The Ruby Programming Language: A Quick Start Guide*

Jason Brownlee
jasonb@CleverAlgorithms.com
The Clever Algorithms Project
http://www.CleverAlgorithms.com

November 26, 2010
Technical Report: CA-TR-20101126-1

Abstract

The Clever Algorithms project aims to describe a large number of Artificial Intelligence algorithms in a complete, consistent, and centralized manner, to improve their general accessibility. The project makes use of a standardized algorithm description template that uses well-defined topics that motivate the collection of specific and useful information about each algorithm described. This report provides a quick-start guide for the Ruby programming Language, suitable for a programmer proficient in another language to be able to understand the code examples provided in the Clever Algorithms Project.

Keywords: `Ruyb, Programming, Language, Quick, Start, Guide`

1 Introduction

The Clever Algorithms project aims to describe a large number of algorithms from the fields of Computational Intelligence, Biologically Inspired Computation, and Metaheuristics in a complete, consistent and centralized manner [1]. The project requires all algorithms to be described using a standardized template that includes a fixed number of sections, each of which is motivated by the presentation of specific information about the technique [3].

All code examples in the Clever Algorithms project are provided in the Ruby programming language [2]. This report provides a high-level introduction to the Ruby programming language. This guide is intended for programmers of an existing imperative or programming language (such as Python, Java, C, C++, C#) to learn enough Ruby to be able to interpret and modify the code examples provided in the Clever Algorithms project.

The Ruby Programming Language is a multi-paradigm dynamic language that appeared in approximately 1995. It's meta-programming capabilities coupled with concise and readable syntax have made it a popular language of choice for web development, scripting, and application development. The classic reference text for the language is Thomas, Fowler, and Hunt's "*Programming Ruby: The Pragmatic Programmers' Guide*" referred to as the 'pickaxe book' because of the picture of the pickaxe on the cover [5]. An updated edition is available that covers version 1.9 (compared to 1.8 in the cited version) that will work just as well for use as a reference for the examples in this book. Flanagan and Matsumoto's "*The Ruby Programming Language*" also provides a seminal reference text with contributions from Yukihiro Matsumoto, the author of the language [4].

Language Basics

This section summarizes the basics of the language, including variables, flow control, data structures, and functions.

1.1 Ruby Files

Ruby is an interpreted language, meaning that programs are typed as text into a `.rb` file which is parsed and executed at the time the script is run. For example, the following example shows how to invoke the Ruby interpreter on a script in the file `genetic_algorithm.rb` from the command line: `ruby genetic_algorithm.rb`

Ruby scripts are written in ASCII text and are parsed and executed in a linear manner (top to bottom). A script can define functionality (as modules, functions, and classes) and invoke functionality (such as calling a function).

Comments in Ruby are defined by a `#` character, after which the remainder of the line is ignored. The only exception is in strings, where the character can have a special meaning.

The ruby interpreter can be used in an interactive manner by typing out a ruby script directly. This can be useful for testing specific behavior. For example, it is encouraged that you open the ruby interpreter and follow along this guide by typing out the examples. The ruby interpreter can be opened from the command line by typing `irb` and exited again by typing `exit` from within the interpreter.

1.2 Variables

A variable holds a piece of information such as an integer, a scalar, boolean or a string.

```
a = 1 # a holds the integer value '1'
b = 2.2 # b holds the floating point value '2.2'
c = false # c holds the boolean value false
d = "hello, world" # d holds the string value 'hello, world'
```

Ruby has a number of different data types (such as numbers and strings) although it does not enforce the type safety of variables. Instead it uses 'duck typing', where as long as the value of a variable responds appropriately to messages it receives, the interpreter is happy.

Strings can be constructed from static text as well as the values of variables. The following example defines a variable and then defines a string that contains the variable. The `#{}` is a special sequence that informs the interrupter to evaluate the contents of inside the brackets, in this case to evaluate the variable n, which happens to be assigned the value 55.

```
n = 55 # an integer
s = "The number is: #{n}" # => The number is: 55
```

The values of variables can be compared using the == for equality and != for inequality. The following provides an example of testing the equality of two variables and assigning the boolean (true or false) result to a third variable.

```
a = 1
b = 2
c = (a == b) # false
```

Ruby supports the classical `&&` and `||` for AND and or OR, but it also support the `and` and `or` keywords themselves.

```
a = 1
b = 2
c = a==1 and b==2 # true
```

1.3 Flow Control

A script is a sequence of statements that invoke pre-defined functionality. There are structures for manipulating the flow of control within the script such as conditional statements and loops.

Conditional statements can take the traditional forms of `if` *condition then* ¡action¿, with the standard variants of *if-then-else* and *if-then-elseif*. For example:

```
a == 1
b == 2
if(a == b)
  a += 1 # equivalent to a = a + a
elsif a == 1 # brackets around conditions are optional
  a = 1 # this line is executed
else
  a = 0
end
```

Conditional statements can also be added to the end of statements. For example a variable can be assigned a value only if a condition holds, defined all on one line.

```
a = 2
b = 99 if a == 2 # b => 99
```

Loops allow a set of statements to be repeatedly executed `until` a condition is met or `while` a condition is not met

```
a = 0
while a < 10 # condition before the statements
  puts a += 1
end
```

```
b = 10
begin
  puts b -= 1
end until b==0 # condition after the statements
```

As with the if conditions, the loops can be added to the end of statements allowing a loop on a single line.

```
a = 0
puts a += 1 while a<10
```

1.4 Arrays and Hashs

An array is a linear collection of variables and can be defined by creating a new `Array` object.

```
a = [] # define a new array implicitly
a = Array.new # explicilty create a new array
a = Array.new(10) # create a new array with space for 10 items
```

The contents of an array can be accessed by the index of the element.

```
a = [1, 2, 3] # inline declaration and definition of an array
b = a[0] # first element, equivilient to a.first
```

Arrays are also not fixed sized and elements can be added and deleted dynamically.

```
a = [1, 2, 3] # inline declaration and definition of an array
a << 4 # => [1, 2, 3, 4]
a.delete_at(0) # => returns 1, a is now [2, 3, 4]
```

A hash is an associative array, where values can be stored and accessed using a key. A key can be an object (such as a string) or a label.

```ruby
h = {} # empty hash
h = Hash.new

h = {"A"=>1, "B"=>2} # string keys
a = h["A"] # => 1
```

```ruby
h = {:a=>1, :b=>2} # label keys
a = h[:a] # => 1
h[:c] = 3 # add new key-value combination
h[:d] # => nil as there is no value
```

1.5 Functions and Blocks

The `puts` function can be used to write a line to the console.

```ruby
puts("Testing 1, 2, 3") # => Testing 1, 2, 3
puts "Testing 4, 5, 6" # note brackets are not required for the function call
```

Functions allow a program to be partitioned into discrete actions and pre-defined and reusable. The following is an example of a simple function

```ruby
def test_function()
  puts "Test!"
end

puts test_function # => Test!
```

A function can take a list of variables called function arguments.

```ruby
def test_function(a)
  puts "Test: #{a}"
end

puts test_function("me") #=> Test: me
```

And a function can return a variable, called a return value.

```ruby
def square(x)
  return x**2 # note the ** is a power-of operator in Ruby
end

puts square(3) # => 9
```

A block is a collection of statements that can be treated as a single unit. A block can be provided to a function and it can be provided with parameters. A block can be defined using curly brackets {} or the `do` and `end` keywords. Parameters to a block are signified by `|var|`.

The following examples shows an array with a block passed to the constructor of the Array object that accepts a parameter of the current array index being initialized and return's the value with which to initialize the array.

```ruby
b = Array.new(10) {|i| i} # define a new array initialized 0..9

# do...end block
b = Array.new(10) do |i| # => [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
  i * i
end
```

Everything is an object in ruby, even numbers, and as such everything has some behaviors defined. For example, an integer has a `.times` function that can be called that takes a block as a parameter, executing the block the integer number of times.

```ruby
10.times {|i| puts i} # prints 0..9 each on a new line
```

Ruby Idioms

There are standard patterns for performing certain tasks in Ruby, such as assignment and enumerating. This section presents the common Ruby idioms used throughout the code examples in the Clever Algorithms project.

1.6 Assignment

Assignment is the definition of variable (setting a variable to a value). Ruby allows mass assignment, for example, multiple variables can be assigned to respective values on single line.

```
a,b,c = 1,2,3
```

Ruby also has special support for arrays, where variables can be mass-assigned from the values in an array. This can be useful if a function returns an array of return values which are mass assigned to a collation of variables.

```
a, b, c = [1, 2, 3]

def get_min_max(vector)
  return [vector.min, vector.max]
end

v = [1,2,3,4,5]
min, max = get_min_max(v) # => 1, 5
```

1.7 Enumerating

Those collections that are enumerable, such as arrays, provide convent functions for visiting each value in the collection. A very common idiom is the use of the `.each` and `.each_with_index` functions on a collection which accepts a block. These functions are typically used with an in-line block {} so that they fit onto one line.

```
[1,2,3,4,5].each {|v| puts v} # in-line block

# a do...end block
[1,2,3,4,5].each_with_index do |v,i|
  puts "#{i} = #{v}"
end
```

The `sort` function is a very heavily used enumeration function. It returns a copy of the collection that is sorted.

```
a = [3, 2, 4, 1]
a = a.sort # => [1, 2, 3, 4]
```

There are a few versions of the sort function including a version that takes a block. This version of the sort function can be used to sort the variables in the collection using something other than the actual direct values in the array. This is heavily used in code examples to sort arrays of hash maps by a particular key-value pair. The <=> operator is used to compare two values together, returning a -1, 0, or 1 if the first value is smaller, the same, or larger than the second.

```
a = {:quality=>2, :quality=>3, :quality=>1}
a = a.sort {|x,y| x[:quality]<=>y[:quality] } # => [1, 2, 3]
```

1.8 Function Names

Given that everything is an object, executing a function on a object (a behavior) can be thought of as sending a message to that object. For some messages sent to objects, there is a convention to adjust the function name accordingly. For example, functions that ask a question of an object (return a boolean) have a question mark (?) on the end of the function name. Those functions that change the internal state of an object (its data) have an exclamation mark on the end (!). When working with an imperative script (a script without objects) this convention applies to the data provided as function arguments.

```
1  def is_rich? (amount)
2    return (amount >= 1000000)
3  end
4  puts is_rich?(99) # => false
5
6  def square_vector!(vector)
7    vector.each_with_index {|v,i| vector[i] = v**2}
8  end
9  v = [2,2]
10 square_vector!(v)
11 puts v.inspect # => [4,4]
```

1.9 Conclusions

This quick-start guide has only scratched the surface of the Ruby programming language. Please refer to one of the referenced text books on the language for a more detailed introduction into this powerful and fun programming language [5, 4].

References

[1] Jason Brownlee. The clever algorithms project: Overview. Technical Report CA-TR-20100105-1, The Clever Algorithms Project http://www.CleverAlgorithms.com, January 2010.

[2] Jason Brownlee. Programming language selection for optimization algorithms. Technical Report CA-TR-20100122-1, The Clever Algorithms Project http://www.CleverAlgorithms.com, January 2010.

[3] Jason Brownlee. A template for standardized algorithm descriptions. Technical Report CA-TR-20100107-1, The Clever Algorithms Project http://www.CleverAlgorithms.com, January 2010.

[4] David Flanagan and Yukihiro Matsumoto. *The Ruby Programming Language*. O'Reilly Media, 2008.

[5] Dave Thomas, Chad Fowler, and Andy Hunt. *Programming Ruby: The Pragmatic Programmers' Guide*. Pragmatic Bookshelf, second edition, 2004.