応用プログラム言語II / 演習II #12

Rubyでお手軽スクリプトプログラミング

情報システム学科
坂本政祐
sakapon@sit.ac.jp

はじめに

- 皆さんが今まで学んできた言語は，2年必修のC，この講義のC++などをはじめ，比較的「固い」言語について学んできたと思います。
  - C:　静的型付け，コンパイラ型，手続き型
  - C++:　静的型付け，コンパイラ型，オブジェクト指向

はじめに

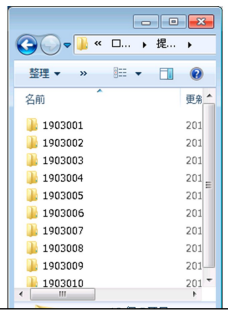
- 一方で，さっと書いてさっと実行することができる言語も最近は良く使われます。
  - Perl
  - Python
  - PHP
  - Ruby
- これらはスクリプト言語とかLightweight Language（LL）とかとも呼ばれます。

はじめに

- 今日はその中のRubyについて学んでみましょう。
- こういった言語が存在することを知ってもらえば，「ちょっとしたツールはLLでささっと書こう」という動機になると思います。
- ちょっとしたツール:
  - CやC++では面倒，でも電卓ではちょっとやりづらい，というような計算
  - Webを自動的にクロールする
  - 大量に何かをしたいのだけどひとつひとつやるのは面倒すぎる

はじめに

- 例えば私は，各講義の「提出用フォルダ」を作成するときに，Ruby のスクリプトで一気に作成しています。



- これらを，右クリックしながら「新規作成→フォルダ」なんて1個1個やっていたら死ぬほど面倒くさいし，間違う可能性も高いですが…

はじめに

- こういうときはまず学籍番号を行ごとに列挙してあるファイルを作っておいて（LiveCampusから容易に作れます），
- Rubyのこういうスクリプトに食わせるだけです。

make-teishutsu-folders.rb

```
File.open(ARGV[0]){|f|
  while gakuseki = f.gets
    Dir::mkdir gakuseki.chomp
  end
}
```

- こういうのをちょっと作ってすぐ使う，というのがRubyの良いところなのです。

※実はこのスクリプトはさらに短く書けますが，意味の想像がつく範囲で普通に書きました。

はじめに

- まあこれぐらいだったら以下のようにC言語で書いても大差ないといえば大差ないのですが…。

make-teishutsu-folders.c

```
#include <stdio.h>
#include <sys/stat.h>

int main(void)
{
    char gakuseki[8];

    while(scanf("%s", gakuseki) != EOF){
        mkdir(gakuseki, S_IREAD|S_IWRITE);
    }
}
```


はじめに

- ただ，Cだと:
  - コンパイルしなきゃいけないし，
  - そもそもこのソースはPOSIXという規格準拠で書いたが，VCはPOSIX準拠じゃないのでまた別の仕様の違うmkdir()を使わなきゃいけないし（要するに，ファイルシステムの操作はOSごとに違いがあり，Cはそれの統一操作方法を提供してくれていない），
  - これがCSVデータだったりすると途端にかなり面倒なことになりそう。
- こういうのを抽象化してあるのもスクリプト言語の嬉しいところ。

Rubyの概要

Rubyの特徴

- Rubyの特徴:
  - インタプリタ型言語。コンパイルは不要。
  - 動的型付け。変数宣言不要。
  - 強力なオブジェクト指向型。すべてがオブジェクト。ただしOOP知らなくてもなんとなく書ける。
    - 以降では，「○○クラスのインスタンス」というのをRubyでの慣習に従い「○○オブジェクト」と表記する。
  - 組み込みライブラリ，標準添付ライブラリ，フリーライブラリともに非常に豊富。

Rubyの特徴

- Rubyの特徴(続き):
  - 日本人のまつもとゆきひろが作った。
    - ので，言語としては珍しく初めから日本語(漢字コード)をサポート。
    - 初めは主に日本で普及した。
  - Ruby on Rails というWebフレームワークが出て以降，世界でも有名になり，Web界でもかなりの地位を占めるようになった。
    - twitter のシステムは以前は Ruby (on Rails)だった。

Rubyの特徴

- Rubyの特徴(続き):
  - ガベージコレクション(GC)というメモリ管理の仕組みがあるので，確保したものを解放することを気にしなくて良い。
  - 実行速度はあまり速くない。
  - GUIは苦手。
  - 実行速度を求めるのではなく，いかに素早く(そして楽しく)コードを書けるかの方に主眼がある。

Rubyの特徴

- Rubyの特徴(続き):
  - Visual Studio のような IDE は(無いわけではないが)あまり使わない。テキストエディタでさっと書く。
  - この実習室では Cygwin 上で使えるようになっているので，今日の実習は Cygwin 上でやる。
  - 情報源
    - 公式サイト: http://www.ruby-lang.org/ja/
    - リファレンスマニュアル(Ruby 1.8.7 用): http://doc.ruby-lang.org/ja/1.8.7/doc/index.html

RubyでHello, World.

- Ruby 初めの一歩
  - Hello, World.

hello.rb
```
# Hello, World Script
puts "Hello, World."
```

  - 上記スクリプトをテキストエディタ(メモ帳，秀丸エディタなど)で打つ。
  - hello.rb というファイル名で C:¥cygwin¥home¥free に保存。Ruby のスクリプトはこのように拡張子は常に .rb。
  - Cygwin を起動。
  - ruby hello.rb と打って Hello, World と表示されれば成功。

実行例
```
$ ruby hello.rb
Hello, World.
```


RubyでHello, World.

- Hello, World. からわかること:
  - 特別なエントリポイント(Cのmain()関数のような)は特にない。ソースコードの上に書いた行から順に実行される。
  - 「#」～行末までがコメントになる。
  - 行末に「;」とかは要らない。
  - puts というのが画面に表示する(改行込み)メソッド。
    - メソッドというのは関数と思って下さい。
    - Rubyのメソッド呼び出しは ( ) を省略できる。より丁寧に puts("Hello, World.") と書いてもよい。
  - インタプリタの起動は「ruby」コマンド。これのコマンドライン引数としてソースコードのファイル名を与える。

puts と p

- 補足
  - 表示するメソッドは puts と p と print と printf というのがある。
  - p の方は，どんなオブジェクトかをわかりやすく表示してくれるので，デバッグしているときとか，配列の中身を全部どばっと見たいときなどに便利。

puts-and-p.rb
```
puts "1"
p "1"
puts [1, 2, 3]
p [1, 2, 3]
```

実行例
```
$ ruby puts-and-p.rb
1
"1"
1
2
3
[1, 2, 3]
```

p の方は " " を補ってくれるので String オブジェクトであることがわかりやすい。

p の方は [ ] を補ってくれるので Array オブジェクトであることがわかりやすい。

Rubyの文法

17

Rubyの変数

applied-progII #12

- 変数

var.rb
```
var = 356
puts var
var = "foo"
puts var
```

  - 宣言しなくてもいきなり代入できる。
  - 変数側には型はない。どんな型のものも代入できる(だから動的型付け)。
  - 変数名(ローカル変数の場合):
    - 小文字で始める。
    - 予約語やメソッド名とかぶらなければ何でも。
  - 同じ変数に違う型のものを代入しなおすこともできてしまう(もちろん可読性の意味からは好ましくは無い)。

18

Rubyのオブジェクト

applied-progII #12

- Rubyではありとあらゆるものがオブジェクト。
- "foo" は Stringオブジェクト(=Stringクラスのインスタンス)。文字列ともいう。
- var = "foo" というのは,
  - 1. 右辺で "foo" という名無しのStringオブジェクトを生成し,
  - 2. 名無しのままだと行方不明になってしまうので, 左辺で var という名札をそのオブジェクトに貼りつけた。
- Rubyの変数というのは入れ物ではなくて, あくまでオブジェクトが主役で, それにくっつける名札にしか過ぎない。
- ちなみに 356 は Fixnumオブジェクト。

19

Arrayオブジェクト

applied-progII #12

- Arrayオブジェクト(配列)

array.rb
```
ary = [100, 101, 102]

puts ary[0] #=> 100
puts ary[1] #=> 101
puts ary[2] #=> 102
```

  - [100, 101, 102] は Arrayオブジェクト。
  - それに ary という名札をつけた。
  - それ以降は ary[0], ary[1] のように各要素にアクセスできる(ゼロオリジン)。
  - #=> というのは, 何か特別な命令ではなくやはりただのコメントだが, 習慣的に「この行を評価した結果こういうことが起こる」というのを #=> で表す。

20

Arrayオブジェクト

applied-progII #12

- Arrayオブジェクトの各要素は:
  - Stringオブジェクトでも OK
  - その他の任意のクラスのオブジェクトでも OK

array2.rb
```
ary2 = ["foo", "bar", "baz"]

puts ary2[0] #=> "foo"
puts ary2[1] #=> "bar"
puts ary2[2] #=> "baz"
```

21

Hashオブジェクト

applied-progII #12

- Hashオブジェクト(連想配列)

hash.rb
```
nenrei = {"sakamoto"=>20, "inoue"=>17, "maeda"=>25}

puts nenrei["sakamoto"] #=> 20
nenrei["ido"] = 45      # あとから追加
```

  - C++で言うところのstd::map。
  - {"sakamoto"=>20, "inoue"=>17, "maeda"=>25} はHashオブジェクト。
  - Hashオブジェクトの各要素は, キー=>値 で表す。
  - また, 後から ハッシュ名["キー"] = 値 で読み書きもできる。

22

インスタンスメソッド

applied-progII #12

- 各クラスにはインスタンスメソッドが用意されている。
  - C++のときもありましたよね。
  - しかも欲しいメソッドは大抵揃っている。
  - インスタンスメソッドは「.」演算子で, そのメソッドを適用させたいインスタンス(レシーバ)にくっつけて使う。
- たとえば String クラスのインスタンスメソッドは以下にすべて載っている。
  - http://doc.ruby-lang.org/ja/1.8.7/class/String.html

23

インスタンスメソッド

applied-progII #12

- Stringクラスのインスタンスメソッド抜粋
  - chomp　末尾に改行コードがあれば削除して返す
  - split　指定したパターンで分割してArrayオブジェクトとして返す
  - sub　文字列中の指定したパターンを別の文字列に変換した結果を返す
  - length　長さを返す
  - upcase　アルファベット小文字を大文字に変換した結果を返す

24

インスタンスメソッド

applied-progII #12

■例1

string-method.rb
```
str1 = "foo"

puts str1.upcase #=> "FOO"
puts str1.length #=> 3
```

■例2

string-method2.rb
```
str1 = "foo,bar,baz"

ary = str1.split(/,/) # str1を「,」で分割。結果はArrayオブジェクト
puts ary[0] #=> "foo"
puts ary[1] #=> "bar"
```

25

名無しオブジェクトに対するインスタンスメソッド

applied-progII #12

■なお，インスタンスメソッドは，変数によって名前をつけたものしかレシーバになれないわけではなく，名前をつけていない Stringオブジェクトを直接レシーバにしてもよい。

■例3

string-method3.rb
```
puts "foo".upcase #=> "FOO"
puts "foobar".length #=> 6
```

26

Arrayクラスのインスタンスメソッド

applied-progII #12

■Arrayクラスのインスタンスメソッド抜粋

- ■join　指定した文字列を間に挟んで連結したStringオブジェクトを返す
- ■reverse　順番をひっくり返して新たなArrayオブジェクトとして返す
- ■size　要素数を返す
- ■sort　ソートして新たなArrayオブジェクトとして返す
- ■uniq　重複した要素を取り除いて新たなArrayオブジェクトとして返す

※これらはレシーバを破壊しない。

27

Arrayクラスのインスタンスメソッド

applied-progII #12

■Arrayクラスのインスタンスメソッド抜粋

- ■map　レシーバの各要素に対してブロックを評価した結果をすべて含む新たなArrayオブジェクトとして返す（後程詳述）
- ■select　レシーバの各要素に対して，ブロックが真を返したもののみで構成される新たなArrayオブジェクトを返す

※これらはレシーバを破壊しない。

28

Arrayクラスのインスタンスメソッド

applied-progII #12

■例

array-method.rb
```
ary = [3, 5, 2, 7, 5]

p ary.reverse   #=> [5, 7, 2, 5, 3]
p ary.size      #=> 5
p ary.sort      #=> [2, 3, 5, 5, 7]
p ary.uniq      #=> [3, 5, 2, 7]
p ary.join("/") #=> "3/5/2/7/5"
```

29

Arrayクラスのインスタンスメソッド

applied-progII #12

■続・Arrayクラスのインスタンスメソッド抜粋

■以下は見た目は演算子のようにも見えるが，全部そういう名前のメソッド。

- ■+　レシーバのArrayオブジェクトと引数で指定したArrayオブジェクトとを連結して新たなArrayオブジェクトとして返す
- ■-　レシーバのArrayオブジェクトから，引数で指定したArrayオブジェクトの要素を取り除いたものを新たなArrayオブジェクトとして返す
- ■shift　先頭要素を取り除いて返す。レシーバを破壊する。

30

Arrayクラスのインスタンスメソッド

applied-progII #12

■例2

array-method2.rb
```
ary1 = [1, 2]
ary2 = [8, 9]

p ary1 + ary2 #=> [1, 2, 8, 9]
p ary1        #=> [1, 2]（特に壊れない）
p ary2        #=> [8, 9]（こちらも）
```

■例3

array-method3.rb
```
ary1 = [1, 2, 1, 3, 1, 4, 1, 5]
ary2 = [2, 3, 4, 5]

p ary1 - ary2 #=> [1, 1, 1, 1]
p ary1        #=> [1, 2, 1, 3, 1, 4, 1, 5]（特に壊れない）
p ary2        #=> [2, 3, 4, 5]（こちらも）
```

31

Arrayクラスのインスタンスメソッド

applied-progII #12

■例4

array-method4.rb
```
ary1 = [100, 101, 102]

p ary1.shift #=> 100
p ary1       #=> [101, 102]
```

32

if

- if

```if.rb
janken = rand(3) # 0 or 1 or 2

if janken == 0
  puts "Guu"
elsif janken == 1
  puts "Choki"
else
  puts "Paa"
end
```

  - if ～ elsif ～ else ～ end （elsif や else は必要に応じて）
  - 条件式の前後に ( ) は不要。

ループ

- for は，あるけどCとかの for と文法が違うし，そもそもあまり使わない。
- なぜなら，例えばArrayクラスには，中身を列挙するためのインスタンスメソッド each があるので，それを使った方がわかりやすいし，後述のメソッドチェーンなどにも応用が利くから。

クラス名#メソッド名

- 以降の説明で
  - クラス名#メソッド名

  という書き方が良く出てくるが，これは「あるクラスに用意されているインスタンスメソッド」という意味。
  - 例1: Array#split　Arrayクラスのsplitメソッド
  - 例2: String#join　Stringクラスのjoinメソッド

Array#each

- Array#each（Arrayクラスのインスタンスメソッド each）

```array-each.rb
ary = [100, 101, 102]

ary.each do |num|
  puts num
end
```

実行例

```
$ ruby array-each.rb
100
101
102
```

  - ary から1要素ずつ取り出して，順に num に入れていく。
  - each メソッドを使えばループカウンタをわざわざ作る必要もない。

Array#each

- each は一見制御構造のようにも見えるが，あくまでただのメソッド呼び出し。
- do～end までをブロックと呼び，このブロックという大きなひと固まりが each メソッドの実引数になっているということ。
- ブロックの中には任意のコードを書ける。複数行でも良い。
- | | に挟まれた部分をブロックパラメータと言って，ここには自分で決めた好きな変数名を書く。
- 以上の言葉で言い直すと，Array#each は:
  - レシーバから1要素ずつ取り出してブロックパラメータの変数に代入し，その都度ブロックの中身を実行するというメソッド。

Array#each

```array-each.rb
ary = [100, 101, 102]

ary.each do |num|
  puts num
end
```

- ary というインスタンスに対して each メソッドを呼び出すときに，　の部分を引数として渡しているということ。
- 日本語的にかつ擬人化的に解釈するなら:
  - aryさん，(ary)
  - あなたの各要素に対してお願いがある。(.each)
  - 「各要素に対して仮に num という一時的な名前をつけていきながらそれを表示するという仕事」をしてね。(do |num| ; puts num; end)

Rangeオブジェクト

- Rangeオブジェクト（範囲）

```range.rb
(1..3).each do |num|
  puts num
end
```

実行例

```
$ ruby range.rb
1
2
3
```

  - 開始値..終了値 として生成できる。
  - Rangeオブジェクトにもインスタンスメソッド each がある。
  - インスタンスメソッド to_a でArrayオブジェクトに変換もできる。

```range2.rb
p (1..3).to_a  #=> [1, 2, 3]
```


その他のクラス

- その他，標準で用意されているクラスとしては以下などがある。
  - Dirクラス　ディレクトリを表す
  - Fileクラス　ファイルを表す
  - Timeクラス　時刻を表す
  - Dateクラス　日付を表す
  - Symbolクラス　言葉を扱う

メソッドチェーン

メソッドチェーン

- 例えば以下のようなRubyコードがあったとする。

array-sort-uniq.rb
```
ary = [3, 5, 2, 7, 5]

ary2 = ary.sort   #=> [2, 3, 5, 5, 7]
ary3 = ary2.uniq  #=> [2, 3, 5, 7]
```

- Arrayオブジェクトに対して，
  - ソートしてから
  - 重複した要素を取り除く

  という2つのことをやるプログラム。
- 上記のように1段階ずつやると，それぞれ新たに生じたArrayオブジェクトに対してそれぞれ ary2 などの名前をつけなければいけない。

メソッドチェーン

- ところが，sort メソッドが返すのはやはり Array オブジェクトであって，Array オブジェクトに対してはそれをレシーバとして uniq メソッドを呼び出せるのだから，これらのメソッドを連続して続けてしまっても良い。

array-sort-uniq2.rb
```
ary = [3, 5, 2, 7, 5]

ary2 = ary.sort.uniq  #=> [2, 3, 5, 7]
```

- このように，メソッドを次々に連結していって，やりたいことを1行にまとめて実現することをメソッドチェーンという。
  - メソッドが鎖（チェーン）のようにつながっているので。

メソッドチェーン

- さて，Array#mapメソッドは，ブロックを引数に取り，レシーバの各要素に対してブロックを評価した結果をすべて含む新たなArrayオブジェクトとして返す。
  - mapという名前通り，1対1の写像をその場で作れる。
  - 超絶便利。
- なので do ～ end 形式で以下のように書ける。

array-map.rb
```
ary = [3, 5, 2, 7, 5]

ary2 = ary.map do |num|
  num * 2
end
p ary2 #=> [6, 10, 4, 14, 10]
```


メソッドチェーン

- ブロックの作り方は実は二通りあり，
  - do |x| ～ end　　単体でループさせたい場合に良く使う
  - {|x| ～ }　　メソッドチェーンで良く使う

  という二通りの書き方ができるので，以下のようにも書ける。

array-map2.rb
```
ary = [3, 5, 2, 7, 5]

ary2 = ary.map {|num|
  num * 2
}
p ary2 #=> [6, 10, 4, 14, 10]
```


メソッドチェーン

- すると，この map も含めてメソッドチェーンも作れる。
- これは非常に強力で，かなりいろいろなことができるようになる。

array-map-sort-uniq.rb
```
ary = [3, 5, 2, 7, 5]

p ary.map{|num| num * 2}.sort.uniq  #=> [4, 6, 10, 14]
```


どんなメソッドを繋げられるかはレシーバ次第

- ここまでのメソッドチェーンではたまたまArrayオブジェクト→Arrayオブジェクト→Arrayオブジェクト… となるような例ばかりだったが，途中が何オブジェクトでも構わない。
- 例えば，Array#join と String#split は正反対の動きをするので，以下のような（無意味だけど）たくさん連鎖したメソッドチェーンを作ることもできる。

join-split-join-split.rb
```
ary = [3, 5, 2, 7, 5]

p ary.join("@").split(/@/).join("@").split(/@/).join("@").split(/@/)
#=> [3, 5, 2, 7, 5]
```


メソッドチェーンの実例

- 以下にメソッドチェーンの実際的な例をいくつか。
  - 学籍番号を1903001～1903100までArrayオブジェクトに入れたい。各要素はStringオブジェクトにしたい。

地道版
```
gakuseki = ["1903001", "1903002", "1903003", (中略), "1903100"]
```

method-chain1.rb
```
gakuseki = (1..100).to_a.map{|n| sprintf("1903%03d", n)}
```

  - Range#to_a でArrayオブジェクト化し，
  - Array#map で，その各要素を n という名前で1つずつ取り出し，sprintfメソッドの引数として与え，その返却値を集めたArrayオブジェクトをさらに作り，
  - それを gakuseki に代入。

メソッドチェーンの実例

applied-progII #12

- 文章に含まれる単語の登場回数をカウントし，辞書順に表示する。

method-chain2.rb

```
sentence = "Foo bar baz foo bar bar."
count = Hash.new(0)

sentence.split(/[¥., ]/).map{|word| word.downcase}.(次の行に継続)
each{|word| count[word] += 1}

count.keys.sort.each do |word|
  puts "#{word}: #{count[word]}"
end
```

49

メソッドチェーンの実例

applied-progII #12

- method-chain2.rbの解釈（前半）
  - キーが単語で値が登場回数であるようなHashオブジェクトcountを用意する。newするときの引数で，新規キーのデフォルトの値を設定することができる。
  - 「.」「,」「 」で分割する（.split(/[¥.,]/)）。結果はArrayオブジェクト。
  - 「This」と「this」などを同一視するため，単語をすべて小文字にする（.map{|word| word.downcase}）。結果はArrayオブジェクト。
  - あるwordのcountの値を1増やす（.each{|word| count[word] += 1}）。ちなみにRubyには++演算子はありません。

50

メソッドチェーンの実例

applied-progII #12

- method-chain2.rbの解釈（後半）
  - countのキー達を配列化する（count.keys）。結果はArrayオブジェクト。
  - それをソートする（.sort）。結果はArrayオブジェクト。
  - そのそれぞれのキーに対して（.each do |word|～end），
  - あるフォーマットで画面に出力（puts "..."）。"" の中で #{文}と書くと，その文を評価した値がそこに埋め込めます。

51

まとめ

applied-progII #12

- スクリプト言語に慣れ親しむために，その代表格であるRubyを学んだ。
- Rubyはオブジェクト指向言語。しかし手続き型的にささっと書くことも可能だしそれで良い場合も多いので，今回は割とそういう書き方について。
- 次回は本格的OOPLとしてのRuby，次々回はどんな応用でも短く直感的に書けるRuby，という視点で学んでいく予定です。

52

今日の課題

applied-progII #12

- 課題1: スライド14以降のすべてのソースコードを写経し，実行して下さい。
  - ただし，一部のソースコードは，本質部分だけを見せるために「代入だけして結果は表示していない」ような構造になっています。これだと実行しても何もおきないので，意味のある実行結果になるように結果を何らかの方法で表示してください。
- 課題2: 何でも良いので，何かひとつメソッドチェーンを作ってみてください。

53

今週の落穂拾い

applied-progII #12

- 今回の演習はCygwin上のRubyでやりましたが，わざわざCygwinを入れなくても，Windows用Rubyバイナリもあります。ので，もっと気軽に試せます。
- スクリプト言語は冒頭で紹介したように他にもいくつかあり，それぞれ特徴があります。
  - Perl: Rubyよりも以前からあった。スクリプト言語の代表格。しかし古いだけあって元々オブジェクト指向の機能などのモダンな機能はない。文法も，$ や @ などの記号を多用することもあってちょっととっつきづらい。

54

今週の落穂拾い

applied-progII #12

- Python: こちらはモダンなスクリプト言語の代表格。海外ではRubyよりも人気。オブジェクト指向。インデントによって構造を表現するのが好き嫌いのわかれるところ。
- それぞれいろいろ試してみると良いとおもいます。

55