my」を指定します。

◆ローカル変数

サブルーチンと、そのサブルーチンから呼び出されるサブルーチン内で有効です。変数宣言として「local」を指定します。

◆グローバル変数

サブルーチン外で宣言した変数は、プログラム内すべてで利用できます。

次の例で、スコープを確認してみます。

scopetest.pl

```
sub func {
    print "Sub routine.¥n";
    my $x = 3;
    print "¥$x = $x¥n";
}

print "Main routine.¥n";
my $x = 1;
print "¥$x = $x¥n";
&func;
print "Main routine.¥n";
print "¥$x = $x¥n";
```

```
$ perl scopetest.pl
Main routine.
$x = 1
Sub routine.
$x = 3
Main routine.
$x = 1
```

第2章 テスト

問題 1

数値、文字列、バイト列などを1つだけ格納できるPerlの変数を何と呼びますか？

問題 2

Perlでは、変数名、配列名、ハッシュ名の頭に、それぞれどのような記号を付けて表しますか？

問題 3

下線部に当てはまるキーワードを記述してください。

```
$ cat arg.pl
$args = $ARGV[0];
if ($args eq "one"){
    print "1¥n";
} _____ ($args eq "two"){
    print "2¥n";
} else{
    print "3¥n";
}
$ perl arg.pl two
2
```

Ruby プログラミング

第3章

**PHPの特徴と
基本的なプログラム**

3.1 PHPの概要

3.1.1 PHPとは

PHP (PHP: Hypertext Preprocessor) は、動的なWebサイトの作成によく利用されるスクリプト言語です。Ruby、Perl、Pythonといった言語が、用途を特定しない汎用言語であるのに対し、PHPはWebサイトの作成に特化しています。PHPの主な特徴は次のとおりです。

- ・ C言語に似た文法
- ・ データベースへの接続が簡単
- ・ フレームワークやライブラリが豊富
- ・ 世界中の多数のサイトで利用
- ・ 小規模なサイトから大規模サイトまで対応
- ・ オープンソースのPHPライセンス

3.1.2 PHPのインストール

PHPは、公式サイト (<http://www.php.net/>) から各種OS用のバイナリやソースを入手できます。多くのLinuxディストリビューションでは、PHPはパッケージとして用意されていますので、パッケージからインストールするのが便利でしょう。CentOSでは次のコマンドを実行します。

```
# yum install php
```

Debian系ディストリビューションでは、次のようにします。

```
# aptitude install php
```

PHPがインストールされているかどうかは、次のようにして確認できます。

```
$ php -v
PHP 5.2.10 (cli) (built: Jun 21 2009 11:10:43)
Copyright (c) 1997-2009 The PHP Group
Zend Engine v2.2.0, Copyright (c) 1998-2009 Zend Technologies
```

PHPのバージョンは、現在、4系と5系があります。4系はサポートが終了していますので、バージョン5系を利用するようにしましょう。

3.1.3 PHPスクリプトの実行準備

PHPスクリプトは通常、Webサーバの公開ディレクトリ内に配置し、Webブラウザからのアクセスによって実行されます。したがって、まずはWebサーバの準備をしましょう。ここでは、~/public_htmlディレクトリ以下にPHPスクリプトを配置するものとします。

まず、公開ディレクトリを作成します。

```
$ mkdir ~/public_html
$ chmod +x ~
```

Apache Webサーバの設定を変更します。/etc/httpd/conf/httpd.confの350～360行目あたりに、以下のような箇所があります。

httpd.conf変更前

```
UserDir disable

#
# To enable requests to /~user/ to serve the user's
public_html
# directory, remove the "UserDir disable" line above, and
uncomment
# the following line instead:
#
#UserDir public_html
```

これを、以下のように変更します。

httpd.conf変更前

```
#UserDir disable

#
# To enable requests to /~user/ to serve the user's
public_html
# directory, remove the "UserDir disable" line above, and
uncomment
# the following line instead:
#
UserDir public_html
```

httpd.confを保存したら、Apacheを再起動します。

```
# service httpd restart
httpd を起動中: [ OK ]
```

サンプルプログラムを記述し、Webブラウザから「http://<ホストのIPアドレス>/<ユーザー名>/hello.php」としてアクセスしてみます。

~/public_html/hello.php

```
<?php
  echo "Hello, PHP!";
?>
```

[図3-1 : hello.php]



なお、コマンドライン上で実行することもできます。

PHPスクリプトの実行

```
php PHPファイル名
```

```
$ php hello.php
Hello, PHP!
```

3.2 PHPプログラミング入門

3.2.1 ソースプログラムの構造

PHPは、HTMLの中に埋め込んで使います。PHPプログラムを記述する書式は次のとおりです。

- ・`<?php` から `?>` までの間
- ・`<?` から `?>` までの間
- ・`<script language="php">` から `</script>` までの間

2番目の書式はXMLの処理命令とバッティングするので好ましくありません。また、3番目の書式は長いので、あまり使われません。

HTMLに埋め込んだ場合の例を以下に示します。拡張子は「`~.php`」です。文字列を出力するには、`print`関数か`echo`文を使います。

sample01.php

```
<html>
  <head>
    <title>PHPテスト</title>
  </head>
  <body>
    <?php
      // コメント
      print "PHPプログラミング入門";
    ?>
  </body>
</html>
```

PHPは、文の末尾に「`;`」が必要です。また、「`//`」以降、および「`」`から「`」`の間はコメントとみなされます。

PHPプログラミング入門

3.2.2 変数

変数には、数値や文字列など、さまざまなデータを格納できます。初期化は必要なく、値を代入してすぐに利用することができます。変数名の規則は次のとおりです。

- ・変数名は「\$」で始まる
- ・1文字目はアルファベットかアンダースコア
- ・2文字目以降はアルファベット、アンダースコアおよび数字いずれか

変数の代入

```
$変数名 = 値
```

以下は変数代入の例です。変数には型があります。

var1.php

```
<?php
    $str = 'Hello, PHP.';
    $int = 1234;
    $float = 3.14159;
    print $str;
    print $int;
    print $float;
?>
```

- ・文字列型 (文字列)
- ・整数型 (整数)
- ・浮動小数点数型 (実数)
- ・論理型 (真偽値: trueもしくはfalseいずれか)

PHPでは、型の扱いが曖昧です。たとえば、文字列として数字を格納していたとしても、式によっては整数として扱うなど、自動的に型変換を行います。

var2.php

```
<?php
    $a = 123; // 整数
    $b = "45"; // 文字列
?>
$a と $b を足すと、<?php print $a + $b; ?> です。
```

\$a と \$b を足すと、168 です。

※変数名は大文字小文字が区別されます。

3.2.3 配列

複数の要素からなるデータを配列といいます。配列は「[]」で囲み、要素は「,」で区切ります。インデックスは0から始まります。

配列の例

```
[1, 2, 3, 4, 5]
['Linux', 'Solaris', 'Windows', 'MacOS']
```

配列を作成し値を代入する方法は2とおりあります。

配列の作成1

```
$配列名[インデックス] = 値;
```

配列の作成2

```
$配列名 = array(値, 値, ...);
```

3.2.4 連想配列

配列のインデックスに、数値以外にも任意の文字列を扱えるようにしたり
ストが連想配列（ハッシュ）です。連想配列の例を示します。

hash.php

```
<?php
  $os["microsoft"] = "Windows";
  $os["sun"] = "Solaris";
  $os["apple"] = "MacOS";
  print $os["microsoft"].", ".$os["sun"].",
  ".$os["apple"];
?>
```

「.」は文字列を連結する演算子です。

```
Windows, Solaris, MacOS
```

3.2.5 演算子

PHPで利用できる演算子をまとめておきます。

[表3-1] 演算子

種類	演算子	説明
算術演算子	+	加算
	-	減算
	*	乗算
	/	除算
	%	剰余
代入演算子	=	代入
	+=	加算して代入
	-=	減算して代入
	*=	乗算して代入
	/=	除算して代入
	%=	剰余算して代入
関係演算子	==	等しければ真
	===	型も含めて等しければ真
	!=	等しくなければ真
	<>	等しくなければ真
	!==	等しくなく型も違えば真
	<	より小さければ真
	>	より大きければ真
	<=	以下なら真
	>=	以上なら真
インクリメント/ デクリメント演算子	\$a++	\$aに1を加算
	\$a--	\$1から1を減算
論理演算子	&&	両方が真の時のみ真
		どちらかが真であれば真
	!	真と偽を反転
連結演算子	.	文字列を連結

3.3 プログラムの制御

3.3.1 if文

条件式の結果がtrue（真）であるかfalse（偽）であるかによって処理を分岐させるにはif文を使います。

if文の書式1

```
if (条件式1) {
    実行文1;
} else {
    実行文2;
}
```

iftest1.php

```
<?php
$a = 1;    // この値をいろいろと変更して試してみることに
if ($a == 1) {
    print '$a は 1 です。';
} else {
    print '$a は 1 以外です。';
}
?>
```

if文の書式2

```
if (条件式1) {
    実行文1;
} elseif (条件式2) {
    実行文2;
} else {
    実行文3;
}
```

iftest2.php

```
<?php
$a = 1;    // この値をいろいろと変更して試してみることに
if ($a == 0) {
    print '$a は 0 です。';
} elseif ($a < 0) {
    print '$a は負の数です。';
} else {
    print '$a は正の数です。';
}
?>
```

3.3.2 switch文

多重分岐をするswitch文は、C言語のswitch文と同様です。

switch文の書式

```
switch (条件式) {
  case 値1:
    実行文1;
    break;
  case 値2:
    実行文2;
    break;

  (...)

  default:
    実行文n;
}
```

switch.php

```
<?php
  $month = 7;    // この値をいろいろと変更して試してみる
  こと
  switch ($month) {
    case 12:
    case 1:
    case 2:
      print "冬です。";
      break;
    case 3:
    case 4:
    case 5:
      print "春です。";
      break;
    case 6:
    case 7:
    case 8:
      print "夏です。";
      break;
    case 9:
    case 10:
    case 11:
      print "秋です。";
      break;
    default:
      print "月は1-12で指定してください。";
  }
?>
```

3.3.3 while文とdo while文

while文は、条件が真である間、ループ内の処理を繰り返します。条件によっては、ループ処理が一度も実行されない場合があります。

while文の書式

```
while (条件式){
    実行文;
}
```

while.php

```
<?php
    $i = 1;
    while ($i < 5){
        print $i++ . '<br>';
    }
?>
```

```
1
2
3
4
```

while文がループ処理の前に条件式を評価するのに対し、do~while文では、ループ内の処理を実行した後で条件式が評価されます。つまり、最低1回はループ内の処理が実行されます。

do while文の書式

```
do {
    実行文;
} while (条件式);
```

dowhile.php

```
<?php
    $i = 1;
    do {
        print $i++ . '<br>';
    } while ($i < 5);
?>
```

```
1
2
3
4
```

3.3.4 for文

PHPでは、C言語と同様のfor文でループ処理が行えます。

for文の書式1

```
for (初期化; 条件式; 更新式) {  
    実行文;  
}
```

for1.php

```
<?php  
    for ($i = 1; $i < 5; $i++) {  
        print "i = $i<br>";  
    }  
?>
```

```
i = 1  
i = 2  
i = 3  
i = 4
```

3.3.5 foreach文

foreach文は、配列や連想配列から要素を抽出しながら、要素の数だけ処理を繰り返します。

foreach文の書式1

```
foreach (配列名 as 変数名) {  
    実行文;  
}
```

foreach1.pl

```
<?php  
    $ary = array("PHP", "Ruby", "Perl",  
    "Python");  
    foreach ($ary as $lang) {  
        print "Programming Language $lang<br>";  
    }  
?>
```

```
Programming Language PHP  
Programming Language Ruby  
Programming Language Perl  
Programming Language Python
```

連想配列の場合は、キーと値を別の変数に読み込みます。

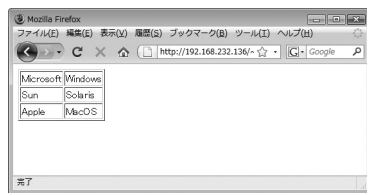
foreach文の書式2

```
foreach (連想配列名 as キー変数 => 要素変数) {  
    実行文;  
}
```

foreach2.php

```
<?php  
    $os["Microsoft"] = "Windows";  
    $os["Sun"] = "Solaris";  
    $os["Apple"] = "MacOS";  
?>  
  
<table border="1">  
    <?php foreach($os as $vendor => $osname) { ?>  
        <tr>  
            <td><?php print($vendor); ?></td>  
            <td><?php print($osname); ?></td>  
        </tr>  
    <?php } ?>  
</table>
```

[図3-2 : foreach2.php]



第3章 テスト

問題 1

PHPの特徴を3つ以上挙げてください。

問題 2

PHPに関する記述として適切なものに○をつけてください。

- () PHPスクリプトは「<!PHP」から「!>」の間に記述する
- () 変数は初期化しないで利用できる
- () 「\$配列名[インデックス]」で要素を取り出せる
- () 連想配列は「@配列名」で表す
- () 文末は「;」を付けるか、改行を入れる

問題 3

配列や連想配列から要素を取り出しながら処理を繰り返すには、何という文を使いますか？

Ruby プログラミング

第4章

**Pythonの特徴と
基本的なプログラム**

4.1 Pythonの概要

4.1.1 Pythonとは

Pythonは、1990年にGuid Van Rossum氏が開発を始めたスクリプト言語です。Pythonの主な特徴は次のとおりです。

- ・オブジェクト指向のスクリプト言語
- ・豊富なライブラリ
- ・動的な型付け
- ・ガベージコレクション
- ・多彩なプラットフォームに対応
- ・インデントによってブロックを表現
- ・対話型シェルを用意
- ・オープンソースのPSFライセンス

Pythonは、Google、Yahoo!、Youtubeをはじめ、多くの企業で活用されています。

4.1.2 Pythonのインストール

Pythonは、公式サイト (<http://www.python.org/>) から各種OS用のバイナリやソースを入手できます。多くのLinuxディストリビューションでは、Pythonはパッケージとして用意されていますので、パッケージからインストールするのが便利でしょう。CentOSでは次のコマンドを実行します。

```
# yum install python
```

Debian系ディストリビューションでは、次のようにします。

```
# aptitude install python
```

Pythonがインストールされているかどうかは、パッケージの有無を調べると分かります。以下はRPM系ディストリビューションでの例です。

```
$ rpm -q python  
python-2.4.3-24.el5
```

Pythonのバージョンは、現在、2.6系と3.1系があります。

4.1.3 Pythonスクリプトの実行

Pythonのスクリプトを実行するには、テキストファイルにソースコードを記述し、pythonコマンドを実行します。

Pythonスクリプトの実行

```
python <ソースファイル名>
```

対話型のシェル (Pythonシェル) も搭載しており、動作を確認するのに役立ちます。Pythonシェルは、pythonコマンドを引数なしで実行すると起動します。Pythonシェルを終了するには、[Ctrl]+[D]を押します。

```
$ python
Python 2.4.3 (#1, Jan 21 2009, 01:10:13)
[GCC 4.1.2 20071124 (Red Hat 4.1.2-42)] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

ソースファイルに実行権限を付けてコマンドのようにプログラムを実行するには、ソースファイルの1行目にインタプリタ指定が必要です。

インタプリタ指定

```
#!/usr/bin/python
```

なお、日本語をファイル内に記述する場合は、以下の文字コード指定をファイルの先頭付近に記述してください。

文字コード指定

```
# coding: UTF-8
```

4.2 Pythonプログラミング入門

4.2.1 文の構造

Pythonプログラムは、基本的に1行1文ずつ記述していきます。文末に「;」は必要ありませんが、文の区切りとして利用することができます。

Pythonプログラムの例

```
print "Prigramming Language"  
print "Python"
```

「#」で始まる行はコメントとみなされます。

コメントの例

```
# この部分はコメント
```

4.2.2 文字列とprint文

「"」または「'」で囲まれた範囲は文字列として扱われます。文字列はprint文で表示できます。

```
>>> print "Hello, World."
Hello, World.
```

基本的な演算は実行され、実行結果が表示されます。

```
>>> print 123 + 456
579
```

文字列の乗算もできます。

```
>>> print "Python" * 3
PythonPythonPython
```

なお、Pythonシェル上では、数式の演算結果はprint文を使わなくても表示されます。

```
>>> 123 + 456
579
```

文字列操作をするメソッドには、次表のようなものがあります。

[表4-1] 文字列関連のメソッド

メソッド	説明
<code>find(str)</code>	文字列 <code>str</code> を検索し、インデックスを返す
<code>replace(old, new)</code>	文字列 <code>old</code> を <code>new</code> に置き換える
<code>split(sep)</code>	文字列を区切り文字 <code>sep</code> で分割したリストを返す
<code>join(seq)</code>	リストなどを連結して文字列とする
<code>lower()</code>	アルファベットを小文字にする
<code>upper()</code>	アルファベットを大文字にする

4.2.3 変数

Pythonでは任意の名前の変数を利用できます。初期化は必要なく、値を代入してすぐに利用することができます。動的な型付けが行われるので、文字列でも数値でも、あらゆるオブジェクトを代入できます。

変数の代入

```
変数名 = 値
```

以下は変数代入の例です。同一の変数に、数値や文字列を代入していることが分かります。

```
>>> var = 1
>>> print var
1
>>> var = 'String'
>>> print var
String
```

複数の変数に、一気に代入することもできます。

```
>>> a = b = c = 0
>>> print a,b,c
0 0 0
>>> d, e = 2, 3
>>> print d,e
2 3
```

4.2.4 リスト

複数の要素からなるデータをリストといいます。他の言語の配列に相当します。リストは「[]」で囲み、要素は「,」で区切ります。

リストの例

```
[1, 2, 3, 4, 5]
['Linux', 'Solaris', 'Windows', 'MacOS']
```

インデックスを指定して要素を取り出します。インデックスの指定方法にはいろいろなパターンがありますので、例を見てください。

```
>>> list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> list[0]
1
>>> list[9]
10
>>> list[:6]
[1, 2, 3, 4, 5, 6]
>>> list[6:]
[7, 8, 9, 10]
>>> list[-3]
8
>>> list[-3:]
[8, 9, 10]
```

なお、文字列もリストとみなされます。

```
>>> word = 'Python and Ruby'
>>> word[0]
'P'
>>> word[-1]
'y'
>>> word[:6]
'Python'
>>> word[7:]
'and Ruby'
>>> word[-4:]
'Ruby'
```

リストの長さはlenメソッドで調べることができます。文字列の長さも同様です。

```
>>> len('Python')
6
>>> len(word)
15
>>> len(list)
10
```

そのほか、リストを操作するメソッドには次表のようなものがあります。

[表4-2] リスト関連のメソッド

メソッド	説明
append(x)	リストの末尾に要素xを追加する
insert(i, x)	1番目の要素としてxを挿入する
remove(x)	値がxの要素を削除する
index(x)	値がxの最初の要素のインデックスを返す
count(x)	リストの中にあるxの個数を返す
sort()	リストをソートする(破壊的)
reverse()	リストを逆順にソートする(破壊的)

4.2.5 タプル

タプルはリストに似たデータ構造です。リストが内容を変更できるのに対し、タプルは変更することができません。タプルは、リストの「[]」の代わりに「()」を使います。

タプルの例

```
(1, 2, 3, 4, 5)
('Linux', 'Solaris', 'Windows', 'MacOS')
```

```
>>> tuple = (1, 2, 3, 4, 5)
>>> print tuple
(1, 2, 3, 4, 5)
>>> tuple[2]
3
```

この例のように、インデックスは「[]」で指定します。リストと異なり、内容を変更することはできません。

```
>>> tuple[2] = 0
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object does not support item assignment
```

リストと同様のメソッドも使えますが、内容を変更するメソッドは利用できません。

4.2.6 辞書型

他の言語におけるハッシュや連想配列に相当するものが辞書型（ディクショナリ型）です。要素はキーと値がペアになっています。インデックスの代わりにキーで指定する点がリストやタプルと異なる点です。

辞書型の例

```
{'Linux':1, 'Solaris':2, 'Windows':3, 'MacOS':4}
{'Python':1, 'Ruby':2, 'Perl':3}
```

```
>>> os = {'Linux':1, 'Solaris':2, 'Windows':3, 'MacOS':4}
>>> os['Linux']
```

辞書型関連のメソッドには、次表のようなものがあります。

[表4-3] 辞書型関連のメソッド

メソッド	説明
<code>has_key(key)</code>	キーにkeyが存在すればtrueを返す
<code>keys()</code>	キー一覧をリストにして返す
<code>items()</code>	キーと値のペアをタプルとし、タプルのリストを返す
<code>get(key)</code>	キーkeyに該当する値を返す

4.3 プログラムの制御

4.3.1 if文

条件式の結果がTrue（真）であるかFalse（偽）であるかによって処理を分岐させるにはif文を使います。以下の2点に注意してください。

- ・ブロックをインデントで表す
- ・文末に「:」を付ける

if文の書式

```
if 条件式1:
    実行文1
elif 条件式2:
    実行文2
else:
    実行文3
```

if文の例を次に示します。raw_inputメソッドは標準入力から文字列を入力します。intメソッドは文字列を整数に変換します。

iftest.py

```
# coding: UTF-8
x = int(raw_input("整数を入力してください: "))

if x < 0:
    print "0以下ですね。"
elif x == 0:
    print "0ですね。"
else:
    print "正の数ですね。"
```

```
$ python iftest.py
整数を入力してください: 20
正の数ですね。
$ python iftest.py
整数を入力してください: -1
0以下ですね。
$ python iftest.py
整数を入力してください: 0
0ですね。
```

条件式で使える演算子を次表に示します。

[表4-4] 比較演算子

比較演算子	説明
<code>a == b</code>	aとbは等しい
<code>a != b</code> 、 <code>a <> b</code>	aとbは等しくない
<code>a < b</code>	aはb未満
<code>a <= b</code>	aはb以下
<code>a > b</code>	aはbより大きい
<code>a >= b</code>	aはb以上もしくは等しい
<code>not a == b</code>	aとbは等しくない

4.3.2 for文

Pythonのfor文は、リストにある値を変数に代入しながらループ内の処理を実行します。リストの末尾には「:」を付けます。ループの範囲はインデントで区別します。

for文の書式

```
for 変数 in リスト:  
    実行文
```

fortest.py

```
for i in [1, 2, 3, 4, 5]:  
    print i  
for i in ['Linux', 'Windows', 'UNIX']:  
    print i  
for i in range(5):  
    print i
```

rangeメソッドは、連続した数値のリストを生成します。

```
$ python fortest.py  
1  
2  
3  
4  
5  
Linux  
Windows  
UNIX  
0  
1  
2  
3  
4
```

次の例では、文字列の文字を1文字ずつ縦に表示します。

listtest.py

```
# coding: UTF-8
str = raw_input("文字列を入力してください：")

for i in range(len(str)):
    print str[i]
```

```
$ python listtest.py
文字列を入力してください：Python
P
Y
t
h
o
n
```

4.3.3 メソッド定義

メソッドはdef文で定義できます。スクリプト内で、最初にメソッドが使われるより以前にメソッド定義が必要です。

メソッド定義

```
def メソッド名(引数):  
    実行文
```

method.py

```
def hello(str):  
    print "hello, " + str  
  
hello('world')  
lang = 'Python'  
hello(lang)
```

```
$ python method.py  
hello, world  
hello, Python
```

第4章 テスト

問題 1

Pythonの特徴を3つ以上挙げてください。

問題 2

Pythonを対話的に実行できるPythonシェルを起動するコマンドを記述してください。

問題 3

Pythonに関する記述として適切なものに○をつけてください。

- () 変数は初期化しないで利用できる
- () 変数名には「\$」を付けて表す
- () 複数の要素からなるデータをリストという
- () ハッシュや連想配列に相当するデータ構造をタプルという

Ruby プログラミング

第5章

**Rubyの特徴と
基本的なプログラム**

5.1 Rubyの特徴

ここでは、Rubyの概要と特徴を取り上げます。

5.1.1 Rubyとは

Rubyは日本で生まれた純粋なオブジェクト指向言語です。まつもとゆきひろ氏によって1993年に公開され、オープンソースのコミュニティによって開発が続けられています。

2000年頃まではほとんど国内でのみ利用される存在であったRubyですが、達人プログラマとして知られるDavid ThomasとAndrew HuntによるRuby初の英語書籍「Programming Ruby」発刊後、急速に海外にも広がっていききました。その流れを決定づけたのが、2004年に発表されたWebアプリケーションフレームワーク「Ruby on Rails」です。15分でプロダクトウェアを作るデモを公開したり、Javaよりも圧倒的に生産性が高いことをアピールすることで、Rubyの知名度は爆発的に高まることとなりました。

現在では、Rubyをビジネス分野で利用することは珍しくなくなりました。多くのLinuxディストリビューションには、標準でRubyの処理系が搭載されるようになってきています。

以下、Rubyの特徴を列挙します。

◆オブジェクト指向言語

RubyはC++言語やJavaのような、クラスベースのオブジェクト指向言語です。Rubyでは、数値、文字列、入出力といったすべてのデータがオブジェクトとして扱われます。Javaにおけるプリミティブ型のようなものはありません。クラスそのものもオブジェクトです。

◆インタープリタ型言語

Rubyプログラムはコンパイルしなくても実行することができます。そのため、C言語やC++言語のようなコンパイル型の言語と比較すると、処理速度は劣ります。しかし、さまざまな処理系が登場し、かつてほど処理速度がネックになるといったことは減ってきています。また、最新版のバージョン1.9系では、処理速度が大幅に向上しています。

◆オープンソース

Rubyはオープンソースとして開発されています。誰もが開発に参加することができ、ソースコードを読むことができます。もちろん、ネット上からダウンロードして無償で利用することができます。ライセンスは、GNU GPLもしくはArtistic Licenseに似た独自ライセンスいずれかを選択することができます。

※ <http://www.gnu.org/licenses/gpl.html>
※ Perlの開発者Larry Wall氏が作成したライセンス。Perlなどで採用されている。

◆用途

汎用プログラミング言語なので、システムツールからGUIアプリケーションまで、さまざまな用途に利用できますが、もっとも広く使われているのはWebアプリケーションの開発でしょう。開発フレームワーク「Ruby on Rails」の爆発的なヒットにより、Webアプリケーション開発言語として注目を集めました。

◆ガベージコレクション

オブジェクトが保持しているメモリが不要になると、処理系が自動的に解放します。プログラマがメモリ管理を気にする必要はありません。

◆多くのプラットフォームで動作

LinuxなどのUNIX系OS、Windows、Mac OS Xはもちろん、MS-DOS、OS/2、BeOSなどでも動作します。

5.1.2 処理系とバージョン

現在、Rubyにはいくつかの処理系があります。主な処理系を紹介します。

◆MRI (Matz's Ruby Implimentation)

まつもとゆきひろ氏 (Matz) によって開発されている処理系です。もともとはMRIのみが唯一のRuby実装だったので、Rubyといえばこの処理系を意味しました。C言語で実装されていることからCRubyと呼ばれることもあります。

◆JRuby

Javaの仮想マシン上で動作するRubyで、MRIに対して高い互換性を持っています。Javaの豊富なクラスライブラリなどを利用できることや、Java仮想マシンのノウハウも使えることから、人気が高まってきています。

◆IronRuby

Microsoftの.NET上で利用できる処理系です。

◆MacRuby

Objective-Cで実装され、Mac OS Xで動作する処理系です。

◆Ruby Enterprise Edition (REE)

MRIをベースに作られた実行環境で、メモリの使用量が少なく、実行パフォーマンスも良くなっています。Ruby on Railsの実行環境に向いています。

これらの処理系の中でもっとも基本となるのがMRIです。以下、処理系を指して「Ruby」という場合はMRIを表すものとします。

Rubyのバージョンは、「1.8.7」「1.9.1」のように、3桁の数値で構成されています。最初の数値はメジャーバージョンです。2番目の数値はマイナーバージョンで、偶数は安定版、奇数は開発版を意味します。ただし1.9系だけは例外で、「1.9.0」は開発版、「1.9.1」以上は安定版となっています。

また、「1.8.7-p160」のように、バージョン番号の後にパッチレベルが追加されている場合、数値が大きいものほどバグ修正パッチが適用されています。

5.1.3 公式サイトとリファレンス

Rubyの情報を得るために役立つ、定番の情報源は以下のとおりです。

◆公式サイト

Rubyの公式サイトでは、最新版のRubyをダウンロードできるほか、Rubyに関する情報が多数掲載されています。

URL : <http://www.ruby-lang.org/ja/>

◆メーリングリスト

用途に合わせていくつかのメーリングリストが運営されています。

- **ruby-list (日本語)**
Rubyでプログラミングをする人が情報交換をするためのMLです。
- **ruby-dev (日本語)**
Rubyの開発者向けMLです。
- **ruby-ext (日本語)**
Rubyの拡張モジュールに関するMLです。
- **ruby-math**
数学関連のMLです。
- **ruby-talk (英語)**
ruby-listの英語版MLです。
- **ruby-core**
ruby-devの英語版MLです。

URL : <http://www.ruby-lang.org/ja/community/mailling-lists/>

◆Rubyist Magazine

日本Rubyの会によって制作されているWebマガジンです。

URL : <http://jp.rubyist.net/magazine/>

◆Rubyリファレンスマニュアル

Rubyのプログラミングで常に参照するリファレンスマニュアルです。一括ダウンロードもできるので、常に参照するようにするとよいでしょう。

URL : <http://www.ruby-lang.org/ja/man/>

5.2 Rubyプログラムの基礎

実際にRubyのプログラムを作成し、実行しながら、Rubyプログラミングの基本を確認します。

5.2.1 最初のプログラム

はじめてのRubyプログラムを作成する前に、Rubyがインストールされているか確認しておきましょう。端末を開き、以下のように入力します。

```
$ ruby -v
ruby 1.8.5 (2006-08-25) [i386-linux]
```

このように、Rubyのバージョンが表示されればインストールが完了しています。

それでは、適当なテキストエディタを使ってソースプログラムを作成します。定番のhello worldです。ファイル名は「hello.rb」とします。一般的に、Rubyプログラムの拡張子は「~.rb」とします。

```
puts("hello, world.")
```

これは、putsというメソッドを使って文字列を出力するプログラムです。多くのプログラミング言語では、行末に「;」が必要ですが、Rubyは改行が文末を表しますので、「;」は不要です。まずはプログラムを実行してみましょう。

```
$ ruby hello.rb
hello, world.
```

1行だけのプログラムであれば、次のようにコマンドライン上で実行することもできます。

```
$ ruby -e 'puts ("hello, world.")'
hello, world.
```

次に、このプログラムを詳しく見ていくことにします。

※「;」を付けてもエラーにはなりません。また、「;」を使って1行に複数の文を記述することもできます。

5.2.2 メソッドとオブジェクト

先のプログラムを見てみましょう。この1行は、「puts」というメソッドと、その引数から構成されています。

```
puts ("hello, world.")
```

メソッドはオブジェクトに対する手続きのことで、C言語における関数のようなものです。メソッドに与えるデータのことを引数といいます。putsメソッドは、引数として指定された文字列を出力するメソッドです。

メソッドの()は、なくても混乱しないような場合は省略することができます。つまり、以下のように記述してもかまいませんし、こちらの方が一般的な表記です。

```
puts "hello, world."
```

※定数は英大文字で始めます。

5.2.3 変数と定数

Rubyの変数は、任意のオブジェクトへの参照が格納されています。変数を使用するときには、あらかじめ宣言をする必要がありません。変数への代入が発生した時点で変数が作られ、動的に型が決まります。変数名にはアルファベット、数字、アンダースコア (_) が使えます。最初の1文字は英小文字で始めます。

```
var = "hello, ruby."  
puts var  
  
var = 1  
puts var  
  
var = 3.14159  
puts var
```

```
$ ruby var1.rb  
hello, ruby.  
1  
3.14159
```

なお、変数は「#{変数名}」とすると、二重引用符内で変数を展開できます。

```
var = foo  
puts "var : #{var}"
```

```
var : foo
```

Rubyの変数には、ローカル変数、インスタンス変数、クラス変数、グローバル変数があります。それぞれ変数にアクセスできる範囲（スコープ）が異なります。ここでは簡単にそれぞれの変数を簡単に説明します。

◆ローカル変数

ローカルスコープからアクセスできます。変数名には任意の英子文字と「_」を使用できます。

◆インスタンス変数

当該クラスのメソッドからのみアクセスできます。変数名は「@」で始まります。

◆クラス変数

当該クラスとサブクラス、および各クラスのインスタンスメソッドからアクセスできます。変数名は「@@」で始まります。

◆グローバル変数

スクリプト内のどこからでもアクセスできますが、バグの原因となりやすいので、できるだけ使わない方がよいでしょう。変数名は「\$」で始まります。

このように、Rubyでは変数名を見ただけで、どの変数なのかが識別できるようになっています。

※実際には、警告は出ますが変更することができます。

5.2.4 定数

変数とは異なり、オブジェクトへの参照が変更できないものが定数です。最初の1文字は英大文字で始めます。

```
Ver = "1.8.7"
puts Ver           #=> "1.8.7"
```

また、処理系の情報を格納している組み込み定数もあります。

[表5-1] 組み込み定数

組み込み定数	説明
ARGF	引数または標準入力により構成される仮想ファイル
ARGV	コマンドライン引数
ENV	環境変数
RUBY_VERSION	Ruby処理系のバージョン
RUBY_RELEASE_DATE	Ruby処理系のリリース日
RUBY_PLATFORM	実行環境
STDIN	標準入力
STDOUT	標準出力
STDERR	標準エラー出力
DATA	__END__ 以降を扱うオブジェクト

擬似変数

特殊な変数として、擬似変数というものがあります。変数とはいいますが、代入をすることはできません。

◆nil

値がないことを示します。他の言語でのnullに近いものです。真偽値では「偽」を表します。

◆true

真偽値の「真」を表します。

◆false

真偽値の「偽」を表します。

◆self

現在のメソッドの実行主体を示します。JavaやC#におけるthisのようなものです。

◆__FILE__

実行中のプログラムのソースファイル名を表します。

◆__LINE__

実行中のプログラムの行番号を表します。

※正確にはオブジェクトへの参照を格納します。

5.2.5 配列とハッシュ

配列とハッシュについては11章と12章で取り上げますが、ここでも簡単に紹介します。配列には、数値でも文字列でも、任意のオブジェクトを格納することができます。配列内のデータは、インデックス（添字）を使って場所を指定します。

```
配列名 = [要素1, 要素2 ...]
```

配列の使用例です。pメソッドは、引数のオブジェクトの内容を表示するメソッドです。

```
num = [1, "2", 3.1415, "Ruby", 5]
p num
p num[1]
p num[2..4]
```

```
$ ruby array1.rb
[1, "2", 3.1415, "Ruby", 5]
"2"
[3.1415, "Ruby", 5]
```

5.2.6 コメント

行頭に「#」があると、行末まではコメントとみなされます。

```
# この部分はコメント  
puts "hello, ruby."
```

「=begin」で始まる行から「=end」で始まる行の間もコメントとみなされます。これは埋め込みドキュメントといい、外部のツールを使ってテキストを抽出する際などに利用されます。

```
print "hello, "  
=begin  
この部分は  
コメントです。  
=end  
puts "ruby."
```

また、「__END__」という行がソースコード中に現れると、それ以後の行はすべてコメントとみなされます。

```
puts "hello, ruby."  
  
__END__  
  
ここはコメントです。  
puts "この行は実行されません"
```

5.3 irb

Rubyには、`irb` (Interactive Ruby) という対話型インターフェースがあります。ちょっとしたプログラムのテストや動作確認などに利用できます。

5.3.1 irbのインストールと実行

CentOSでは、`irb`は`ruby-irb`パッケージとして用意されています。標準ではインストールされていないので、次のコマンドでインストールします。

```
# yum install ruby-irb -y
```

`irb`コマンドを実行すると、プロンプトが変わり、対話的にRubyプログラムを入力できるようになります。

```
$ irb
irb(main):001:0>
```

たとえば「`puts "Hello, Ruby."`」を実行してみましょう。

```
irb(main):001:0> puts "Hello, Ruby."
Hello, Ruby.
=> nil
```

プログラムを入力してEnterキーを押すと、すぐに実行結果が表示されます。次の行にある「`=> nil`」は戻り値を表します。`nil`は何も戻り値が返ってこなかったことを表します。

`irb`を終了するには「`quit`」もしくは「`exit`」を入力します。

```
irb(main):002:0>
```

第5章 テスト

問題 1

Rubyの特徴を3つ以上挙げてください。

問題 2

変数varを二重引用符内で展開するには、どう記述するのが適切ですか。

- A. "var : #{var}"
- B. "var : \${var}"
- C. "var : &{var}"

問題 3

定数名を決める際に注意すべきことはどれですか。

- A. 最初の1文字目は英大文字とする
- B. 最初の1文字目は「@」とする
- C. 最初の1文字目は「#」とする

問題 4

真偽値で「偽」を表す2つの疑似変数名を記述してください。

Ruby プログラミング

第6章

Rubyの基本構文(1)

6.1 式と演算子

ここでは、プログラムの基本的な構成要素となる式と、主な演算子を紹介します。

6.1.1 式

評価することによって何らかの値を返す単位が式です。変数、値、演算子、メソッドなどを組み合わせたものと見てもよいでしょう。式の例と、返される値を以下に示します。

- `1 + 2` `#=> 3`
- `"Ruby"` `#=> "Ruby"` という文字列
- `var` `#=>` 変数`var`の内容
- `a < 0` `#=>` `true`もしくは`false`
- `var.size` `#=>` 変数`var`のサイズ (`size`メソッドの戻り値)

Rubyプログラムの実行は、式の評価によって行われます。

6.1.2 演算子

Rubyで利用できる演算子は、他のプログラミング言語と同様です。代表的な演算子を使った例を以下に示します。

```
irb(main):001:0> 5 + 3
=> 8
irb(main):002:0> 5 - 3
=> 2
irb(main):003:0> 5 * 3
=> 15
irb(main):004:0> 5 / 3
=> 1
irb(main):005:0> 5 % 3
=> 2
```

Rubyでは、インクリメント演算子 (++) やデクリメント演算子 (--) がありません。「+=」「-=」などを利用します。

```
irb(main):006:0> i = 1
=> 1
irb(main):007:0> i += 3
=> 4
```

演算子には優先順位があります。表6-1に演算子をまとめます。上の方ほど優先度が高くなります。

[表6-1] 演算子の優先順位

演算子	優先度	
[]	高い	
+ ! ~		
**		
-		
* / %		
+ -		
<< >>		
&		↑
^		低い
> >= < <=		
<=> == === != =~ !~		
&&		
.. ...		
? :		
"= += -= *= /= %= =		
~= <<= >>= &&= = **="		
not		
and or		

なお、Rubyの演算子は、多くがメソッドのシンタックスシュガーです。たとえば「1 + 2」というのは「1.+(2)」、つまり、1という数値オブジェクトの+メソッドを利用しているのです。

6.2 条件分岐

他の言語と同様、Rubyにもifやcaseがあります。

6.2.1 条件式

「aとbは等しいか」「aはbより大きいか」「aかつbであるか」のように、条件を調べるための式のことを条件式といいます。Rubyでは、条件式で使われる演算子として次表のものがああります。

[表6-2] 比較演算子と論理演算子

演算子	説明
==	等しい
===	等しい (case式で用いられる)
!=	等しくない
>	右辺より左辺が大きい
>=	右辺より左辺が大きいか、等しい
<	左辺より右辺が大きい
<=	左辺より右辺が大きいか、等しい
<=>	左辺が小さければ-1、等しければ0、大きければ1
=~	正規表現のパターンマッチで一致する
!~	正規表現のパターンマッチで一致しない
&&, and	論理積
, or	論理和
!, not	否定

条件式は「式」ですので、それ自体が値を持ちます。

```
irb(main):001:0> a = 2
=> 2
irb(main):002:0> a < 3
=> true
irb(main):003:0> a < 1
=> false
```

このように、条件が正しければ「true」、正しくなければ「false」となります。条件式の値は、変数に代入することもできます。

```
irb(main):004:0> c = a < 3
=> true
irb(main):005:0> p c
true
```

※「if文」ではなく「if式」と呼ぶのは、if式自体が値を持つからです。

6.2.3 if~then~end

もっとも基本となる条件分岐がif式です。

if式の書式1

```
if 条件式 [then]
  条件式が真のときに実行するコード
end
```

elseも利用できます。

if式の書式2

```
if 条件式 [then]
  条件式が真のときに実行する処理
else
  条件式が偽のときに実行する処理
end
```

なお、条件式の直後に改行がある場合、thenは省略することができます。次の例は、入力した数値を判定するプログラムです。getsはコマンドラインから1行入力するメソッド、to_iは整数に変換するメソッドです。

if1.rb

```
puts "任意の整数を入力してください。"
num = gets.to_i

if num < 0
  puts "入力された数値は負の数です。"
else
  puts "入力された数値は0以上の数です。"
end
```

さらに条件を分岐させる場合はelsifを使います。「elseif」ではないので注意してください。

if式の書式3

```
if 条件式1 [then]
  条件式1が真のときに実行する処理
elsif 条件式2 [then]
  条件式2が真のときに実行する処理
else
  条件式1も条件式2も真でないときに実行する処理
end
```

if2.rb

```
puts "任意の整数を入力してください。"  
num = gets.to_i  
  
if num < 0  
  puts "入力された数値は負の数です。"  
elsif num > 0  
  puts "入力された数値は正の数です。"  
else  
  puts "入力された数値は0です。"  
end
```

ところで、条件式に数値が指定された場合はどうなるか確認してみましょう。

if3.rb

```
num = 0  
if num  
  puts "true"  
else  
  puts "false"  
end
```

実行すると「true」が表示されます。つまり、値が0の場合はtrueとみなされます。同様に、変数の値が空文字列の場合もtrueとみなされます。

6.2.4 unless

unless式はif式の否定形です。if式で代用できますが、「～ではないなら」という場合はunless式で書いた方が直感的に分かりやすいでしょう。

unless式の書式

```
unless 条件式
  条件式が正しくない時の処理
end
```

unless式の例を示します。

unlesstest.rb

```
puts "任意の整数を入力してください。"
num = gets.to_i

unless num == 0
  puts "ゼロ以外の数値です。"
end
```

このプログラムは次の例とまったく同じです。

iftest3.rb

```
puts "任意の整数を入力してください。"
num = gets.to_i

if num != 0
  puts "ゼロ以外の数値です。"
end
```

6.2.5 if、unlessの後置

if式やunless式を後置することもできます (if/unless修飾子)。

iftest4.rb

```
puts "任意の整数を入力してください。"  
num = gets.to_i  
  
puts "偶数です。" if num % 2 == 0  
puts "奇数です。" if num % 2 == 1
```

これは、次のプログラムと同じです。

iftest5.rb

```
puts "任意の整数を入力してください。"  
num = gets.to_i  
  
if num % 2 == 0  
  puts "偶数です。"  
end  
if num % 2 == 1  
  puts "奇数です。"  
end
```

1行で書く場合は、次のように「then」が必要となります。thenは「:」におきかえてもかまいません。

iftest6.rb

```
puts "任意の整数を入力してください。"  
num = gets.to_i  
  
if num % 2 == 0 then puts "偶数です。" end  
if num % 2 == 1 :   puts "奇数です。" end
```

6.2.6 case

条件がたくさんある場合は、if式よりもcase式を使った方が見やすくなります。case式の書式は2つあります。

case式の書式1

```
case
when 条件式1 [then]
  条件式1が正しいときに実行する処理
when 条件式2
  条件式2が正しいときに実行する処理
(...)
else
  いずれの条件式も正しくないときに実行する処理
end
```

case式の書式2

```
case 変数
when 式1 [then]
  変数が式1と等しいときに実行する処理
when 式2
  変数が式2と等しいときに実行する処理
(...)
else
  変数がいずれの式とも一致しないときに実行する処理
end
```

次の例では、入力した月の数値に対応した季節を表示します。printメソッドは、putsと同様に文字列を出力しますが、行末に改行されません。

casetest.rb

```
print "月を入力してください："
month = gets.to_i
print "#{month}月は"

case month
when 12,1,2
  puts "冬です。"
when 3,4,5
  puts "春です。"
when 6,7,8
  puts "夏です。"
when 9,10,11
  puts "秋です。"
else
  puts "無効な値です。1~12の間で指定してください。"
end
```

式は「,」で区切って複数指定できる点、C言語やJavaのような「break」が必要ない点に注意してください。

```
$ ruby casetest.rb
月を入力してください: 7
7月は夏です。
$ ruby casetest.rb
月を入力してください: 13
13月は無効な値です。1~12の間で指定してください。
```

また、whenの後に改行しないで「:」で区切って記述することもできます。

casetest2.rb

```
print "月を入力してください: "
month = gets.to_i
print "#{month}月は"

case month
  when 12,1,2    : puts "冬です。"
  when 3,4,5    : puts "春です。"
  when 6,7,8    : puts "夏です。"
  when 9,10,11  : puts "秋です。"
  else          : puts "無効な値です。1~12の間で指定
してください。"
end
```

次のように、whenには範囲を指定することもできます。

範囲の指定

```
when 3..5      : puts "春です。"
when 6..8      : puts "夏です。"
when 9..11     : puts "秋です。"
```

演習問題

P.81のiftest6.rbを、case~whenを使って書き換えなさい。

第6章 テスト

問題 1

Rubyには「++」のようなインクリメント演算子がありません。「i++」(変数iの値を1増加させる)と同じことをRubyで記述するにはどうすればよいですか？

問題 2

「a < 3」という式が満たされる場合、式自体が持つ値は何ですか？

問題 3

下線部に当てはまるキーワードを記述してください。

```
if num < 0
  puts "入力された数値は負の数です。"
  _____ num > 0
  puts "入力された数値は正の数です。"
  _____
  puts "入力された数値は0です。"
end
```

問題 4

下線部に当てはまるキーワードを記述してください。

```
num = gets.to_i
puts "ゼロ以外の数値です。" _____ num == 0
```

問題 5

下線部に当てはまるキーワードを記述してください。いずれも同じキーワードが当てはまります。

```
case month
  _____ 12, 1, 2
    puts "冬です。"
  _____ 3, 4, 5
    puts "春です。"
  _____ 6, 7, 8
    puts "夏です。"
  _____ 9, 10, 11
    puts "秋です。"
  else
    puts "無効な値です。1~12の間で指定してください。"
  end
```

Ruby プログラミング

第7章

Rubyの基本構文(2)

7.1 ループ構造

何度も処理を繰り返したい場合はループ処理を行います。他の言語と同様、Rubyにもwhile、forなどがありますが、Ruby独自の便利なメソッドもあります。

7.1.1 while

while式は、条件式が満たされている (true) 間、do~endまでを繰り返します。

while式の書式

```
while 条件式 [do]
  処理
end
```

if式の場合のthenと同様、条件式の直後に改行があればdoは省略できます。

whilettest.rb

```
i = 0
while i < 10
  puts i
  i += 1
end
```

```
$ ruby whilettest.rb
0
1
2
3
4
5
6
7
8
9
```

7.1.2 until

until式を使っても、while式と同様のループ処理ができます。ただし、until式は、条件式が正しくない間だけループ処理が行われます。

until式の様式

```
until 条件式 [do]
  処理
end
```

untiltest.rb

```
i = 0
until i > 10
  puts i
  i += 1
end
```

```
$ ruby untiltest.rb
0
1
2
3
4
5
6
7
8
9
10
```

7.1.3 for

リストに指定された値を変数に代入しながら、do~endの間を繰り返します。リストには、配列やハッシュ、範囲オブジェクトなどが利用できます。リストの直後に改行がある場合は「do」を省略できます。

for式の様式

```
for 変数 in リスト do
  処理
end
```

次の例では、配列の内容を変数langに代入しながら処理を行います。

fortest1.rb

```
for lang in ["Ruby", "PHP", "Perl", "Python",
"Java"] do
  puts "Programming Language #{lang}"
end
```

```
$ ruby fortest1.rb
Programming Language Ruby
Programming Language PHP
Programming Language Perl
Programming Language Python
Programming Language Java
```

一般的なプログラミング言語と同様の使い方もできます。次の例では、forを使って10回のループ処理を行っています。「(1..10)」は、1から10までの範囲を表します。

fortest2.rb

```
sum = 0
for i in (1..10)
  sum += i
end
puts sum
```

```
$ ruby fortest2.rb
55
```

7.1.4 times、each、loop

Rubyでは、単純なループ処理にwhileやforを使うことはあまりありません。繰り返し処理を行うさまざまなメソッドが用意されているからです。そのようなメソッドをイテレータといいます。

※イテレータについては「Ruby 中級編」で詳しく取り上げます。

◆timesメソッド

timesメソッドは一定回数の繰り返しの利用します。次の例では、doとdoneの間を5回繰り返します。

timestest1.rb

```
5.times do
  puts "Hello!"
end
```

```
$ ruby timestest1.rb
Hello!
Hello!
Hello!
Hello!
Hello!
```

ループ処理の中で変数を扱うこともできます。次の例では、繰り返しが行われる度に、変数*i*に数値が格納され、ループ内で利用されます。

timestest2.rb

```
5.times do |i|
  puts i
end
```

```
$ ruby timestest2.rb
0
1
2
3
4
```

do~endの代わりに「{」「}」を使うこともできます。

timestest3.rb

```
5.times {
  puts "Hello!"
}

5.times { |i|
  puts i
}
```

◆eachメソッド

オブジェクトを一つずつ取り出してループ内で利用するには、eachメソッドが便利です。次の例では、配列内の要素を一つずつ取り出し、それを使って文字列を出力しています。

eachtest.rb

```
languages = ["Ruby", "PHP", "Perl", "Python",
"Java"]
languages.each { |lang|
  puts "Programming Language #{lang}"
}
```

```
$ ruby eachtest.rb
Programming Language Ruby
Programming Language PHP
Programming Language Perl
Programming Language Python
Programming Language Java
```

◆loopメソッド

loopメソッドは、単純な繰り返しに利用します。ループ内にループを終了する処理がないと、無限ループになってしまいます。

infinitemloop.rb

```
# 無限ループを停止するにはCtrl+C
loop {
  print "BABEL "
}
```

7.1.5 break、next、redo

繰り返し処理を途中で中断したり、ループを1回飛ばしたりする制御処理用に、break、next、redoが用意されています。

◆break

繰り返し処理を中断します。

break.rb

```
5.times { |i|
  if i == 3
    break
  end
  puts i
}
```

```
$ ruby break.rb
0
1
2
```

◆next

ループ中で、next以降の処理を飛ばして、次のループを実行します。

next.rb

```
5.times { |i|
  if i == 3
    next
  end
  puts i
}
```

```
$ ruby next.rb
0
1
2
4
```

◆redo

ループ中で、同じ処理を再度実行します。

redo.rb

```
i = 0
5.times {
  i += 1
  if i == 3
    redo
  end
  puts i
}
```

```
$ ruby redo.rb
1
2
4
5
6
```

redoは使い方を誤ると無限ループに陥ることがありますので、利用する際は注意をしてください。次の例では、iの値が3になった段階で無限ループに入ってしまいます。

redo_loop.rb

```
5.times { |i|
  if i == 3
    redo
  end
  puts i
}
```

7.2 例外処理

例外処理を利用することで、正常な処理と例外（異常な処理）とを明確に区別して記述することができ、プログラムが見やすくなります。

7.2.1 begin~rescue~end

Rubyにおける例外処理の書式は次のとおりです。

例外処理

```
begin
  例外が発生するかもしれない処理（正常な処理）
rescue [=> 変数]
  例外が発生した時の処理
[ensure
  例外の発生にかかわらず必ず実行される処理]
end
```

次の例では、指定されたファイルが見つからないなど、4行目でファイルを開くことができなかったときに例外が発生します。

exception1.rb

```
begin
  print "ファイル名を入力してください。"
  filename = gets.chomp
  f = File.open(filename)
  print f.read
  f.close
rescue => ex
  p ex
  puts "エラーです。ファイル名を確認してください。¥n"
  retry
ensure
  puts "¥nプログラムを終了します。"
end
```

```
$ ruby exception1.rb
ファイル名を入力してください。hello
#<Errno::ENOENT: No such file or directory -
hello>
エラーです。ファイル名を確認してください。

ファイル名を入力してください。hello.rb
puts "hello, world."

プログラムを終了します。
```

7.2.2 例外の生成

raiseメソッドを使うと、明示的に例外を発生させることができます。

raiseの書式

```
raise [例外クラス, ]例外メッセージ  
raise 例外オブジェクト
```

exception2.rb

```
begin  
  raise "Exception!"  
rescue => ex  
  puts "Class    : #{ex.class.name}"  
  puts "Message : #{ex.message}"  
end
```

```
$ ruby exception2.rb  
Class    : RuntimeError  
Message : Exception!
```

7.3 メソッド

メソッドはオブジェクトに対する操作です。メソッドについては中級編で詳しく取り上げますので、ここでは基本的な利用方法を説明します。

7.3.1 メソッドの利用

Rubyでは基本的に、すべての操作はメソッド呼び出しによって行われます。メソッド呼び出しの書式は次のとおりです。

メソッドの書式

```
レシーバ.メソッド名(引数1, 引数2 ...)
```

メソッドの操作対象となるオブジェクトのことをレシーバといいます。レシーバが省略された場合は、self（現在のオブジェクト）のメソッドが呼び出されます。

レシーバなしで呼び出されるメソッドを関数的メソッドといいます。非オブジェクト指向言語での関数のような使い方ができます。たとえば、数学的な処理をまとめたMathモジュールで定義されたメソッドは関数的メソッドの代表例です。

sqrtメソッドの例

```
include Math  
sqrt(9)
```

7.3.2 メソッドの定義

メソッドは、def~endを使って定義することができます。

メソッド定義

```
def メソッド名 (引数リスト)
  処理
end
```

次の例では、引数を一つ取るhelloメソッドを定義しています。

methodtest1.rb

```
def hello(name)
  puts "Hello, #{name}"
end

hello("World.")
hello("Ruby.")
```

```
$ ruby methodtest1.rb
Hello, World.
Hello, Ruby.
```

第7章 テスト

問題 1

以下のコードを実行したとき、出力されるメッセージを記述してください。

```
i = 1
while i < 10
  puts i
  i += 2
end
```

問題 2

以下のコードを実行したとき、出力されるメッセージを記述してください。

```
for x in [1,2,3,4,5] do
  print "#{x} #{x+1} "
end
```

問題 3

以下のコードを実行したとき、出力されるメッセージを記述してください。

```
3.times {|i| puts i}
```

問題 4

以下のコードを実行したとき、出力されるメッセージを記述してください。

```
5.times { |i|
  if i == 2
    break
  end
  puts i
}
```

問題 5

以下のコードを実行したとき、出力されるメッセージを記述してください。

```
5.times { |i|
  if i == 3
    next
  end
  puts i
}
```


Ruby プログラミング

第8章

オブジェクト指向
プログラミング入門

8.1 オブジェクト指向の基本

現在では、オブジェクト指向によるプログラミングが標準となっています。オブジェクト指向の基本を確認しておきましょう。

8.1.1 オブジェクト指向とは

オブジェクトとは「もの」を意味する言葉ですが、オブジェクト指向言語では、オブジェクトは「データ」と「手続き」をまとめたものを意味します。

◆クラスとインスタンス

オブジェクトの設計図をクラス、そこから具体化されたものをインスタンスといいます。クラスは抽象的なものですが、インスタンスはメモリが割り当てられ、実際にデータを格納できる実体です。

◆継承

あるクラスの定義を引き継いで新しいクラスを定義することを「継承 (Inheritance)」といいます。継承元のクラスを (新しいクラスの) スーパークラス、新しく作られたクラスを (元のクラスの) サブクラスといいます。継承により、共通する部分をスーパークラスで定義し、異なる部分はサブクラスで定義するという差分プログラミングが可能になり、コードの再利用も促進されます。

◆カプセル化

あるオブジェクト内のメソッドを外部のオブジェクトから呼び出せないようにしたり、オブジェクト内のデータを外部から参照できないようにしたりすることをカプセル化といいます。カプセル化によって保守性を高めたり、データの不整合を起りにくくしたりすることができます。

◆ポリモルフィズム

同じ名前のメソッドでも、オブジェクトの種類によってそれぞれ異なった処理が行われることをポリモルフィズム (多態性) といいます。オブジェクトがどんなクラスから生成されているかを気にすることなく、直感的にプログラミングをすることができます。

8.2 Rubyにおけるオブジェクト指向

ここではRubyにおけるオブジェクト指向の基礎を取り上げます。詳細は中級編で学びます。

8.2.1 オブジェクト

Rubyでは、すべてがオブジェクトです。数値「3」、文字列「Ruby」もそれぞれ、数値オブジェクト、文字列オブジェクトであり、Fixnumクラス、Stringクラスのインスタンスです。

```
irb(main):001:0> 3.times { puts "Ruby" }  
Ruby  
Ruby  
Ruby  
=> 3
```

この例では、Fixnumクラスのtimesメソッドを利用しています。

```
irb(main):002:0> "Ruby"*3  
=> "RubyRubyRuby"
```

この例では、Stringクラスの「*」メソッドを利用しています。

8.2.2 クラスとインスタンス

クラスに対してnewメソッドを実行すると、インスタンスが作成できます。次の例では、Stringクラスのインスタンスを作成し、それぞれ変数r、pで参照できるようにしています。

instance1.rb

```
r = String.new("Ruby")
p = String.new("Python")
puts r
puts p
```

```
$ ruby instance1.rb
Ruby
Python
```

Rubyでは多くの組み込みクラスが用意されていますが、もちろん自分でクラスを作成することもできます。

class1.rb

```
class Myclass
  def hello
    puts "Hello!"
  end
end

a = Myclass.new
b = Myclass.new
a.hello
b.hello
```

```
$ ruby class1.rb
Hello!
Hello!
```

Rubyの組み込みクラスには次のようなものがあります。

[図8-1] Rubyの組み込みクラス

```
Object
+ Array
+ Binding
+ Continuation
+ Data
+ Exception
+ Dir
+ FalseClass
+ File::Stat
+ Hash
+ IO
  + File
+ MatchData
+ Method
+ Module
  + Class
+ Numeric
  + Integer
    + Bignum
    + Fixnum
  + Float
+ Proc
+ Range
+ Regexp
+ String
+ Struct
+ Symbol
+ Thread
+ ThreadGroup
+ Time
+ TrueClass
+ NilClass
```

8.2.3 継承

クラスを継承して、新しいクラスを作成できます。スーパークラスのメソッドはサブクラスでも利用できます。また、スーパークラスのメソッドを再定義することもできます。

class2.rb

```
# スーパークラスの定義
class MyClass
  def hello
    puts "Hello!"
  end
  def bye
    puts "Bye!"
  end
end

# サブクラスの定義
class MySub < MyClass
  def chao
    puts "Chao!"
  end
  def bye
    puts "Good-bye!"
  end
end

my_super = MyClass.new
my_sub = MySub.new

puts "Super Class:"
my_super.hello
my_super.bye
puts "Sub Class:"
my_sub.hello
my_sub.chao
my_sub.bye
```

```
$ ruby class2.rb
Super Class:
Hello!
Bye!
Sub Class:
Hello!
Chao!
Good-bye!
```

8.2.4 カプセル化

オブジェクト内の変数（インスタンス変数）は、外部から直接参照したり、変数の値を変更することはできません。

capsule.rb

```
class Capsule
  def initialize(var)
    @var = var
    puts "set var = #{@var}"
  end
end

c = Capsule.new(1)
p c.var      #=> undefined methodエラー
```

インスタンス変数にアクセスするためには、アクセスメソッドの定義が必要です（Ruby中級の第2章で学習します）。

initializeメソッドについては、P.106を参照のこと。

8.2.5 ポリモルフィズム

たとえば、sizeメソッドを例に取りましょう。

size.rb

```
ary = ["Ruby", "Python", "PHP", "Perl"]
str = "Ruby Python Perl PHP Java"
hash = { "R" => "Ruby", "P" => "Python", "J"
=> "Java" }
p ary.size
p str.size
p hash.size
```

```
$ ruby size.rb
4
25
3
```

これらは一見、同じsizeメソッドが呼び出されているように見えますが、実際には、配列 (Arrayクラス)、文字列 (Stringクラス)、ハッシュ (Hashクラス) それぞれのsizeメソッドが呼び出されています。Array#sizeは配列の要素数を、String#sizeは文字列の文字数 (バイト数) を、Hash#sizeはハッシュの要素数を返します。それぞれのsizeメソッドは実装も動作も異なりますが、いずれも「大きさを求める」という点では同じです。このように、共通の概念を持つメソッドに同じ名前を付けておけば、メソッドの処理対象となるクラスの違いを気にせず、直感的にプログラミングすることができます。これがポリモルフィズムのメリットです。

※ 「Arrayクラスのsizeメソッド」は「Array#size」のように表します。

8.2.6 クラスの定義と利用

クラスの定義をもう少し詳しく見ておきましょう。クラスは「class ~ end」の間に記述します。クラス名は必ず大文字で始まります。

最低限のクラス定義

```
class クラス名
  (変数やメソッドの定義)
end
```

クラスからインスタンスを生成するには、newメソッドを使います。

インスタンスの生成

```
変数名 = クラス名.new(引数)
```

以下は、クラスの作成とインスタンス生成の例です。クラス内には何も定義されていないので、それぞれのインスタンスを識別できるよう、object_idメソッドを使ってオブジェクトIDを表示させています。オブジェクトIDは、オブジェクトごとに割り当てられるユニークなIDです。

classtest.rb

```
class Sample
end

a = Sample.new
b = Sample.new
puts a.object_id
puts b.object_id
```

実行すると、2つのインスタンスに別々のオブジェクトIDが割り当てられているのが分かります。

```
$ ruby classtest.rb
-604164568
-604164588
```

class~endには、任意のインスタンスメソッドを定義することができます。initializeメソッドには、インスタンス生成時に実行される処理を記述しますが、必須ではありません。

クラスの定義

```
class クラス名
  def initialize(変数)
    初期化処理
  end
  def クラスメソッド名
    処理
  end
  ...
end
```

実際のプログラムを見てみましょう。この例では、Helloクラスを定義しています。Helloクラスはデータとして名前を持ち、その名前を使って挨拶を返すメソッドを持っています。なお、クラス名とファイル名は無関係です。また、1つのファイル内で複数のクラスを定義してもかまいません。

helloclass.rb

```
# クラスの定義
class Hello
  def initialize(name)
    @name = name
  end
  def hello
    puts "Hello, #{@name}."
  end
  def bye
    puts "Good-bye, #{@name}."
  end
end

# インスタンスの生成
fred = Hello.new("Fred")
john = Hello.new("John")
fred.hello
fred.bye
john.hello
john.bye
```

```
$ ruby helloclass.rb
Hello, Fred.
Good-bye, Fred.
Hello, John.
Good-bye, John.
```

@で始まる名前の変数はインスタンス変数です。それぞれのインスタンス内が有効な範囲です。また、挨拶を返すhelloメソッドとbyeメソッドを定義しています。

インスタンスの生成では、2つのインスタンスを生成し、それぞれ変数fredと変数johnで参照できるようにしています。「変数名.メソッド名」のようにすることでインスタンスメソッドを呼び出すことができます。

「=」は参照

Rubyでは、「=」は「右辺の値への参照を左辺に代入する」ということを表します。次の例を見てください。

refctest.rb

```
str1 = "Ruby"
str2 = str1

puts str1.object_id
puts str2.object_id

str1 = "NewObject"
puts "str1: #{str1}"
puts "str2: #{str2}"
puts str1.object_id
puts str2.object_id
```

最初、str1変数には文字列オブジェクト"Ruby"への参照が格納されています。それをstr2に代入すると、str2も同じ文字列オブジェクトを参照するようになります（オブジェクトIDが同じです）。次にstr1変数に別の文字列オブジェクトへの参照を代入すると、変数str1は別のオブジェクトを参照するようになることが分かります（オブジェクトIDが異なります）。

```
$ ruby refctest.rb
-604179188
-604179188
str1: NewObject
str2: Ruby
-604179218
-604179188
```

第8章 テスト

問題 1

クラスからインスタンスを作成するメソッドは何ですか？

問題 2

スーパークラスからサブクラスを作成することを何と言いますか？

問題 3

メソッドの処理対象となるクラスが別々であっても、同じ名前のメソッドで処理できるようにする、オブジェクト指向の概念を何と呼びますか？

Ruby プログラミング

第9章

数値

9.1 数値クラス

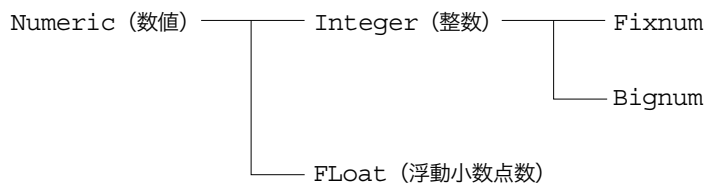
Rubyにはいわゆるデータ型がありません。その代わりに、数値クラスや文字列クラスなど、データ操作に特化したクラスが組み込まれています。

9.1.1 数値 (Numeric) クラスの概要

※Numericクラスは抽象クラスです。

Rubyでは数値もオブジェクトです。数値に関するクラスは、数値 (Numeric) クラスのサブクラスとして、Integer (整数) クラスとFloat (浮動小数点数) クラスがあります。Integerクラスは、さらにFixnumクラスとBignumクラスに分かれます。

[図9-1：数値 (Numeric) クラス]



数値オブジェクトを表すリテラルには、表のようなものがあります。

[表9-1：数値リテラル]

数値リテラル	説明
12345	10進数
012345	8進数
0x12345	16進数
0b01010101	2進数
12.345	浮動小数点数
1.2345e5	指数表記 (1.2345×10 ⁵)
1.2345e-5	指数表記 (1.2345×10 ⁻⁵)

なお、数値リテラル中のアンダースコア (_) は無視されますので、桁の大きな数値を表すときは、見やすいように挿入してもかまいません。

9.1.2 算術演算

基本的な算術演算の演算子は、他の言語と同様です。

[表9-2：算術演算子]

演算子	説明
+	加算
-	減算
*	乗算
/	除算
%	剰余
**	べき乗

FixnumクラスとBignumクラスの違いは、プログラマは意識する必要がありません。通常はFixnumクラスで十分ですが、Fixnumクラスに収まらなくなると、自動的にBignumクラスに変換されます。また、IntegerクラスとFloatクラスのオブジェクトを計算すると、計算結果はFloatクラスになります。

以下に、代表的な算術演算の例を示します。

```
irb(main):001:0> 5 + 3
=> 8
irb(main):002:0> 5 - 3
=> 2
irb(main):003:0> 5 * 3
=> 15
irb(main):004:0> 5 / 3
=> 1
irb(main):005:0> 5 % 3
=> 2
irb(main):006:0> 5 + 3.0
=> 8.0
irb(main):007:0> 5 * -3.0
=> -15.0
irb(main):008:0> 5 ** 3
=> 125
```

Mathモジュール

三角関数や平方根など、よく使われる数値演算メソッドは、Mathモジュールをインクルードすれば利用することができます。

```
irb(main):001:0> include Math
=> Object
irb(main):002:0> p sqrt(9)
3.0
=> nil
```

次のようにすれば、Mathモジュールをインクルードしていなくても利用することができます。

```
irb(main):001:0> p Math.sqrt(9)
=> 3.0
```

9.1.3 型変換

`to_f`メソッドと`to_i`メソッドを使うと、IntegerオブジェクトとFloatオブジェクトを相互に変換することができます。

```
irb(main):001:0> 1.to_f
=> 1.0
irb(main):002:0> 1.0.to_i
=> 1
```

文字列も変換することができます。

```
irb(main):003:0> "12345".to_i
=> 12345
irb(main):004:0> "12345".to_f
=> 12345.0
irb(main):005:0> "123" + 45
TypeError: can't convert Fixnum into String
    from (irb):5:in `+'
    from (irb):5
    from :0
irb(main):006:0> "123".to_i + 45
=> 168
```

`round`メソッドを使うと、小数点以下を四捨五入することができます。

```
irb(main):007:0> 12.345.round
=> 12
```

9.1.4 その他のメソッド

`integer?`メソッドを使うと、整数であるかどうかを確認できます。

```
irb(main):001:0> a = 10
=> 10
irb(main):002:0> a.integer?
=> true
irb(main):003:0> a = 10.0
=> 10.0
irb(main):004:0> a.integer?
=> false
```

※反対の働きをする`nonzero?`メソッドもあります。

`zero?`メソッドを使うと、0であるかどうかを確認できます。

```
irb(main):001:0> 0.zero?
=> true
irb(main):002:0> 1.zero?
=> false
```

9.1.5 ビット演算

Rubyでは、次表にあるビット演算子を使ってビット演算ができます。

[表9-3：ビット演算]

演算子	説明
~	反転
&	積
	和
^	排他的論理和
>>	右ビットシフト
<<	左ビットシフト

```
a = 0b11001100
~a          #=> 0b00110011
a & 0b01010101  #=> 0b01000100
a | 0b01010101  #=> 0b11011101
a >> 2          #=> 0b00110011
a << 3          #=> 0b01100000
```

第9章 テスト

問題 1

数値クラス (Numericクラス) のサブクラスを4つ挙げてください。

問題 2

「"123" + 45」のようにして、文字列オブジェクトに数値オブジェクトを足すことは可能ですか？

問題 3

桁の大きな数値を「1_234_567_890」のように表すことは可能ですか？

Ruby プログラミング

第10章

文字列、エンコーディング

10.1 文字列クラス

文字列 (String) クラスは任意の長さの文字列を扱います。

10.1.1 文字列の生成

シングルクォート (") もしくはダブルクォート (") で囲むと、文字列オブジェクトが生成されます。シングルクォートとダブルクォートの違いは、ダブルクォートでは「#{変数名}」を展開するような式展開ができるという点と、バックslash記号を使ったエスケープシーケンスが利用できるという点です。

シングルクォートとダブルクォート

```
str = "sample"
p "#{str}"           #=> "sample"
p '#{str}'          #=> "¥#{str}"
```

シングルクォートとダブルクォートを使う代わりに、「%q{}」「%Q{}」を使うこともできます。「%q{}」はシングルクォートと、「%Q{}」はダブルクォートと同じです。

quotetest.rb

```
str1 = %Q{任意の文字列}
str2 = %q{この中では'も'"も使えます。}
puts str1
puts str2
```

```
$ ruby quotetest.rb
任意の文字列
この中では'も'"も使えます。
```

ヒアドキュメントは、次の書式で利用できます。

ヒアドキュメント

```
~ << 終端文字
  任意の文字列
  終端文字
```

10.1.2 文字列の長さ

文字列の長さは、`length`メソッドもしくは`size`メソッドで調べることができます。

```
irb(main):001:0> "Hello,Ruby".length
=> 10
irb(main):002:0> "Hello,Ruby".size
=> 10
```

ただし、1.9系以前のRubyでは、文字列はバイト列として扱われるため、マルチバイト文字列では文字数と一致しません。

```
irb(main):003:0> "日本語の文字列".length
=> 21
```

その場合は、次のようにすることで日本語の文字数をカウントできます。

```
irb(main):005:0> "日本語の文字列".split(/u).size
=> 7
```

Ruby 1.9系では、日本語の文字列の長さも正常に扱われます。

※これはエンコーディングがUTF8の例です。ShiftJISの場合は、「(/u)」の代わりに「(/s)」とします。

10.1.3 文字列の分割と結合

splitメソッドを使って文字列を分割できます。分割された文字列は配列に格納されます。次の例では、文字列を「,」で区切り、それぞれを要素として配列aryに格納しています。

```
irb(main):001:0> str = "Ruby,Perl,Python,PHP,Java"
=> "Ruby,Perl,Python,PHP,Java"
irb(main):002:0> ary = str.split(/,/,)
=> ["Ruby", "Perl", "Python", "PHP", "Java"]
```

文字列同士は「+」を使って結合できます。この場合、変数strが参照する文字列オブジェクトは変更されません。

```
irb(main):001:0> str = "Hello, "
=> "Hello, "
irb(main):002:0> p str + "Ruby."
"Hello, Ruby."
```

「*」を使うと、文字列の内容を繰り返した文字列を生成します。

```
irb(main):001:0> "Ruby " * 3
=> "Ruby Ruby Ruby "
```

「<<」を使うと、元の文字列オブジェクトに新しく文字列が追加されます。

```
irb(main):003:0> p str << "Ruby."
"Hello, Ruby."
irb(main):004:0> p str
"Hello, Ruby."
```

concatメソッドも「<<」と同様です。

```
irb(main):001:0> str = "Hello, "
=> "Hello, "
irb(main):002:0> str.concat("Ruby.")
=> "Hello, Ruby."
irb(main):003:0> p str
"Hello, Ruby."
```

文字列を配列のようにとらえ、[]演算子を使って文字列の一部を取り出すことができます。1文字目のインデックスは0、-1は末尾を表します。たとえば「[3, 5]」は、インデックス3から5文字を意味します。「[3..5]」は、インデックス3からインデックス5を表します。

文字列の部分取りだし

```
str = "Programming Language Ruby"
p str[0, 7]          #=> "Program"
p str[0..7]         #=> "Programm"
p str[-4, 4]        #=> "Ruby"
p str[-4..-1]       #=> "Ruby"
```

10.1.4 文字列の比較

2つの文字列が同じであるかどうかを調べるには「==」「!=」を利用します。「==」は、同一の文字列である場合はtrueを、そうでない場合はfalseを返します。「!=」の場合はその逆です。

```
irb(main):001:0> "Ruby" == "Ruby"  
=> true  
irb(main):002:0> "Ruby" == "ruby"  
=> false  
irb(main):003:0> "Ruby" != "Ruby"  
=> false  
irb(main):004:0> "Ruby" != "ruby"  
=> true
```

「<」「>」記号を使うと、文字コードの順で比較されます。

```
irb(main):005:0> "a" < "b"  
=> true  
irb(main):006:0> "a" > "b"  
=> false
```

10.1.5 文字列の検索

`include?`メソッドを使って、文字列の中に特定の文字列が含まれているかどうかを検索することができます。含まれている場合は`true`を、含まれていない場合は`false`を返します。

```
irb(main):001:0> str = "Ruby, Python, Perl, PHP, Java"
=> "Ruby, Python, Perl, PHP, Java"
irb(main):002:0> str.include?("Ruby")
=> true
irb(main):003:0> str.include?("Lisp")
=> false
```

`index`メソッドもしくは`rindex`メソッドを使うと、検索している文字列が何文字目から始まるか、そのインデックスを調べることができます。探している文字列が見つからなかった場合は`nil`を返します。

```
irb(main):012:0> str.index("Python")
=> 6
irb(main):013:0> str.index("Lisp")
=> nil
irb(main):014:0> str.rindex("Java")
=> 25
```

※`index`メソッドは、文字列の左側から検索します。`rindex`メソッドを使うと右側から検索します。

10.1.6 主なメソッド

ここでは、文字列操作に関する代表的なメソッドを紹介します。

◆chomp、chomp!

chompメソッドもしくはchomp!メソッドは、文字列の末尾にある改行文字を削除するメソッドです。chompとchomp!の違いは、レシーバの文字列自体を変更するかどうかという点です。そのような、レシーバに変更を加えるメソッドのことを破壊的メソッドといいます。破壊的メソッドの多くは、メソッド名の末尾に「!」が付けられています。

chompメソッドの例

```
p "Ruby".chomp      #=> "Ruby"
p "Ruby\n".chomp    #=> "Ruby"
p "Ruby\r\n".chomp  #=> "Ruby"
```

◆chop、chop!

chomp/chomp!メソッドが行末の改行文字を削除するのに対し、chop/chop!メソッドは行末の1文字を何でも削除します。

chompメソッドの例

```
p "Ruby".chop       #=> "Rub"
p "Ruby\n".chop     #=> "Ruby"
p "Ruby\r\n".chop   #=> "Ruby"
```

◆downcase、upcase

downcaseメソッドは、アルファベットの大文字をすべて小文字に変換します。反対にupcaseメソッドは、アルファベットの小文字をすべて大文字に変換します。

downcase、upcaseメソッドの例

```
p "Ruby".downcase   #=> "ruby"
p "Ruby".upcase     #=> "RUBY"
```

◆reverse

reverseメソッドは、文字列の並びを逆順にします。

reverseメソッドの例

```
p "Ruby".reverse    #=> "ybuR"
```

※ただし行末が「\r\n」であれば、その2文字を削除します。

◆delete

deleteメソッドは、文字列の中から任意の文字を削除します。

deleteメソッドの例

```
p "Ruby".delete("R")      #=> "uby"  
p "abcdefg".delete("c-e") #=> "abfg"
```

10.2 エンコーディング

エンコーディングとはデータの符号化のことで、文字と文字コードとの対応付けなどを意味します。Rubyではさまざまなエンコーディングを扱えますが、Rubyのバージョンによって対応が異なります。

10.2.1 バージョンによる日本語処理の違い

Ruby 1.8では、文字列は単なるバイト列として扱われます。しかしRuby 1.9のStringオブジェクトはエンコーディング情報を知っていて、sizeメソッドやlengthメソッドも適切に扱われます。

Ruby 1.8の場合

```
p "日本語の文字列".size #=> 21
```

Ruby 1.9の場合

```
p "日本語の文字列".size #=> 7
```

10.2.2 エンコーディングの指定

Ruby 1.8では、プログラム実行時にエンコーディングを指定するには、`-K` オプションを使います。たとえば、ソースコードがShift-JISであれば、次のように指定します。

```
$ ruby -Ks source.rb
```

同様に、`-Ke`はEUC、`-Ku`はUTF-8を意味します。

グローバル変数`$KCODE`には、現在のエンコーディングが格納されています。プログラム内でエンコーディングを取得したり、指定したりすることができます。

encoding1.rb

```
str = "日本語の文字列"  
$KCODE = 'UTF-8'  
p str  
$KCODE = 'SJIS'  
p str  
$KCODE = 'EUC'  
p str
```

UTF-8の環境で実行すると、以下のようになります。

```
$ ruby encoding1.rb  
"日本語の文字列"  
"觸245觸254語の文藹255"  
"日觸234ャ隱縞觸226¥207蟄¥227¥227"
```

`$KCODE`はプログラム内のどこからでも変更ができるので、その点には注意が必要です。

10.2.3 エンコーディングの変換 (1.8)

nkfモジュールを使うと、文字コードを変換して出力することができます。次のプログラムでは、Shift-JISの文字列をUTF-8に変換して出力します。

encoding2.rb

```
require "nkf"

str = "Shift-JISで記述された文字列"
p NKF.nkf("-S -w -xm0", str)
```

```
$ ruby encoding2.rb
Shift-JISで記述された文字列
```

また、iconvモジュールを使っても同様の処理ができます。

encoding3.rb

```
require "iconv"

str = "Shift-JISで記述された文字列"
p Iconv.conv("UTF-8", "Shift-JIS", str)
```

10.2.4 エンコーディングの変換 (1.9)

Ruby 1.9では、`encode`メソッドを使って文字列のエンコーディングを変換することができます。また、`encoding`メソッドを使って文字列のエンコーディングを確認することができます。

encoding

```
str = "Shift-JISで記述された文字列"  
p str.encoding  
p str.encode("UTF-8")
```

```
#<Encoding:Shift-JIS>  
Shift-JISで記述された文字列
```

※ShebangでRubyインタプリタのパスが指定されている場合は、その次の行に記述します。

10.2.5 ソースコードのエンコーディング

どのエンコーディングがソースコードで使われているかは、マジックコメントという仕組みを使って認識されます。ソースコードの1行目に、次のように記述してください。

マジックコメント

```
# -*- coding: utf-8 -*-
```

第10章 テスト

問題 1

"I love Ruby." と同じ文字列オブジェクトを生成するものを選択してください。

- A. %Q{I love Ruby.}
- B. %q{I love Ruby.}
- C. #Q{I love Ruby.}
- D. #q{I love Ruby.}

問題 2

以下のコードを実行したとき、出力されるメッセージを記述してください。

```
puts "A" * 3
```

問題 3

以下のコードを実行したとき、出力されるメッセージを記述してください。

```
str = "ABC" ; puts str << "DE"
```

問題 4

以下のコードを実行したとき、出力されるメッセージを記述してください。

```
str = "Open Source"  
puts str[1,3]  
puts str[-3..-1]
```

問題 5

文字列の末尾にある改行コードを削除する（レシーバの文字列自体は変更しない）メソッドは何ですか？

Ruby プログラミング

第11章

配列

11.1 配列

さまざまなオブジェクトを順に並べたものが配列です。配列に格納されたデータは、インデックス（添字）を使って指定できます。

11.1.1 配列の概要

※正確には、オブジェクトへの参照を並べたものであり、オブジェクトそのものを格納しているわけではありません。

Rubyでは、1つの配列にさまざまな種類のオブジェクトを格納することができます。つまり、1つめの要素には文字列、2つめには数値、のように、異なるクラスのオブジェクトを格納できます。また、配列のサイズは固定されていませんので、配列作成後にいくつでも要素を追加できます。

配列は、[]内に要素を「,」で区切って列挙することで生成できます。また、配列を配列の中に入れることもできます。

配列の生成1

```
num = [1, "2", 3.14, 4, 5]
lang = ["Ruby", "Python", "PHP", "Perl"]
ary = ["a1", ["b1", "b2"], "c1", [1, 2, 3]]
```

要素が文字列の場合は、%wを使った指定が便利です。[]の代わりに任意の文字を指定することができ、要素の区切りには空白を使うことができます。

配列の生成2

```
lang = %w(Ruby Python PHP Perl)
```

Arrayクラスのインスタンスを作ると、空の配列を作ることができます。

配列の生成3

```
ary = Array.new
```

11.1.2 インデックス

配列の要素はインデックス（添字）を使って指定します。インデックスは0から始まります。「[インデックス, 要素数]」とすると、取り出す要素の数を指定できます。負数は末尾からのインデックスで、末尾の要素は-1、その一つ前は-2となります。要素の存在しないインデックスを指定するとnilを返します。

arytest1.rb

```
ary = %w(Ruby Python Perl PHP Java)
p ary[1]
p ary[1,2]
p ary[-3,2]
p ary[5]
```

```
$ ruby arytest1.rb
"Python"
["Python", "Perl"]
["Perl", "PHP"]
nil
```

「[..]」や「[...]」のような範囲式を使って指定することもできます。「[..]」と「[...]」の違いは、末端を含むかどうかです。

arytest2.rb

```
ary = %w(Ruby Python Perl PHP Java)
p ary[1..3]
p ary[1...3]
p ary[-3..-1]
```

```
$ ruby arytest2.rb
["Python", "Perl", "PHP"]
["Python", "Perl"]
["Perl", "PHP", "Java"]
```

11.1.3 配列の操作

配列に要素を追加するには、「<<」を使います。

arytest3.rb

```
ary = %w(Ruby Python Perl PHP Java)
ary << "Lisp"
p ary
```

```
$ ruby arytest3.rb
["Ruby", "Python", "Perl", "PHP", "Java",
"Lisp"]
```

インデックスを指定して要素を変更することもできます。

```
irb(main):001:0> ary = %w(Ruby Python Perl
PHP Java)
=> ["Ruby", "Python", "Perl", "PHP", "Java"]
irb(main):002:0> ary[4] = "Lisp"
=> "Lisp"
irb(main):003:0> p ary
["Ruby", "Python", "Perl", "PHP", "Lisp"]
=> nil
```

複数の要素をまとめて変更することもできます。

```
irb(main):004:0> ary[3,2] = ["C++", "C#"]
=> ["C++", "C#"]
irb(main):005:0> p ary
["Ruby", "Python", "Perl", "C++", "C#"]
=> nil
```

11.1.4 配列の主なメソッド

配列の主なメソッドを紹介します。

◆要素数の確認

配列の要素数を確認するには、`size`メソッドもしくは`length`メソッドを使います。

arytest4.rb

```
ary = %w(Ruby Python Perl PHP Java)
p ary.size           #=> 5
p ary.length        #=> 5
```

◆要素の取り出し

配列の要素を一つずつ取り出すには`each`メソッドを使います。

arytest5.rb

```
ary = %w(Ruby Python Perl PHP Java)
ary.each { |lang|
  p lang
}
```

```
$ ruby arytest5.rb
"Ruby"
"Python"
"Perl"
"PHP"
"Java"
```

`each_with_index`メソッドを使うと、インデックスも合わせて取り出すことができます。

arytest6.rb

```
ary = %w(Ruby Python Perl PHP Java)
ary.each_with_index { |lang, i|
  puts "ary[#{i}] : #{lang}"
}
```

```
$ ruby arytest6.rb
ary[0] : Ruby
ary[1] : Python
ary[2] : Perl
ary[3] : PHP
ary[4] : Java
```

◆要素の削除

`delete_at`メソッドを使うと、指定したインデックスの要素を削除します。

```
irb(main):001:0> ary = %w(Ruby Python Perl
PHP Java)
=> ["Ruby", "Python", "Perl", "PHP", "Java"]
irb(main):002:0> ary.delete_at(4)
=> "Java"
irb(main):003:0> p ary
["Ruby", "Python", "Perl", "PHP"]
=> nil
```

`slice`メソッドを使うと、指定した範囲の要素を削除します。

```
irb(main):004:0> ary.slice!(1,2)
=> ["Python", "Perl"]
```

`clear`メソッドを使うと、すべての要素を削除して空の配列にします。

```
irb(main):004:0> ary.clear
=> []
```

◆要素のソート

要素をソートするには、`sort`メソッドを使います。

`arytest7.rb`

```
ary1 = %w(Ruby Python Perl PHP Java)
ary2 = [24, 54, 103, 7, 13, 48]
p ary1.sort
p ary2.sort
```

```
$ ruby arytest7.rb
["Java", "PHP", "Perl", "Python", "Ruby"]
[7, 13, 24, 48, 54, 103]
```

文字列を数値に変換してソートするには、次のように「<=>」演算子を使います。

arytest8.rb

```
ary = ["24", "54", "103", "7", "13", "48"]
p ary.sort
p ary.sort { |x, y| x.to_i <=> y.to_i }
```

```
$ ruby arytest8.rb
["103", "13", "24", "48", "54", "7"]
["7", "13", "24", "48", "54", "103"]
```

◆要素の置換

配列の要素にブロック内の処理を適用し、その結果から新しい配列を作るメソッドがcollectメソッド/mapメソッドです。collect!/map!メソッドは破壊的メソッドです。

arytest9.rb

```
ary = [1, 2, 3, 4, 5]
p ary.collect { |i| i * 10 }
p ary.map { |i| i / 5 }
```

```
$ ruby arytest9.rb
[10, 20, 30, 40, 50]
[2, 4, 6, 8, 10]
```

第11章 テスト

問題 1

配列を作成する方法として適切なものをすべて選択してください。

- A. `ary = ["A", "B", 3]`
- B. `ary = %w(A B 3)`
- C. `ary = Array.new()`
- D. `ary = ("A", "B", 3)`
- E. `@ary = ("A", "B", 3)`

問題 2

以下のコードを実行したとき、出力されるメッセージを記述してください。

```
ary = %w(A B C D E)
puts ary[2..3]
puts ary[2...3]
puts ary[1..-1]
```

問題 3

配列に関する記述として適切なものに○をつけてください。

- () 配列に要素を追加するには「<<」メソッドが利用できる
- () 配列の要素を1つずつ取り出すにはeachメソッドが利用できる
- () 配列からインデックスと要素を1つずつ取り出すにはeach_indexメソッドが利用できる
- () 配列をソートするにはsortメソッドが利用できる

Ruby プログラミング

第12章

ハッシュ

12.1 ハッシュ

ハッシュは、JavaやC++のマップ、Perlの連想配列に相当するオブジェクトです。

12.1.1 ハッシュの概要

ハッシュは配列と似ていますが、インデックスの代わりに任意のオブジェクトをキーとして使える点が異なります。

ハッシュは、「{ キー => 値 }」という書式で生成できます。

ハッシュの生成1

```
lang1 = { "R" => "Ruby", "P" => "Python", "J"
=> "Java" }
```

※newメソッドに引数を指定すると、値のデフォルト値として扱われます。

また、Hashクラスのインスタンスを作ると、空のハッシュが生成されます。

ハッシュの生成2

```
lang2 = Hash.new
lang2['R'] = "Ruby"
lang2['P'] = "Python"
lang2['J'] = "Java"
```

次のようにしてもハッシュを生成することができます。

ハッシュの生成3

```
os = Hash["Linux", "OSS", "Windows",
"Microsoft", "MacOS", "Apple"]
```

12.1.2 値の取り出し

ハッシュの値を取り出すには、「ハッシュ名[キー]」で参照します。キーに対応する値がない場合はnilを返します。

hashtest1.rb

```
lang = { "R" => "Ruby", "P" => "Python", "J"
=> "Java" }

p lang
p lang["R"]
p lang["P"]
p lang["J"]
```

```
$ ruby hashtest1.rb
{"P"=>"Python", "R"=>"Ruby", "J"=>"Java"}
"Ruby"
"Python"
"Java"
```

eachメソッドを使うと、すべてのキーと値のペアを取り出すことができます。

hashtest2.rb

```
lang = { "R" => "Ruby", "P" => "Python", "J"
=> "Java" }

lang.each { |key, value|
  puts "#{key} : #{value}"
}
```

```
$ ruby hashtest2.rb
P : Python
R : Ruby
J : Java
```

キーのみを取り出すには`each_key`メソッドを、値のみを取り出すには`each_value`メソッドを使います。

hashtest3.rb

```
lang = { "R" => "Ruby", "P" => "Python", "J"
=> "Java" }

lang.each_key { |key|
  puts key
}

lang.each_value { |value|
  puts value
}
```

```
$ ruby hashtest3.rb
P
R
J
Python
Ruby
Java
```

`keys`メソッドは、キーから配列を生成します。また、`values`メソッドは、値から配列を生成します。`to_a`メソッドを使うと、ハッシュを配列に変換します。

hashtest4.rb

```
lang = { "R" => "Ruby", "P" => "Python", "J"
=> "Java" }

p lang.keys
p lang.values
p lang.to_a
```

```
$ ruby hashtest4.rb
["P", "R", "J"]
["Python", "Ruby", "Java"]
[["P", "Python"], ["R", "Ruby"], ["J",
"Java"]]
```

なお、ハッシュでは要素の並びを保持しませんので、取り出される順序は保証されません。

12.1.3 ハッシュのメソッド

ハッシュの主なメソッドを紹介します。

◆要素数の確認

ハッシュの要素数を確認するには、`size`メソッドもしくは`length`メソッドを使います。

hashtest5.rb

```
lang = { "R" => "Ruby", "P" => "Python", "J"
=> "Java" }

p lang.size           #=> 3
p lang.length        #=> 3
```

◆要素の存在確認

指定したキーが存在するかどうかを確認するには、`has_key?`メソッドもしくは`key?`メソッドを使います。

hashtest6.rb

```
lang = { "R" => "Ruby", "P" => "Python", "J"
=> "Java" }

p lang.has_key?("R")      #=> true
p lang.has_key?("A")      #=> false
```

指定した値が存在するかどうかを確認するには、`has_value?`メソッドもしくは`value?`メソッドを使います。

hashtest7.rb

```
lang = { "R" => "Ruby", "P" => "Python", "J"
=> "Java" }

p lang.has_value?("Ruby")  #=> true
p lang.has_value?("AWK")  #=> false
```

◆要素の登録

ハッシュに要素を登録するには、`store`メソッドを使います。

storeメソッドの例

```
lang.store("A", "AWK")
```

◆要素の削除

deleteメソッドを使うと、キーを指定して要素を削除します。また、clearメソッドを使うと、すべての要素を削除します。

hashtest8.rb

```
lang = { "R" => "Ruby", "P" => "Python", "J"
=> "Java" }

lang.delete("J")
p lang
lang.clear
p lang
```

```
$ ruby hashtest8.rb
{"P"=>"Python", "R"=>"Ruby"}
{}
```

◆ハッシュのマージ

mergeメソッドを使うと、ハッシュをマージすることができます。

hashtest9.rb

```
hash1 = {"a" => 97, "b" => 98}
hash2 = {"c" => 99, "d" => 100}

p hash1.merge(hash2)
p hash2.merge(hash1)
```

```
$ ruby hashtest9.rb
{"a"=>97, "b"=>98, "c"=>99, "d"=>100}
{"a"=>97, "b"=>98, "c"=>99, "d"=>100}
```

第12章 テスト

問題 1

ハッシュを作成する方法として適切なものをすべて選択してください。

- A. `str = ["A"->1, "B"->2, "C"->3]`
- B. `str = ["A"=>1, "B"=>2, "C"=>3]`
- C. `str = {"A"->1, "B"->2, "C"->3}`
- D. `str = {"A"=>1, "B"=>2, "C"=>3}`
- E. `str = Hash.new`

問題 2

以下のコードは、ハッシュ`str`のすべてのキーと値のペアを出力します。下線部に当てはまるメソッド名を記述してください。

```
str._____ { |key, value|  
  puts "#{key} : #{value}"  
}
```

問題 3

ハッシュに関する記述として適切なものに○をつけてください。

- () ハッシュでは、ハッシュに登録した順に要素が取り出せる
- () 指定したキーが存在するかどうかを確認するには、`has_key?`メソッドが利用できる
- () ハッシュに要素を追加するには`insert`メソッドを使用する
- () `clear`メソッドを使うとすべての要素を削除できる

索引

記号

\$KCODE	137
%w	144
"	128
""	128
*	130
+	130
<<	130
<<	146
<=>	149
__END__	77
__FILE__	75
__LINE__	75
=begin	77
=end	77

アルファベット

A

ActiveScriptRub	8
ARGF	74
ARGV	74

B

begin	101
Bignumクラス	120
break	99

C

case式	90
chomp!メソッド	134
chomp/chomp!メソッド	134
chompメソッド	134
chop/chop!メソッド	134
class	115
clearメソッド	148・156
collectメソッド	149
concatメソッド	130

D

DATA	74
------	----

de	104
delete_atメソッド	148
deleteメソッド	135・156
do	94
downcaseメソッド	134

E

each_keyメソッド	154
each_valueメソッド	154
each_with_indexメソッド	147
eachメソッド	98・147・153
else	86
elsif	86
encodeメソッド	139
encodingメソッド	139
end	86・94・101・104
ensure	101
ENV	74

F

FALSE	75
Fixnumクラス	109・120
Floatクラス	121
for式	96

G

gets	86
------	----

H

has_key?メソッド	155
has_value?メソッド	155
Hashクラス	152

I

iconvモジュール	138
if	86
include?メソッド	133
indexメソッド	133
initializeメソッド	116
integer?メソッド	124
Integerクラス	120
irb	78
IronRuby	68

J

JRuby	68
-------	----

K
 key?メソッド155
 keysメソッド154

L
 lengthメソッド129・147・155
 loopメソッド98
 MacRuby68

M
 mapメソッド149
 Mathモジュール103・122
 mergeメソッド156
 MRI68

N
 newメソッド110・115
 next99
 nil75
 nkfモジュール138
 nonzero?メソッド124
 Numericクラス120

O
 One-Click Ruby Installer for Windows ... 8

P
 Perl5・12
 PHP6・30
 printメソッド90
 puts70
 Python7・48
 pメソッド76

R
 raiseメソッド102
 redo100
 rescue101
 reverseメソッド134
 rindexメソッド133
 roundメソッド123
 Ruby8
 RUBY_PLATFORM74
 RUBY_RELEASE_DATE74
 RUBY_VERSION74

S
 self75

sizeメソッド114
 sizeメソッド129・147・155
 sliceメソッド148
 sortメソッド148
 splitメソッド130
 STDERR74
 STDIN74
 STDOUT74
 storeメソッド155
 Stringクラス109
 (String) クラス128

T
 then86
 timesメソッド97
 to_aメソッド154
 to_fメソッド123
 to_i86
 to_iメソッド123
 TRUE75

U
 unless式88
 until式95
 upcaseメソッド134

V
 value?メソッド155
 valueメソッド154

W
 when90
 while式94

Z
 zero?メソッド124

カ ナ

ア
 イテレータ97
 インスタンス108
 インスタンス変数72・113
 インデックス76・144・145
 エンコーディング136
 演算子83

オブジェクト	108・109
オブジェクトID	115

カ

型変換	123
カプセル化	108・113
ガベージコレクション	67
関数的メソッド	103
キー	152
擬似変数	75
組み込みクラス	111
組み込み定数	74
クラス	108
クラス変数	73
クラス名	115
グローバル変数	73
継承	108・112
コメント	77

サ

式	82
辞書型	57
シングルクォート	128
数値クラス	120
スカラー	17

タ

タプル	56
ダブルクォート	128
定数	74

ハ

配列	76・144
ハッシュ	76・152
ヒアドキュメント	128
ビット演算子	125
変数	72
ポリモルフィズム	108・114

マ

マイナーバージョン	68
マジックコメント	140
メーリングリスト	69
メジャーバージョン	68
メソッド	71・103
文字列クラス	128

ラ

ライセンス	66
リテラル	120
リファレンスマニュアル	69
レシーバ	103
ローカル変数	72

Rubyプログラミング入門

Ver 1.0.0

2009年7月1日 初版 第1刷 発行

著者 株式会社 リナックスアカデミー
監修 株式会社 万葉
発行 株式会社 リナックスアカデミー
〒160-0023
東京都新宿区西新宿7-4-3 升本ビル
TEL: 03-3365-2072 FAX: 03-3365-2076
URL: <http://www.linuxacademy.ne.jp/>
<http://www.linuxacademy.ne.jp/biz/>

※本書は、「クリエイティブ・コモンズ・ライセンス 表示 2.1 日本」により、株式会社リナックスアカデミーから利用許諾されています。
詳しい利用許諾条項は、<http://creativecommons.org/licenses/by/2.1/jp/legalcode> をご覧ください。