Duelling banjos part II - Ruby

January 26, 2012

Control Structures, iteration, ranges again

```
# The Ruby if-statement begins with "if" and a conditional (no do)
# and ends with "end"

>> if nil
>>   print "nil is true"
>> end
=> nil

>> if false
>>   print "false is true"
>> end
=> nil

# "nil" and "false" evaluates to false in Ruby, everything else
# (including 0) eveluates to true

>> if 0
>>   print "0 is true"
>> end
0 is true=> nil          # Only nil and false are false

# Logical operators ("and" and "or") can be used to combine
# conditionals

>> if 0 and [] and {} and ''
>>   puts "0 and [] and {} and '' are true"
>> end
(irb):15: warning: string literal in condition
0 and [] and {} and '' are true
=> nil

# If-statements can be combined using "elsif". The "else" clause
# will be run if none of the conditionals above evaluated to
# true.
```

```
>> name = "Yukihiro"
=> "Yukihiro"
>> if name == "Yukihiro"
>>   puts "Ruby"
>> elsif name == "Guido"
>>   puts "Python"
>> else
?>   puts "None of the above"
>> end
Ruby
=> nil

# "or" and "and" will be evaluated as boolean true/false but
# will also always return one of the objects in the expression

>> names = { "Yukihiro" => "Ruby",
?>           "Guido" => "Python",
?>          }
=> {"Yukihiro"=>"Ruby", "Guido"=>"Python"}
>> puts names[name] or "None of the above"
Ruby
=> "None of the above"

>> puts (names[name] or "None of the above")
Ruby
=> nil

>> result = (false or 'Beatrice' or false)
=> "Beatrice"
>> if result
>>   puts result
>> end
Beatrice
=> nil

>>  result = ('Beatrice' and [] and 'Isak')
=> "Isak"
>> if result
>>   puts result
>> end
Erik
=> nil

# Negation of boolean values is done using "not".

>> not false
=> true

>> (not true) == false
```

```
=> true

# The true and false values are represented by objects, and there
# is only one true object and one false object

>> (not true) === false
=> true
>> (not true).object_id == false.object_id
=> true

# The Ruby while-loop starts with "while" and a conditional and
# ends with end

>> i = 0
=> 0
>> while i < 10
>>   puts i
>>   i += 1
>> end
0
1
.
.
8
9
=> nil

# Several other ways of looping exists, e.g. the "each" iterator
# in the Range class.

>> lst = 1..10
=> 1..10
>> lst.each {|i| puts i}
1
2
.
.
8
9
10
=> 1..10

# Below we see an example of operations that we could do with
# lists. Using the "zip" method from the Array class we create a
# new list containing lists of the elements from the original
# lists that were found on the same index.

>> cs = ['a','b','c']
=> ["a", "b", "c"]
>> ns = [1,2,3]
```

```
=> [1, 2, 3]
>> zip = cs.zip(ns)
=> [["a", 1], ["b", 2], ["c", 3]]

# The list of lists is looped over using the "each" operator and
# the elements of the lists inside the list are printed out.

>> zip.each do |ch, no|
?>   puts "#{ch} #{no}"
>> end
a 1
b 2
c 3
=> [["a", 1], ["b", 2], ["c", 3]]

# The "collect" iterator will loop over the list collecting all
# elements from the original list for which the conditional in the
# code block evaluates to true. In this case, the code block
# contains no conditional and all elements will be collected.

>> chars = zip.collect {|c, n| c}
=> ["A", "b", "c"]

>> nums = zip.collect {|c, n| n}
=> [1, 2, 3]
```

A first simple program

```
# Let's say we have a file named "words.txt" with the following
# words stored in it:

#    Conan
#    is
#    awesome
#    says
#    Jonathan
#    and
#    Andreas

# The following code returns a list of words starting with a
# (lowercase) vowel.

>> matches = []
=> []
>> vowels = "aoueiy".chars.to_a
=> ["a", "o", "u", "e", "i", "y"]
```

```
>> f = open('words.txt')
=> #<File:words.txt>
>> f.each do |line|
?>   matches << line.strip if vowels.include? line.strip.chars.first
>> end
=> #<File:words.txt>
>> matches
=> ["is", "awesome", "and"]
```

Putting it into a function

```
# Ruby methods start with "def", a method name and an optional
# list of arguments. The result of evaluating the last line of
# the method will be the return value of the method.

>> def filter_words(file_name, filter_as_str)
>>   matches = []
>>   filter = filter_as_str.chars.to_a
>>   f = open(file_name)
>>   f.each_line do |line|
?>     matches << line.strip if filter.include? line.strip.chars.first
>>   end
>>   f.close
>>   matches
>> end
=> nil

>> puts filter_words('words.txt', 'aeiouy')
is
awesome
=> nil
```

Classes & Inheritance

```
# A Ruby class-definition starts with "class" and a class name. If
# no superclass is specified, the superclass will be Object. Below
# we create an empty class named Person as a subclass to Object.
# Objects of the Person class are created by sending the "new"
# message to the object that represents the Person class at runtime.

>> class Person
>> end
=> nil
```

```
>> Person.new
=> #<Person:0x1012124e0>

# Below, we open the Person class again and add functionality to
# it. The method called "initialize" will be run automatically
# upon object creation (when calling "new").

>> class Person
>>   def initialize(name)
>>     @name = name
>>   end
>>   def say_hi
>>     puts "Hi! My name is #{@name}"
>>   end
>> end
=> nil

# Since the "initialize" method takes one argument, we can no
# longer create objects without providing an argument.

>> p = Person.new
ArgumentError: wrong number of arguments (0 for 1)
from (irb):157:in `initialize'
from (irb):157:in `new'
from (irb):157
from :0

p = Person.new("Yukihiro")
=> #<Person:0x1011eed38 @name="Yukihiro">

# The object representing the Person class (and other classes) can
# be treated as any object. We can assign it to a variable and we
# can send messages to it.

>> my_class = Person
=> Person
>> my_class
=> Person
>> my_class.new("Guido")
=> #<Person:0x1011e29e8 @name="Guido">

# Methods are defined _only_ by their name. No overloading allowed.

>> class Person
>>   def a_method
>>     puts 1
>>   end
>>   def a_method(arg)
>>     puts 2
>>   end
```

```
>> end
=> nil
>> p.a_method
ArgumentError: wrong number of arguments (0 for 1)
from (irb):172:in `a_method'
from (irb):172
from :0

# A method that contains "yield" will expect a code block to
# be executed.

>> class Person
>>   def give_me_something_to_do
>>     puts "Start of method"
>>     yield
>>     puts "End of method"
>>   end
>> end
=> nil

# The call to the method defined above will look similar to the
# calls we've already made to various iterators.

>> p.give_me_something_to_do {p "Something"}
Start of method
"Something"
End of method
=> nil

# The code block can use locally defined variables.

>> a_string = "Yukihiro"
=> "Yukihiro"
>> p.give_me_something_to_do {p a_string}
Start of method
"Yukihiro"
End of method
=> nil

# A subclass can be defined by including "<" and the name of the
# superclass in the class definition.

>> class SubPerson < Person
>>   def a_method(arg) # Overriding
>>     puts "In SubPerson: #{arg}"
>>   end
>> end
=> nil
>> sp = SubPerson.new("Yukihiro")
=> #<SubPerson:0x100618fb8 @name="Yukihiro">
```

```
>> sp.a_method("Ruby")
In SubPerson: Ruby
=> nil
```

Back to example program

```
# wordlist.rb

class WordList
  def initialize(file_name, filter_str)
    @file_name, @filter_str = file_name, filter_str
    @matches = Array.new
  end

  def filter(file_name = nil, filter_str = nil)
    filter = (filter_str or @filter_str).chars.to_a
    file = (file_name or @file_name)
    open(file) do |file|      # opens file and closes after reading it
      file.each_line do |line|
        matches << line.strip if filter.include? line.strip.chars.first
      end
    end
  end

  def matches
    @matches
  end
end

require 'wordlist'
=> true
>> wl = WordList.new('words.txt','aoueiy')
=> #<WordList:0x1005c43f0 @matches=[], @filter_str="aoueiy", @file_name="words.txt">
>> wl.filter
=> #<File:words.txt (closed)>
>> wl.matches
=> ["is", "awesome"]
```