Introduction to Ruby

Mark Armstrong

October 20, 2020

Contents

1 Introduction

These notes were created for, and in some parts **during**, the lecture on October 9th and the following tutorials.

2 Motivation

So far, we have been investigating

- the functional paradigm, using Scala,
  - which happens to also be a pure object-oriented language, and
- the logical paradigm, using Prolog.

We now investigate an *imperative* pure object-oriented language, Ruby.

Ruby's syntax is also heavily influenced by Lisp, and the final language we will investigate later in the course is a Lisp. So Ruby acts as a stepping stone to that point.

3 Background on Ruby

Ruby is an almost purely object-oriented language, heavily inspired by Smalltalk, Lisp and Perl.

It's creator, Yukihiro "Matz" Matsumoto, has documented these inspirations in a post on ruby-talk.

> Ruby is a language designed in the following steps:
>
> - take a simple lisp language (like one prior to CL).
> - remove macros, s-expression.
> - add simple object system (much simpler than CLOS).
> - add blocks, inspired by higher order functions.
> - add methods found in Smalltalk.
> - add functionality found in Perl (in OO way).

Ruby was a language born out of Matz's interests and love of certain language features. He's said about its creation

> Well, Ruby was born on February 24 1993. I was talking with my colleague about the possibility of an object-oriented scripting language. I knew Perl (Perl4, not Perl5), but I didn't like it really, because it had smell of toy language (it still has). The object-oriented scripting language seemed very promising.
>
> I knew Python then. But I didn't like it, because I didn't think it was a true object-oriented language OO features appeared to

> be add-on to the language. As a language manic and OO fan for 15 years, I really wanted a genuine object-oriented, easy-to-use scripting language. I looked for, but couldn't find one.
>
> So, I decided to make it. It took several months to make the interpreter run. I put it the features I love to have in my language, such as iterators, exception handling, garbage collection.
>
> Then, I reorganized the features of Perl into a class library, and implemented them. I posted Ruby 0.95 to the Japanese domestic newsgroups in Dec. 1995.

4 General syntax

See the Ruby documentation for a good overview of the syntax. We try to cover the most important parts here and in the following sections.

Ruby, being a "scripting language", has a somewhat C-like syntax that may remind you of Python or possibly Javascript.

The major difference between Ruby and Python is that whitespace does not indicate structure in Ruby.

Instead, to indicate the beginning and ending of a scope, we use `begin` and `end` keywords. (`begin` is not used if there is already an indication of the beginning of the scope.)

```ruby
# Denote the end of a method with an end keyword
# begin is not used, since we have a def.
def f(x)
  return x
end

puts f(5)
```

Ruby has a good variety of control structures.

```ruby
if (true)
  puts "hello"
end

if false
  puts "goodbye"
elsif true
  puts "world"
```

```ruby
else
  puts "I don't get here."
end

# There is the "ternary if"
true ? puts("first one") : puts("second one")
x = true ? "first one" : "second one"
puts x

# It's not needed really.
y = if true then "this works" else "or does it?" end
puts y

# We don't need to negate conditions; we have an unless
 ↪  operator
unless true
  puts "The meaning of life"
else
  puts "You're out of luck"
end

# There is a switch statement
x = 12345
case x
when 12345
  puts "here we are"
when 123
  puts "not here"
else
  puts "and not here"
end
```

Not discussed here:

- `while` loops
- `until` loops
- `for` loops
- `break` statements

- `next` statements
  - (skips to the next iteration in a loop.)

5 Dynamic typing, heterogenous collections

In most languages, lists, arrays and (hash)maps are *homogenous* collections; they include only one type of element.

This is not the case in a dynamic (typed) language such as Ruby.

So we can have arrays and maps with multiple types.

```
a = ["Arrays", :can, "contain", 1, "or more different types"]
h = {:and => "So can hashes.", "see" => a, 0 => 12345}

print a
print "\n"
print h
```

6 "Purely object-oriented"

What does it mean when I have said, both about Ruby and Scala, that they are purely object oriented?

Quite simply, that all *data* is represented as an *object, and all operations are /methods*!

Another view of this: all data is objects, and all operations are messages between objects.

6.1 Integers are objects, operations are methods

For instance, consider integers, which in many languages are a *basic*, *builtin* type that do not have an implementation within the language. This is not the case in a pure language! In both Ruby and Scala, integers are objects, and operations on them are methods.

So code such as

```
5 + 8
```

or in Scala,

```
5 + 2
```

could instead be written

```
5.+(9)
```

```
5.+(2)
```

that is, `+` is a method of the first argument, being passed the second argument as a parameter. The form `x` $\oplus$ `y` is just *syntactic sugar*.

6.2 Integers are objects, operations are *messages*

In Ruby, we can move one level of abstraction higher: all data are objects, and all operations are *messages between objects*. This harkens back to SmallTalk, one of the founding languages of the object oriented paradigm.

```
5.send("+", 3)
```

So even the form `x.`$\oplus$`(y)` is syntactic sugar!

In Scala, moving to this message passing abstraction is not possible, at least not easily; why?

- Answer: because Scala is statically typed! It is unlikely the typechecker will parse a string to see what method is being invoked.
  - And even if we wanted to make the typechecker do so, that only handles to case of constant strings; what if the string is constructed earlier in the program, or its value is given as an argument?

7 Output methods

There are at least three useful output methods in Ruby.

- `print` prints out an object without a newline at the end.
- `puts` does output a newline, and on arrays, outputs each element on its own line.
- `p` just outputs raw objects.

```
print(["Prints", "out", "the", "object", "without", "a",
 ↪  "new", "line"])
print "\n"

puts(["Each", "element", "on", "a", "newline"])
```

```
puts "And a newline at the end as well."
puts "See?"

print "Outputs a string with a newline\n"
p "Outputs the raw string\n"
```

8 Postfix forms

One nice feature of Ruby is that it has an abuntant amount of syntactic sugar. For instance, we have the standard control structures such as `if`:

```
x = if 2 + 3 == 5
      "2 + 3 is 5; good"
    elsif 2 + 3 == 6
      "oh no!!!"
    else
      "oh no"
    end

print("x is: ", x)
```

If you are only using a "then" branch, you can write the `if` after the body.

```
puts("1 + 1 is 2") if 1 + 1 == 2
```

We can do the same with at least some of the loops

```
x = 0
y = 0

x = x + 1 while x < 10
while y < 10 do y = y + 1 end

puts(x)
puts(y)
```

9 Method naming conventions

Note; in this section, we use the phrase "by convention" frequently. This is because the method naming conventions are exactly that: conventions. Their properties are not actually enforced by Ruby.

By convention, methods ending with a `?` are *predicates*. They do not *necessarily* return a boolean, but should return a "truth value" of some kind.

```
6.even?
```

By convention, methods ending with a `!` are *destructive*; they modify the receiver.

```
x = [1,3,2]
y = [5,6,4]

x.sort! # Changes the value of x
z = y.sort # Does not change the value of y.

print("[1,3,2] sorted is ", x, "\n")
print("The original ", y, " did not get sorted; see?\n")
print("Here's the sorted version: ", z)
```

See the Ruby documentation on method names for discussion of the the bang and question mark postfixes.

Methods ending with a `=` indicate an *assignment* method. These methods should "behave like assignment". See the Ruby documentation. Unlike with the `?` and `!` postfixes, **using the `=` postfix *does* actually change the method's behaviour**; specifically, the return value of a `=` method is ignored. The arguments to the method are returned instead.

Assignment methods can be called using a special syntax; given an assignment method `my_attribute=` and an instance of the class `my_instance`, we can write

```
my_instance.my_attribute = "A new value"
```

instead of

```
my_instance.my_attribute=("A new value")
```

10 Defining classes

10.1 The basics

A class declaration in Ruby for a class named `Name` is begun by simply saying

```
class Name
```

Instance variables (whose value is unique per object of the class) begin with a `@`. We do not declare explicitly declare variables in Ruby, but you can initialise them to "declare" them (you don't need to though; they can be initialised inside a constructor or other methods.)

Class variables (whose value is shared by each object of the class) begin with a `@@`. (In other languages, these are often called `static` members.)

It is common to want to define *getters* and *setters* for instance variables in OO programming. For example,

```
class MyContainer

  def initialize(thing=nil)
    @thing = thing
  end

  def thing
    @thing
  end

  # Assignment method syntax
  def thing=(thing)
    @thing = thing
  end
end

container = MyContainer.new()
container.thing = 5
puts(container.thing)
```

Because this is so common, there is a shorthand to avoid declaring these methods.

```
class MyContainer
  attr_accessor :thing   # :thing is a symbol; essentially in
   ↪  interned string
  # attr_reader   provides only the getter
  # attr_writer   provides only the setter

  def initialize(thing=nil)
    @thing = thing
  end
```

```
end

container = MyContainer.new()
container.thing = 5
puts(container.thing)
```

:TODO: initialize is the constructor

10.2 Inheritance

:TODO:

```
class C1

class C2 < C1
```

10.3 TODO Mixins

https://ruby-doc.com/docs/ProgrammingRuby/html/tut_modules.html