テキスト処理 第4回 (2006-05-16)

田中哲
産業技術総合研究所
情報技術研究部門
akr@isc.senshu-u.ac.jp
http://staff.aist.go.jp/tanaka-akira/textprocess/

今日の内容

- 前回のレポートの解説
- egrep.rb で -i オプションをサポートする
- 大文字小文字を区別しない正規表現
- ライブラリの使いかた
- optparse: コマンドラインオプション解析ライブラリ
- test/unit: ユニットテストライブラリ

前回のegrep.rb

```
pattern = ARGV.shift
regexp = Regexp.compile(pattern)
ARGF.each {|line|
  print line if regexp =~ line
}
```

今回のegrep.rb

```
require 'optparse'
patopts = 0
opt = OptionParser.new
opt.def_option("-i", "case insensitive") {
  patopts |= Regexp::IGNORECASE
}
opt.parse!
pattern = ARGV.shift
regexp = Regexp.compile(pattern, patopts)
ARGF.each {|line|
  print line if regexp =~ line
}
```

-i オプションの使用

- 使用例

```
% ruby egrep.rb -i ruby words
Ruby
Ruby's
ruby
ruby's
rubying
```

新しい機能

- 大文字小文字を区別しない正規表現
- Regexp.compile(pat, opts)
- require
- optparse ライブラリ
- ブロックをとっておいて後で実行する

大文字小文字を区別しない正規表現

- /abc/i というように i を付けると大文字小文字を区別しない (case-insensitive の i)

```
p /abcdef/i =~ "ABCDEF"        #=> 0
p /abcdef/i =~ “AbCdEf”        #=> 0
p /ABCDEF/i =~ “abcdef”        #=> 0
p /ABCDEF/i =~ “aBCdeF”        #=> 0
p /abCDeF/i =~ “AbCdEf”        #=> 0

p /abcdef/ =~ “ABCDEF”         #=> nil
```

Regexp.compile(pat, opts)

- 正規表現オブジェクトを作る
- 正規表現の意味を変えるオプションを指定可能
  - 大文字小文字を区別しない: Regexp::IGNORECASE
  - . を改行にもマッチさせる: Regexp::MULTILINE
  - 空白を無視する: Regexp::EXTENDED
  - オプションはビット論理和して opts に指定

Regexp::IGNORECACE

- Regexp.compile で生成する正規表現に /abc/i の i と同じ機能をつける

```
r = Regexp.compile(“abc”, Regexp::IGNORECASE)
p r                 #=> /abc/i
p r =~ “ABC”    #=> 0
```

他のオプション

- Regexp::MULTILINE は . を改行にマッチさせる
  リテラルは /.../m
  p /a.b/m =~ “a\nb” #=> 0
  p /a.b/ =~ “a\nb”　#=> nil
- Regexp::EXTENDED は空白(含改行)を無視する
  リテラルは /.../x
  p /a　　b/x =~ “ab”　　#=> 0
  p /a　　b/ =~ “ab”　　#=> nil
  複数行に渡る長い正規表現をインデントしてわかりやすく書くときに使う

egrep.rbで大文字小文字無視

```
require 'optparse'
patopts = 0
opt = OptionParser.new
opt.def_option("-i", "case insensitive") {
  patopts |= Regexp::IGNORECASE
}
opt.parse!
pattern = ARGV.shift
regexp = Regexp.compile(pattern, patopts)
ARGF.each {|line|
  print line if regexp =~ line
}
```

-i があったら
ビット論理和

require

- ライブラリを使用する (ロードする)
  例
  require 'pp'　　# pp ライブラリを使用
  pp ENV
- コマンドラインから -r オプションで require できる
  例
  ruby -rpp -e 'pp ENV'
- ロードした後のライブラリの使いかたはライブラリによる
  (pp ライブラリでは pp メソッドが使えるようになる)

egrep.rb の require

```
require 'optparse'
patopts = 0
opt = OptionParser.new
opt.def_option("-i", "case insensitive") {
  patopts |= Regexp::IGNORECASE
}
opt.parse!
pattern = ARGV.shift
regexp = Regexp.compile(pattern, patopts)
ARGF.each {|line|
  print line if regexp =~ line
}
```

optparse

- コマンドラインオプション解析ライブラリ
- オプションには標準的な形式がある
  - cmd -x　　ショートオプション
  - cmd -xARG　引数付ショートオプション(区切り無し)
  - cmd -x ARG　引数付ショートオプション(空白区切り)
  - cmd -xyz　　ショートオプションの連鎖
  - cmd --opt　　ロングオプション
  - cmd --opt=ARG　引数付ロングオプション (= 区切り)
  - cmd --opt ARG　引数付ロングオプション(空白区切り)
  - cmd -- ARGS　　-- から後はオプションでない
  - オプション引数は省略可能な場合もある
- 毎回実装しなくていいようにライブラリがある
- ライブラリを使うと簡単で、標準的な動作になる

optparseの使いかた

- OptionParserオブジェクト生成、ブロック登録、解析
- 使いかた

```
require 'optparse'
opts = OptionParser.new    # オブジェクト生成
opts.def_option(“-i”, “case insensitive”) { # ブロック登録
  # -i が指定されたときの処理
}
opts.parse!  # ARGV を実際に解析する
             # 登録されたブロックを呼び出し、
             # ARGV からオプションを削る
```

オブジェクト生成

- クラス.new
- new というクラスメソッド
- opts = OptionParser.new
  p opts #=> #<OptionParser ... なんかたくさん ...>

egrep.rb の OptionParser.new

```
require 'optparse'
patopts = 0
opt = OptionParser.new
opt.def_option("-i", "case insensitive") {
  patopts |= Regexp::IGNORECASE
}
opt.parse!
pattern = ARGV.shift
regexp = Regexp.compile(pattern, patopts)
ARGF.each {|line|
  print line if regexp =~ line
}
```

ブロック登録

- OptionParser#def_option
- opts.def_option(“-x”, “description”) {
  オプションが指定されたときの処理
  }
- このブロックは、登録されるだけで実行されない
- 後で実行されるかもしれない
- ブロック内では、オプションが指定されたときの処理を行う
- description は --help オプションで表示される
- OptionParser#def_option の引数の指定は他にもいろいろある

egrep.rb の def_option

```
require 'optparse'
patopts = 0
opt = OptionParser.new
opt.def_option("-i", "case insensitive") {
  patopts |= Regexp::IGNORECASE
}
opt.parse!
pattern = ARGV.shift
regexp = Regexp.compile(pattern, patopts)
ARGF.each {|line|
  print line if regexp =~ line
}
```

OptionParser#def_option

- opts.def_option(“-a”, “--text”, “all file as text file”)
  複数同時指定
- opts.def_option(“-A NUM”, “--after-context NUM”)
  必須オプション引数
- opts.def_option(“--color [WHEN]”)
  省略可能オプション引数
- opts.def_option(“-C NUM”, Integer)
  オプション引数が整数であることを指定
- 他にも多彩な指定ができる

ブロックをとっておく

- ブロックを Proc オブジェクトとして受け取れる
- Proc オブジェクトは call メソッドで呼び出せる

```
def m(arg, &block)                    def m(arg)
  block.call(arg)       ほぼ等価        yield arg
end                                   end
```

- 受け取った Proc オブジェクトはメソッドが終わった後でも呼び出せる

```
def m(&block)                 b = m { p 1 }
  block                       b.call
end
```

OptionParser#parse!

- 実際に ARGV を解析する
- ARGV の内容に応じて登録されたブロックを呼び出す
- ARGV から処理済みのオプションを削除する
- 注: Ruby では xxx! とか xxx? というメソッド名を使える

egrep.rb の parse!

```
require 'optparse'
patopts = 0
opt = OptionParser.new
opt.def_option("-i", "case insensitive") {
  patopts |= Regexp::IGNORECASE
}
opt.parse!
pattern = ARGV.shift
regexp = Regexp.compile(pattern, patopts)
ARGF.each {|line|
  print line if regexp =~ line
}
```

ARGVの変化

- ruby egrep.rb -i ruby words
- [“-i”, “ruby”, “words”]
- opt.parse!
- [“ruby”, “words”]
- ARGV.shift
- [“words”]
- ARGF.each
- []

test/unit

- ユニットテストフレームワーク
- プログラムをテストする
- p で表示して目視検査するかわりに自動でテスト
- あくまでもテストなので正しさを完全に検証できるわけではない
- 用途
  - テストを書くことによってプログラムの仕様を考える
  - ライブラリを修正したときにバグが増えてるか調べる
  - いろんな環境でユーザにテストしてもらう
  - レポートに「正しく書けたらこのテストが通る」と使う

map を test/unit でテストする

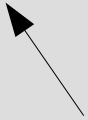
```
require 'test/unit'                                  # require
def map(ary) ... end
class TestMap < Test::Unit::TestCase
  def test_twice                                     # テストの定義1
    assert_equal([], map([]) {|v| v*2})              # 表明1
    assert_equal([2,4,6], map([1,2,3]) {|v| v*2})    # 表明2
  end
  def test_non_destructive                           # テストの定義2
    a = [1,2,3]
    map(a) {|v| }
    assert_equal([1,2,3], a)                         # 表明3
  end
end
```

テストの実行: 成功した場合

```
% ruby map-test.rb
Loaded suite map-test
Started
..
Finished in 0.000371 seconds.

2 tests, 3 assertions, 0 failures, 0 errors
```

テストの数



表明の数

テストの実行: 失敗した場合

```
% ruby map-test.rb
Loaded suite map-test
Started
.F
Finished in 0.00695 seconds.

  1) Failure:
test_twice(TestMap) [map-test.rb:10]:
<[2, 4, 6]> expected but was
<[nil, nil, nil]>.

2 tests, 3 assertions, 1 failures, 0 errors
```

[2,4,6] になるはずが
[nil,nil,nil]になった

テストの書き方

- require 'test/unit' と先頭に書く
- Test::Unit::TestCase を継承したクラスを作る

```
class TestXXX < Test::Unit::TestCase
  ...
end
```

- test_ で始まるメソッドをそのクラスに定義する

```
  def test_xxx
    ...
  end
```

- assert_equal などを使って答え合わせする

表明: **assert_equal** など

- assert_equal(答, 式)
- assert_not_equal(答でないもの, 式)
- assert_nil(式)
- assert_match(パターン, 文字列)
- assert_no_match(パターン, 文字列)
- assert_same(答，式)
- assert_raise(例外クラス) { コード }
- etc.

map を test/unit でテストする

```
require 'test/unit'                                          # require
def map(ary) ... end    # 直接定義する代わりに require してもよい
class TestMap < Test::Unit::TestCase
  def test_twice                                             # テストの定義1
    assert_equal([], map([]) {|v| v*2})    # 空配列が空配列になるか?
    assert_equal([2,4,6], map([1,2,3]) {|v| v*2})      # [2,4,6] になるか?
  end
  def test_non_destructive                                   # テストの定義2
    a = [1,2,3]
    map(a) {|v| v }
    assert_equal([1,2,3], a)      # map しても配列が破壊されないか?
  end
end
```

レポート

- egrep.rb で -v オプションをサポートし、動いている様子を示し、解説せよ
- -v の指定によりマッチ「しない」行を表示する
- -i も同時にサポートすること
- IT's class で 2006-05-23 16:20 まで
- プレインテキスト

まとめ

- egrep もどきで -i をサポートした
- 大文字小文字を区別しない正規表現
- ライブラリの使いかた
  - optparse
  - test/unit