Intro to Ruby

Bryce Kerley

November 9, 2006

Ruby is an object-oriented language that is becoming increasingly popular for tasks of all sizes, from short automation scripts to complicated websites. It has achieved this with terse syntax, a feeling of similarity to users familiar with many other languages (Ruby draws heavily from Perl, Smalltalk, LISP, and yet remains somewhat familiar to C users), and a well-designed set of core and standard library tools.

1 Installing Ruby

- Microsoft Windows - get the installer from `http://www.ruby-lang.org/en/downloads/`

- Mac OS X 10.2 and newer - it's installed already, but some applications (Metasploit 3) require an upgrade to the newest version. Use MacPorts or Fink to get it (`sudo port install ruby`).

- Other UNIX-like operating systems - use your distribution's package manager (`sudo emerge ruby`, `sudo apt-get install ruby`, `sudo yum install ruby`, etc.) or download and compile the source from `http://www.ruby-lang.org/en/downloads/`

- TOO LAZY - there's an online interpreter at `http://tryruby.hobix.com/`

2 Documentation

Ruby has a large and growing set of documentation available online:

- Ruby-Doc.org - Core and Standard API documents generated from source, links to other documentation. `http://ruby-doc.org/`

- Programming Ruby: The Pragmatic Programmer's Guide (Pickaxe book) - Good reference for syntax and semantics; the first edition is available at `http://ruby-doc.org/docs/ProgrammingRuby/`.

- Why's Poignant Guide to Ruby - Self-indulgent and humorous tutorial with interesting examples and illustrations, free at `http://qa.poignantguide.net/`

3 Getting Started

Ruby's used in two common situations - either running a script through (the `ruby` executable), or as an interactive interpreter (`irb`, or through `http://tryruby.hobix.com`).

This is an example of a script - you can save this and try it.

```
1    #!/usr/bin/env ruby
2    # comments in Ruby start with the octothorpe
3    puts "Hello world!"
```

Running this in `ruby`:

```
> ruby hello.rb
Hello world!
```

And in `irb`:

```
irb(main):001:0> puts "Hello world!"
Hello world!
=> nil
```

As we can see from the ""`=> nil`" in the `irb` example, the interpreter helpfully prints out the value the previous statement evaluates to. This means you can mess around in the interpreter and see results without having to litter your code with printing statements.

```
irb(main):002:0> 10 + 5
=> 15
irb(main):003:0> 589**13
=> 102683067063535245896688355245399869
```

Like LISP, Ruby doesn't get messed up by large integers - its math functions will switch their type to a larger one as necessary.

```
irb(main):004:0> (10 + 5).class
=> Fixnum
irb(main):005:0> (2353 ** 2135).class
=> Bignum
```

4 Simple flow control

Ruby has `if` and `while`, and their opposites `unless` and `until`. The opposites are purely superfluous, but they do make for easy reading:

```
irb(main):001:0> puts "40 isn't 39" unless 40 == 39
40 isn't 39
=> nil
irb(main):002:0> puts "40 isn't 39" if 40 != 39
40 isn't 39
```

```
=> nil
irb(main):003:0> puts "40 isn't 39" if !(40 == 39)
40 isn't 39
=> nil
```

There's two styles for these flow controls. Above, they're postfixed to a statement, and affect only that statement. If you have more than one statement to be conditioned, use it block-style:

```
irb(main):004:0> if (40 != 39)
irb(main):005:1> puts "40 isn't 39"
irb(main):006:1> end
40 isn't 39
=> nil
```

5   Core Classes

The included functionality in Ruby is split up into two sections - core and the standard library. Core classes expose the language's power and features, while standard library modules and classes use that power to connect it to the world outside Ruby or extra features. Detailed documentation on the core is available at `http://ruby-doc.org/core/`.

5.1   Nil and Booleans

`nil` (the only possible instance of `NilClass`) is the rough equivalent of `null` in many other languages. Unlike LISP's `nil`, it's not a list. It's simply nothing at all.

true and `false` (from the classes `TrueClass` and `FalseClass`, respectively) are what many of the flow-control constructions expect (more on these). `false` and `nil` are the only things that are ever logically false. Even the number zero (0) evaluates as `true`. This is important when finding things in strings and arrays.

5.2   Numeric types

The most common classes you'll see here are `Fixnum` and `Float`. `Float`s behave much like `Fixnum`s, except there's no other type they get converted to.

Ruby also has `Bignum`, `Rational`, and `Complex` classes. `Bignum` is functionally identical to `Fixnum` (and switching between the two happens as needed), except it holds even bigger numbers. `Rational` provides non-integer operations without loss, and `Complex` allows the use of imaginary numbers in calculations.

5.3   Strings

Ruby strings are first class objects, and can be sliced up, split, and concatenated with ease.

```
irb(main):001:0> x = "It's my birthday!"
=> "It's my birthday!"
irb(main):002:0> x.class
=> String
irb(main):003:0> x + "  Woot!"
=> "It's my birthday!  Woot!"
irb(main):004:0> x[0..3]
=> "It's"
irb(main):005:0> x.split
=> ["It's", "my", "birthday!"]
```

Some kinds of object in Ruby have exclamation point versions of methods (such as `String#upcase!`). These more-exciting versions of the method don't just return the result of the operation, they change the object too:

```
irb(main):006:0> x.upcase
=> "IT'S MY BIRTHDAY!"
irb(main):007:0> x # still the same
=> "It's my birthday!"
irb(main):008:0> x.upcase!
=> "IT'S MY BIRTHDAY!"
irb(main):009:0> x # now upcased
=> "IT'S MY BIRTHDAY!"
```

Literal strings can be expressed several ways:

```
irb(main):001:0> "test"
=> "test"
irb(main):002:0> 'test'
=> "test"
irb(main):003:0> <<END
irb(main):004:0" this is a really long string
irb(main):005:0" look how big it is
irb(main):006:0" END
=> "this is a really long string\nlook how big it is\n"
irb(main):007:0> x = 24
=> 24
irb(main):008:0> "the number is #{x}"
=> "the number is 24"
irb(main):009:0> %Q{The number is #{x}}
=> "The number is 24"
irb(main):010:0> 'the number is #{x}'
=> "the number is \#{x}"
```

Lines 1 and 2 both produce strings (indeed, the same string). Lines 3-6 demonstrate a Heredoc, used to make really long multi-line string literals. The same could be accomplished with the string returned at the end of line 6.

Lines 8 and 9 demonstrate expression interpolation (the contents of the curly braces can be any Ruby expression), and line 10 shows why you might want to use single-quotes sometimes.

5.4   Regular Expressions

Ruby provides regular expressions (in the `Regexp` class) for matching strings.

```
irb(main):001:0> x = "Let's match things"
=> "Let's match things"
irb(main):002:0> x =~ /match/
=> 6
irb(main):003:0> x =~ /Let's/
=> 0
irb(main):004:0> x =~ /Let'z/
=> nil
```

Line 2 demonstrates a simple regular expression: `/match/`. The slashes (`/`) surrounding it tell Ruby that it's a regex, and the contents (`match`) simply state that the regular expression looks for the letters 'm', 'a', 't', 'c', and 'h' in sequence. Running this against the string returns `6`, which tells us that the regular expression first matched at character 6. (If the match fails, it returns `nil`.)

Lines 3 and 4 demonstrate why it's important that zero evaluate to `true`. Line 3 is a successful match, while line 4 is unsuccessful. You could immediately use this in an `if` statement without any additional comparisons.

```
irb(main):001:0> x = "A bad snackbar"
=> "A bad snackbar"
irb(main):002:0> x =~ /a/
=> 3
irb(main):003:0> x =~ /a/i
=> 0
irb(main):004:0> x =~ /a.*d/i
=> 0
irb(main):005:0> x =~ /a.*d/
=> 3
```

Lines 2 and 3 show how you switch to a case insensitive match (add an `i` after the final slash), and lines 4 and 5 show two things: the . wildcard (matches any character except a newline) and the `*` Kleene star closure.

```
irb(main):006:0> x =~ /^/
=> 0
irb(main):007:0> x =~ /$/
=> 14
```

The caret (`^`) and dollar sign (`$`) match the beginning and end of the line, respectively.

```
irb(main):008:0> x =~ /r|s/
=> 6
irb(main):009:0> x =~ /[rstlne]/
=> 6
```

The pipe (`|`) matches either the character to the left or right. In line 8, it matches the 's' in "snackbar". Square brackets (`[]`) create character classes. In line 9, it again matches the 's' in "snackbar".

You can use a regular expression to break a string into little pieces:

```
irb(main):010:0> x.split(/[rstlne]/)
=> ["A bad ", "", "ackba"]
irb(main):011:0> x.split(/[aeiou]/i).join
=> " bd snckbr"
```

In line 10, the string "A bad snackbar" was split on any of the "Wheel of Fortune" letters, returning an array containing the splits. In line 11, we instead split the string on vowels (being case insensitive), and join them back together to get the original string sans vowels.

For convenience, the regular expression engine used by Ruby provides special patterns to match common character classes:

```
irb(main):012:0> "13:37 is the best time of day" =~ /\d\d:\d\d/
=> 0
irb(main):013:0> "13:37 is the best time of day" =~ /\w\w\w\w/
=> 13
irb(main):014:0> "13:37 is the best time of day" =~ /\D\D/
=> 5
irb(main):015:0> "13:37 is the best time of day" =~ /\W\W/
=> nil
```

`/d` matches any digit, `/w` matches any word or digit character (but not punctuation), and the capitalized versions of these match non-digits and non-word characters respectively. Note line 15, which fails to match two non-word characters adjacent to each other.

5.5 Arrays

Strings in Ruby lead nicely into looking at arrays:

```
irb(main):001:0> s = "This here's a string"
=> "This here's a string"
irb(main):002:0> a = s.split
=> ["This", "here's", "a", "string"]
irb(main):003:0> a[3] = 'array'
=> "array"
irb(main):004:0> a
=> ["This", "here's", "a", "array"]
irb(main):005:0> a[2] += 'n'
```

```
=> "an"
irb(main):006:0> a
=> ["This", "here's", "an", "array"]
irb(main):007:0> a[4] = '!'
=> "!"
irb(main):008:0> a[10] = 4
=> 4
irb(main):009:0> a
=> ["This", "here's", "an", "array", "!", nil, nil, nil, nil, nil, 4]
```

The `String#split` method (when not passed an argument) splits a string into an array based on whitespace, as seen in line 2. Line 3 shows array assignment on top of an existing element, line 5 changes the string in an existing element, and line 8 shows what happens when you assign out of bounds (the spaces between get filled with `nil`s).

You can slice up an array into a subarray:

```
irb(main):010:0> a[0..4]
=> ["This", "here's", "an", "array", "!"]
irb(main):011:0> a[2..4]
=> ["an", "array", "!"]
```

One thing you might miss is the lack of `car` and `cdr` in Ruby. Let's fix that.

6   Classes and Methods

Ruby's object-oriented nature is extremely dynamic. You don't have to compile in new methods to classes or objects - you can just add them whenever needed.

We want to add `car` and `cdr` to Ruby's `Array` class. To do that, we simply get into the class and define the methods:

```
1    # carcdr.rb
2    class Array
3          def car
4                  self[0]
5          end
6          def cdr
7                  return self[1..-1]
8          end
9    end
```

Save this in `carcdr.rb`, and fire up `irb`:

```
irb(main):001:0> require 'carcdr.rb'
=> true
irb(main):002:0> x = [1, 2, 3, 4]
=> [1, 2, 3, 4]
```

```
irb(main):003:0> x.car
=> 1
irb(main):004:0> x.cdr
=> [2, 3, 4]
```

That seems to work as expected. However, we don't have the fancy cadaadr-style functions from LISP. That's a shame, innit?

Creating a new class is just as easy as redefining an existing one.

```
1   # disco.rb
2   class Disco
3
4     def initialize
5       #constructor
6       @dancers = Array.new
7       @currentsong = "Stayin\' Alive"
8     end
9
10    def to_s
11      return "#{@currentsong} playing to: #{@dancers}"
12    end
13
14    def song=(newsong)
15      @currentsong = newsong
16    end
17
18    def <<(dancer)
19      @dancers << dancer
20    end
21  end
```

Instance variables in Ruby start with the @ sign. All such variables are 'private' (in C++/C#/Java terms). How variables are made accessible to the world is with assignment methods. Above, we see the `song=` method being defined, as well as the `<<` and `to_s` methods.

Let's use them:

```
irb(main):001:0> require 'disco'
=> true
irb(main):002:0> x = Disco.new
=> #<Disco:0x47bc88 @currentsong="Stayin' Alive", @dancers=[]>
irb(main):003:0> x << Charles
NameError: uninitialized constant Charles
        from (irb):3
irb(main):004:0> x << "Charles"
=> ["Charles"]
irb(main):005:0> x << "Bill"
```

```
=> ["Charles", "Bill"]
irb(main):006:0> x << "John"
=> ["Charles", "Bill", "John"]
irb(main):007:0> x.song = "Remind Me"
=> "Remind Me"
irb(main):008:0> x.to_s
=> "Remind Me playing to: CharlesBillJohn"
```

Yuch, look at the output of `Disco#to_s`. We can do better:

```
irb(main):009:0> class Disco
irb(main):010:1> def to_s
irb(main):011:2> "#{@currentsong} playing to #{@dancers.join(', ')}"
irb(main):012:2> end
irb(main):013:1> end
=> nil
irb(main):014:0> x.to_s
=> "Remind Me playing to Homer, Bill, John"
```

Changing a method in a class changes the method in every instance of the class. Sometimes you want this (it's fine in this case, since nobody's used the Disco since 1981).

6.1   Iterators

Let's say something about our disco friends:

```
x = [irb(main):001:0> x = ['Homer', 'Bill', 'John']
=> ["Homer", "Bill", "John"]
irb(main):002:0> x.map do |v|
irb(main):003:1* v + " is a Disco freak!"
irb(main):004:1> end
=> ["Homer is a Disco freak!", "Bill is a Disco freak!", "John is a Disco freak!"]
```

The `Array#map` method is a lot like the `mapcar` function in LISP. It takes a special kind of object, a `Proc`, as an argument. It then executes that argument on every element of the array, and makes a new array out of the values the block (proc and block are used interchangeably) returns when called.

`map` isn't the most frequently used iterator, `each` is:

```
1   # proc.rb
2   x = ["zeroth", "first", "gaius baltar"]
3   n = 0
4   x.each do |e|
5     puts "The #{n}th entry is #{e}"
6     n += 1
7   end
```

```
> ruby proc.rb
The 0th entry is zeroth
The 1th entry is first
The 2th entry is gaius baltar
```

The calling process behind a block is more complicated than these examples give credit for.

1. The contents of the block (between the `do` and `end` are collapsed into a `Proc` object).

2. The arguments to the block (between the pipe symbols) are counted and stored as the proc's `arity`.

3. The method is passed the block.

4. At some point during the method, `yield` is called with arguments.

5. The proc's `call` method is called by Ruby with the arguments given to `yield`.

6. The code inside the proc executes.

While this is complicated, it allows Ruby programmers to even keep different procs hanging around in variables to be used at their discretion. You don't even have to write a proc yourself, but can pull a method off an object and use it as one:

```
irb(main):001:0> p = 10.method('+').to_proc
=> #<Proc:0x0047dad8@(irb):1>
irb(main):002:0> p.call(5)
=> 15
```

Later, we'll see what we can do with an anonymous proc object.