Introduction to Ruby

Reuven M. Lerner

Abstract

Everything you need to know to start programming in Ruby.

We programmers are lucky to be working today. I say this because there are so many excellent programming languages from which to choose, especially in the Open Source world.

One of the most talked-about languages is Ruby. Ruby isn't actually all that new. Yukihiro “Matz” Matsumoto released the first public version in 1995, and it has grown in popularity ever since. As the Ruby on Rails framework for Web development has become increasingly popular, interest in Ruby has soared along with it.

Ruby often has been described as a cross between Perl and Smalltalk, and I don't think this is a bad way to look at it. Certainly, if you have experience with both Perl and object-oriented programming, you probably will feel right at home working with Ruby.

In this article, I introduce the basics of Ruby, showing how it is similar to other high-level languages and where it adds its own, special twist. By the end of this article, I hope you'll know enough about Ruby to try it out for some small applications. If you're like me, you'll quickly discover that Ruby is surprisingly compact and elegant, making it possible to write maintainable code quickly and easily.

The Basics

Downloading and installing Ruby is fairly easy, particularly because a recent version (1.8.2) is included with many distributions of Linux. You either can use that version or install the latest version (1.8.4) from the main Ruby site. As an open-source product, you shouldn't be surprised to find that the main Ruby site (http://www.ruby-lang.org) offers the source code in .tar.gz format. Additional formats, such as RPMs and Debs, are available from the official repositories for your favorite distribution.

If you want to install the latest version of Ruby from source, download and unpack the .tar.gz file:

```
$ cd Downloads
$ tar -zxvf ruby-1.8.4.tar.gz
```

Now use the standard configure program to find the system configuration automatically, make to compile it and then make test to ensure that the compiled version of Ruby works correctly:

```
$ ./configure && make && make test
```

If all goes well, the final line of output from the above commands will read **`test succeeded`**. Now you can become the root user and install Ruby onto your system:

```
$ su
# make install
```

This installs a variety of Ruby programs and libraries onto your computer.

Interactive Ruby: Irb

The Ruby language itself exists as an executable called ruby, which you can run manually by typing it on the command line:

```
$ ruby
```

However, this version of Ruby is designed for non-interactive use. To test code or experiment with the Ruby language, there is irb, the interactive Ruby shell. Irb is something like a debugger, in that it takes input from a user (terminated by pressing the Enter key) and executes it. For example, type:

```
$ irb
```

And, irb responds with its prompt:

```
irb(main):001:0>
```

Now we can type a bit of Ruby:

```
irb(main):001:0> print "Hello, world"
```

And, irb responds with:

```
Hello, world=> nil
```

The above output indicates that print displays Hello, world on the screen and returns a nil value; nil is Ruby's way of representing a null value, much like undef in Perl, None in Python and NULL in SQL.

Like many other high-level languages, Ruby allows us to assign values to variables without pre-declaring them. Thus, we can write:

```
greeting = "Hello, world"
print greeting
```

Ruby also can do math, using the familiar operators +, -, * and /:

```
5 + 3
60 - 23
60 * 23
10 / 2
```

I have omitted the call to print in the above lines, because it's unnecessary in irb. However, in a standalone Ruby program, no output would be sent to the screen (or elsewhere) without using print.

If you are a seasoned Perl programmer, you might be somewhat surprised to discover the result of the following:

```
5 / 2
```

The above returns 2 because both 5 and 2 are integers, and Ruby assumes you want to perform integer arithmetic. To get a floating-point result, you must ensure that at least one of the numbers is a float:

```
5 / 2.0
```

Sure enough, that returns 2.5. Unlike many other languages, Ruby requires a leading 0 for fractional numbers; you must say 0.5, rather than .5.

You can convert a string to an integer or float using the to_i and to_s methods:

```
"5".to_i
"5".to_f
```

All objects in Ruby have a similar to_s method, which turns the object into a string.

One datatype in Ruby that surprises some newcomers is the symbol. You can think of symbols as special kinds of strings that take up far less room in memory, especially when they are used in multiple locations. Symbols, which begin with a colon (for example, :reader) cannot always be used in place of strings, but they allow programmers to make programs more readable. They also are used on occasion to refer to objects and methods, as I explain later in this article.

Interpolation and Methods

Like many other high-level languages, Ruby lets us interpolate values inside of double-quoted strings. (Single-quoted strings are taken literally, as is the convention in many other languages.) For example:

```
name = "Reuven"
"Hello, #{name}"
```

The above expression is equivalent to:

```
Hello, Reuven
```

Within the #{ }, we can put any Ruby expression, not only a variable name:

```
name = "Reuven"
print "Hello, #{name}. Your name is #{name.length} letters long."
print "Backwards, your name is '#{name.reverse}'."
print "Capitalized, your backwards name is '#{name.reverse.capitalize}'."
```

As you can see, interpolation lets us put arbitrarily complex expressions within a double-quoted string. But wait a second—what are we doing with the expressions name.length, name.reverse and name.reverse.capitalize?

The answer is that strings, like everything in Ruby, are objects. Nearly anything we will do with a string is expressed as a method, rather than as a standalone function. If you want to reverse a string, get its length, capitalize it or break it apart, you will invoke a method on the object using Ruby's object.message syntax. For example:

```
name.reverse
```

The above code returns a new string object, whose value is the reverse of name. Name itself is not

altered in the process. Because this new returned object is also a string, we can invoke any string method on it, including capitalize, as we saw before. Ruby programmers often end up chaining methods together to accomplish a task.

Methods invoked on an instance of an object are often referred to as Object#method in Ruby documentation. So, the above method would be referred to as String#reverse.

How do we know to which methods a particular object will respond? One way is to ask the object what class it is:

```
name.class
```

We also can ask an object whether it is a member of a particular class:

```
name.is_a?(String)
```

This might look a bit strange, both because of the question mark in the method name and the parameter that comes after it. But it works just like the other methods we have invoked so far. We send an is_a? message to name, which returns a Boolean (true or false) response. The argument to is_a? is a class name, which is String.

If we would prefer not to look up the API documentation for Ruby strings, we simply can ask the object itself what methods it will respond to:

```
name.methods
```

This returns an array (that is, a list) of methods to which name responds. We will look at arrays in a moment, but it's important to realize that name.methods is not a string; rather, it's an array whose contents happen to be strings. However, arrays respond to a built-in sort method, which returns a new array whose contents are ordered:

```
name.methods.sort
```

I probably invoke OBJECT.methods.sort at least once each day, rather than look through a book or on-line API for Ruby.

Arrays and Hashes

If you have worked with Perl or Python in the past, you won't be surprised to learn that Ruby has built-in arrays (as mentioned above) and hashes. We create an array with square brackets:

```
an_array = [1, "two", true]
```

An array can contain any number of objects, and each object can be of any type, including another array. The above array contains three objects (of types Fixnum, String and Boolean, respectively). Each item in an array has a unique index; the first element has an index of 0. We can retrieve items as follows:

```
an_array[1]
```

The above expression returns **`"TWO"`**, the item with an index of 1 in an_array. Arrays are mutable, meaning that we can replace any of the items by assigning to that index:

```
an_array[1] = "TWO"
```

We can use a negative index to count from the back of the array; thus an_array[-1] returns the Boolean value true. We also can view a subset of the original array by passing two indexes separated by a comma, indicating the first and last index that we want:

```
an_array[0,1]
```

To combine all of the elements of an array into a string, we can use the join method, for example:

```
an_array.join(", ")
```

The above code creates a single string, whose contents are the values from an_array, with ", " between each pair of elements.

Hashes are similar to arrays, except that instead of storing values using an ordered, numeric index, they are stored with keys, for example:

```
my_hash = {'a' => 1, 'b' => 2}
```

We can now retrieve either of the two values, by using its key:

```
my_hash['a']
my_hash['b']
```

The above lines of code return the numbers 1 and 2, respectively. As with arrays, we can store any object as a value in a hash; it doesn't have to be an integer.

We can retrieve the keys and values of a hash with the Hash#keys and Hash#values methods, respectively. (Later, I explain how to iterate over the keys and values to retrieve contents from a hash.) Sometimes, however, we simply want to know if a particular key exists in a hash. This is easily accomplished with Hash#has_key?, which takes a string as a parameter and returns a Boolean value. The following code thus would return true:

```
my_hash.has_key?("a")
```

Conditionals

Every language lets us execute code conditionally. In Ruby, this normally is done with an if statement. Consider the following (somewhat contrived) example:

```
if server_status == 0
print "Server is in single-user mode"
elsif server_status == 1
print "Server is being fixed "
elsif network_response == 3
print "Server is available"
else
print "Network response was unexpected value '#{network_response}'"
end
```

Notice that Ruby does not require parentheses around the condition. And although the condition does not have to return a Boolean value, Ruby will produce a warning if you try to use = (that is, assignment)

in the condition, rather than == (that is, comparison). The == comparison operator works on all objects; there are no separate text comparison and numeric comparison operators as in Perl. This is true for < and > also, which can be used to compare strings as well as numbers. Finally, Ruby does not use opening or closing braces; instead, it closes the conditionally executed block of code with end.

As with Perl, you can use if and unless as suffixes to make a statement conditional:

```
print "We won!" if our_score > their_score
print "Here is your change of #{amount_paid - price}!"
    unless amount_paid <= price
```

You also can do things like:

```
if inputs.length < 4
    print "Not enough inputs!\n"
end
```

And, also:

```
if not my_hash.has_key?("debug")
    print "Debugging is inactive.\n"
end
```

Loops

Ruby does have some looping operators, such as for and while. But the real fun and excitement is in doing things such as this:

```
5.times {print "hello\n"}
```

Think about it—we're invoking a method on a number, using the standard Ruby method-invocation syntax. The times method for integers executes a block of code a particular number of times. So, the above line of code executes five times, printing the word hello (followed by a new line) each time.

Blocks can take parameters as well, between pipe (|) characters:

```
5.times {|iteration| print "Hello, iteration number #{iteration}.\n"}
```

We similarly can iterate over the elements of an array with the each method:

```
an_array = ['Reuven', 'Shira', 'Atara', 'Shikma', 'Amotz']
an_array.each {|name| print "#{name}\n"}
```

A variation of the each method, called each_with_index, requires a block that takes two parameters. The first parameter is the item, and the second is the index:

```
an_array = ['Reuven', 'Shira', 'Atara', 'Shikma', 'Amotz']
an_array.each_with_index {|name, index| print "#{index}: #{name}\n"}
```

At a certain point, blocks become difficult to read in this syntax. Ruby provides an alternate syntax, replacing the curly braces with do and end:

```
an_array = ['Reuven', 'Shira', 'Atara', 'Shikma', 'Amotz']
an_array.each_with_index do |name, index|
    print "#{index}: #{name}\n"
end
```

We can iterate over a hash in several ways. One way is to use the type of iteration that Perl and Python programmers have used for years, getting the hash's keys (via Hash#keys, which returns an array) and then grabbing the value that goes with the key:

```
state_codes = {'Illinois' => 'IL', 'New York' => 'NY',
               'New Jersey' => 'NJ', 'Massachusetts' => 'MA',
               'California' => 'CA'}

state_codes.keys.each do |state|
    print "State code for #{state} is #{state_codes[state]}.\n"
end
```

Of course, we might want to sort the keys before iterating over them:

```
state_codes.keys.sort.each do |state|
    print "State code for #{state} is #{state_codes[state]}.\n"
end
```

Ruby provides an easier way to perform this task, the each_pair method:

```
state_codes.each_pair do |state, code|
    print "State code for #{state} is #{code}.\n"
end
```

Classes and Methods

Finally, we can put this all together in defining a class and some methods. We can create a class in irb, or anywhere else in Ruby, simply by saying:

```
class Simple
end
```

Sure enough, we've managed to create a class in only two lines. Is this enough to create an object of type Simple? Let's see:

```
foo = Simple.new
foo.class
```

It would seem so; our variable foo claims that it is of class Simple. We didn't specify what object Simple inherits from, so it automatically inherits from Object, the ultimate Ruby superclass. Ruby supports only single inheritance, which is stated in the class definition as:

```
class SimpleArray < Array
end
```

We already have defined two classes, which is nice, but we haven't defined any methods specific to those classes. Ruby allows us to open up a class at any time, adding or replacing methods in a class. We define a method with the def statement, indicating whether the method takes any parameters, for example:

```
class Simple
    def id_squared
        return self.object_id * self.object_id
    end
end
```

The method we have defined is quite simple, and it does something that I don't expect we would ever want to do—namely, it takes the object's unique ID (available via the inherited method object_id) and returns its doubled value (which will likely be an instance of Bignum).

If we type the above definition into irb, something amazing happens: our foo variable of class Simple now responds to the method Simple#id_squared! Yes, Ruby allows us to modify methods on the fly and to open up existing classes. We could, for example, modify the built-in Array or String classes, replacing the built-in methods with some of our own.

Finally, we might want to store some state in our object. This is done via instance variables. In Ruby, instance variables are preceded with the @ character, which might be a bit confusing if you are coming from the Perl world:

```
class Simple
    def initialize
        @simple_data = [ ]
    end
end
```

The special initialize method is invoked whenever we create a new instance of Simple. So if we once again define foo to be an instance of Simple:

```
foo = Simple.new
```

we can see that foo now has an instance variable defined, by invoking:

```
foo.instance_variables
```

The above returns an array:

```
["@simple_data"]
```

How can we assign to @simple_data? And how can we retrieve its value? One way is to define a number of methods: one for writing this instance variable and one for retrieving its value. But a shorthand way would be to use the attr_reader and attr_writer methods:

```
class Simple
    attr_reader :simple_data
    attr_writer :simple_data
end
```

The above code tells Ruby we have an instance variable named @simple_data, and that we would like to have methods created that will allow us to read and set its value. You can see here how symbols allow us to refer to an instance variable by something that is not a string, but not the literal variable either. With this in place, we can do things like:

```
foo = Simple.new
foo.simple_data = 'abc'
foo.simple_data = [1, 2, 3]
print foo.simple_data.join(', ')
```

Conclusion

Ruby has become extremely popular in the last year or two, in no small part because of the growth of Ruby on Rails among Web developers. Even without Rails though, Ruby deserves much of the attention it has received. The fact that all data is stored in objects, the compactness and elegance of the method and block structures, and the very large number of objects included in the standard library all make for an impressive language.

This article didn't have space to go into some additional features that will be of interest to many Ruby programmers, such as modules, class variables, input/output with files, networking, XML parsing, the RubyGems library available on the Internet and built-in support for regular expressions. Ruby is a rich language, but it is fairly consistent and easy to learn—assuming you already have some background with object-oriented programming, which I think is the greatest hurdle to understanding Ruby.

Ruby still has a number of issues to resolve, including its relatively slow speed and a lack of Unicode support, but these are being addressed for future versions, and the community is one of the strongest that I've seen.

I have been using Ruby more and more during the last year and have grown to be quite impressed with the language. I suggest that you give Ruby a whirl as well. Even if you don't make it your primary programming language, it will get you thinking in new ways, and it might make programming in other languages more enjoyable too.

Resources for this article: http://www.linuxjournal.com/article/9017.