ruby: a lightning tour

Daniel Jackson

For now, all rights reserved. Daniel Jackson, 2013
Ruby’s origins

Ruby, 1996
› Yukihiro “Matz” Matsumoto

Ruby on Rails, 2004
› web framework
› extracted from Basecamp app
› by David Hansson

Yukihiro “Matz” Matsumoto
## Similarities to Other Languages

<table>
<thead>
<tr>
<th></th>
<th>Like Java &amp; JavaScript</th>
<th>Like Java</th>
<th>Like JavaScript</th>
<th>Like Smalltalk</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strongly typed</td>
<td>Class-based</td>
<td>Dynamic</td>
<td></td>
<td>All values are objects</td>
</tr>
<tr>
<td>Garbage collected</td>
<td>Access control</td>
<td>Arrays with holes</td>
<td></td>
<td>Iterators</td>
</tr>
<tr>
<td>Exceptions</td>
<td>Namespaces</td>
<td>Has first class functions</td>
<td></td>
<td>???</td>
</tr>
<tr>
<td>Few implicit conversions</td>
<td>Lexical closures</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
online resources

Ruby communal website (http://ruby-lang.org)
› Programming Ruby book
› API docs & language guides
› comparisons with Python & Java

for newcomers (http://docs.ruby-doc.com/docs/Newcomers)
› tips for advanced programmers

interactive courses
› http://codecademy.com
› http://codeschool.com

online Ruby shell
› http://tryruby.org
lexical structure & syntax

comments
› start with #

whitespace sensitivity
› end of line indicates end of expression/statement
› semicolons separate expressions on single line
› spaces matters in various places in nasty ways

capitalization
› class names must be capitalized
› capitalized variable is a constant; reassignment gives warning

syntax
› def/end instead of curlies
› parens not needed for calls

```
a = [2,4,5]
>> [2, 4, 5]
a.count - 1
>> 2
a.count - 1
>> 0
a.count 4
>> 1
def f; [1]; end
>> nil
f[0]
>> 1
f [0]
>> ArgumentError
```

```
MAX = 10
>> 10
MAX = MAX + 1
>> warning
```
namespace

name indicates kind
› variable is local by default
› @x is an instance variable
› @@x is a class variable
› $x is a global variable

modules
› like Java packages
› with classes, help structure namespace

```ruby
module Util
  def Util.inc (x); x+1; end
end

Util.inc(5)  #=> 6
```
values, types, conversions

floats and integers
› distinct, as in Java and Python

everything’s an object
› not only values (eg, numbers)
› also classes (this is reflection)
› including nil: `x.nil?` test

to string
› method `to_s` returns string
› not defined on built-ins (phew!)

```
1/3
>> 0
1.0/3
>> 0.3333333333333333

3.times {puts "hi"}
hi
hi
hi
>> 3
3.1415926.to_s
>> 3.1415926

1 + "2"
>> TypeError
```
strings & arrays

same operators for each
› a[i], a[i..j], <<, +

strings are mutable
› a+b: concatenation of a and b, as new string
› a << b: append b to a, modifying it

expression substitution
› inside a string literal, #{e} evaluates e

```
"answer is #{1 + 2}"
>> "answer is 3"
```

```
a = [2,4,6,8]
>> [2,4,6,8]
a[1..2]
>> [4,6]
s = "hello"
>> "hello"
t = s
>> "hello"
s += " there"
>> "hello there"
s
>> "hello there"
t
>> "hello"
```
functions and methods

define a method like this
›  def name (args)
      body
  end

a method defined outside a class
›  behaves like a function
›  (actually a private method of Object)
exercise

open a ruby shell
  › http://tryruby.org
  › on OS X:  > irb
  › format code with spaces for indents, and copy/paste

write a function *indent* that...
  › takes a string *s*, count *k*, indentation string *i*
  › returns *s* prefixed with *k* copies of *i*

```ruby
def indent(s, k, i)
  k.times {s = i + s}
  s
end
indent("hello", 3, "..")
```
hashes & symbols

p = {"first" => "James", "last" => "Bond"}
>> {"first" => "James", "last" => "Bond"}
p["first"]
>> "James"

p = {first => "James", last => "Bond"}
>> {first => "James", last => "Bond"}
p[first]
>> "James"

hashmaps
› often just called “hashes”
› built-in to Ruby

similar to JavaScript objects
› can update and replace slot
› but any type of key (even mutable!)

symbols, a built-in type
› immutable and no string methods (good!)
classes

instance variables
› created when assigned
› never directly accessible
› initialized with initialize method

method
› returns last expression evaluated
infix methods

=      Set field
[ ] [ ]=  Element reference, element set
**     Exponentiation
! ~ + -  Not, complement, unary plus and minus (method names for last two are +@ and -@)
* / %    Multiply, divide, and modulo
+ -      Plus and minus
>> <<    Right and left shift
&       Bitwise and
^ |      Bitwise exclusive `or' and regular `or'
<= < > >=  Comparison
<=> == === != =~ !~  Equality, pattern match (!= and !~ defined only implicitly)

can define methods for infix operators
>  o.f = e : call method f= on object o with arg e

class Coord
  def x
    @x
  end
  def x=(a)
    @x = a
  end
end
c = Coord.new
>> #<Coord:0x10ddd5190>
c.x = 5
>> 5
c.x
>> 5
accessors

attr_reader, attr_writer, attr_accessor
  › generate get and set methods

access control
  › instance vars are never accessible
  › this is the only way to get/set them

private and public methods
  › methods are public by default
  › keyword private between methods makes all following methods private
self & class methods

special variable `self`, refers to
› receiver in instance methods
› class in class methods

class Msg
  attr_reader :str
  @@count = 0;
  def self.reset
    @@count = 0;
  end
  def initialize(s)
    @seq = @@count
    @str = s
    @@count = @@count + 1
  end
  def to_s
    "str: #{self.str}, seq: #{@seq.to_s}"
  end
end

Msg.new("hi").to_s
>> "str: hi, seq: 0"
Msg.new("hi").to_s
>> "str: hi, seq: 1"
Msg::reset
>> 0
Msg.new("hi").to_s
>> "str: hi, seq: 0"

puzzle: is @x = e equivalent to self.x = e?
write a class `Person` that...

› provides public “fields” `first` and `last`
› allows no changes to `first`, at most 2 changes to `last`

```ruby
class Person
  def initialize(first, last)
    @first = first
    @last = last
  end
end

p = Person.new("Ben", "Bernanke")
p.last >> "Bernanke"
p.last = "Bertucci"
p.last >> "Bertucci"
p.last = "Bitdiddle"
p.last >> "Bitdiddle"
p.last = "Bernanke"
p.last >> "Bitdiddle"
```
code blocks

code blocks
› just anonymous functions
› use *do/end* or {} 

special syntax
› when just one fun passed
› not named as arg
› called with *yield*

def from_to (from, to)
    i = from
    while i <= to
        yield i
        i = i + 1
    end
end

from_to(1,3) do |i|
    puts i
end

# how you’d actually do it
for i in (1..3)
    puts i
end
set example

like functions

› iterators often implemented using iterators...

class Set
  def initialize
    @elts = []
  end
  def insert(e)
    @elts << e unless self.contains?(e)
    self
  end
  def contains?(e)
    @elts.each do |x|
      return true if (x == e)
    end
    return false
  end
  def each
    @elts.each do |e|
      yield e
    end
  end
  def size
    @elts.count
  end
end

s = Set.new
s.insert(1).insert(2).insert(3)
total = 0
s.each do |e|
  total += e
end
puts total # prints 6
functionals

also implemented with blocks
› Ruby arrays have map etc
› what do these do?

```ruby
[1, 2, 3].each { |value| puts value }
[1, 2, 3].map { |value| 2**value }
[1, 2, 3].reject { |value| value % 2 == 0 }
```
Ruby blocks vs. JavaScript functions

what’s the difference?
› compare these two examples
› can you figure it out?

```ruby
def contains?(e)
  @elts.each do |x|
    return true if (x == e)
  end
  return false
end
```

```javascript
var Set = function () {
  var elts = []; var that = {};
  that.insert = function (e) {
    if (!that.contains(e)) elts.push(e); return that;
  }
  that.contains = function (e) {
    var result = false;
    elts.forEach(function (x) {if (x === e) result = true;});
    return result;
  }
  that.size = function () {return elts.length;};
  return that;
}
```

ps: Ruby also has lambda
› same as JavaScript’s function
write a function \textit{sum} that
\begin{itemize}
  \item takes an array \textit{a} of numbers
  \item returns their arithmetic sum
\end{itemize}

\begin{verbatim}
def sum(a):
    sum = 0
    a.each { |e| sum = sum + e }
    sum

sum [1,2,3]
\end{verbatim}
superclasses and mixins

inheritance
› very similar to Java
› *class A < B* means *A* extends *B*

mixins
› unlike Java, methods from >1 parent
› parents are modules, not classes
**mixin example**

class Set
  def initialize
    @elts = []
  end
  def insert(e)
    @elts << e unless self.contains?(e)
  end
  def contains?(e)
    @elts.each do |x|
      return true if (x == e)
    end
    return false
  end
  def each
    @elts.each do |e|
      yield e
    end
  end
  def find (&b)
    self.each do |e|
      return e if b.call(e)
    end
    return false
  end
end

s = Set.new
s.insert(1).insert(2).insert(3)
puts s.find{|e| e > 1} # prints 2

**find** method written in Set

module Enumerable
  def find (&b)
    self.each do |e|
      return e if b.call(e)
    end
  end
end

class Set
  include Enumerable
  def initialize
    @elts = []
  end
  def insert(e)
    @elts << e unless self.contains?(e)
  end
  def contains?(e)
    @elts.each do |x|
      return true if (x == e)
    end
    return false
  end
  def each
    @elts.each do |e|
      yield e
    end
  end
  def find (&b)
    self.each do |e|
      return e if b.call(e)
    end
    return false
  end
  def each
    @elts.each do |e|
      yield e
    end
  end
end

**find** method from mixin

define the mixin

include the mixin

another syntax for blocks