LUKE KANIES

why you should use Ruby

Luke Kanies runs Reductive Labs (http://reductivelabs.com), a startup producing OSS software for centralized, automated server administration. He has been a UNIX sysadmin for nine years and has published multiple articles on UNIX tools and best practices.

luke@madstop.com

ABOUT TWO YEARS AGO, I SWITCHED from doing all of my development in Perl [1] to using Ruby [2] for everything. I was an independent consultant at the time, writing a prototype for a new tool (now available and called Puppet [3]); I had been writing tons of Perl since about 1998, and until I tried to write this prototype I really thought Perl was the perfect solution to my programming needs—I could do OO, I could do quick one-offs, and it was available everywhere. It seemed I could also think pretty well in Perl, so it didn't take long to translate what I wanted into functional code.

That all changed when I tried to write this prototype. I tried to do it in Perl, but I just couldn't turn my idea into code; I don't know why, but I couldn't make the translation. I tried it in Python, because I'd heard lots about how great Python was, but for some reason Python always makes my eyes bleed. So, in desperation, I took a stab at Ruby. Prior to this work, I had never seen a line of Ruby and had never heard anything concrete about it or why one would use it, but in four short hours I had a functional prototype; I felt as though a veil had been drawn from my eyes, that I had previously been working much harder than necessary.

I couldn't have told you then what it was about Ruby exactly, but there was something that clearly just seemed to make it easier to write code, even code that did complicated things. Since then, I've gotten much better at Ruby and a good bit more cognizant of what sets it apart.

You could argue that mine was a personal experience and that most people would not benefit as much from a switch to Ruby, and in some ways you would be right—I wrote a lot of OO code in Perl, which isn't exactly pleasant, and I generally hate it when I have to do work that the computer can do (you almost never have to actually use a semicolon to end a line in Ruby). But the goal of this article is to convince you that just about anyone would find benefit from a switch to Ruby, especially if you use or write many libraries or if you use your scripting language as your primary interface to your network.

This article is definitely not meant to teach you how to write Ruby; the Pragmatic Programmers [4] have written a great book [5] on Ruby, and I

highly recommend it. I also recommend reading Paul Graham's essays on programming language power [6] and beating the averages [7]; they do a good job of discussing what to think about in language choice.

All of the following examples were written as simple Ruby scripts in a separate file (I keep a "test.rb" file or equivalent for every interpreted language I write in, and that's what I used for all of these examples), and the output of each is presented prefixed with =>. To run the examples yourself, just put the code into something like example.rb and run ruby example.rb. You could also use the separate irb executable to run these examples, but the output will be slightly different.

What's So Special About Ruby?

I can't point to one hard thing that makes Ruby great—if I tried, I would just end up saying something silly like, "It just works the way I expect," and that wouldn't be very useful. Instead, I'll run through some of the things that I love and that really change how I use it.

EVERYTHING IS AN OBJECT

Yes, Marjorie, everything. No, there are no exceptions (heh, rather, even the Exceptions are objects); there aren't special cases. Classes are objects, strings are objects, numbers are objects:

```
[Class, "a string", 15, File.open("/etc/passwd")].each { |obj|
        puts "'%s' is of type %s" % [obj, obj.class]
}
=> 'Class' is of type Class
=> 'a string' is of type String
=> '15' is of type Fixnum
=> '#<File:0x210310>' is of type File
```

Here we have a list of objects and we print a string describing each object in turn (puts just prints the string with a carriage return at the end). When you create a new class, it's an instance of the Class object. This each syntax is how pretty much all iteration is done in Ruby—you can use for loops, but $10 says you won't once you get used to each.

THERE ARE NO OPERATORS

Did you notice that % method in the example above? Yeah, that's a method, not an operator. What's the difference? Well, the parser defines an operator, but the object's class defines a method. In Perl, you have one operator for adding strings, . (a period), and one operator for adding numbers, +, because those operators are part of the language and the parser can't easily type-check the arguments to verify that you passed the right arguments all around. But in Ruby, each class just implements a + method that behaves correctly, including any type-checking.

It gets better. The indexing syntax for hashes and arrays is also a method, and you can define your own versions:

```
class Yayness
        def initialize(hash)
                @params = hash

        end
```

```
        def [](name)
                @params[name]
        end

        def []=(name, value)
                @params[name] = value
        end

end

y = Yayness.new(:param => "value", :foo => "bar")
puts y[:foo]
y[:funtest] = "a string"
puts y[:funtest]

=> bar

=> a string
```

That funny term with the colons is called a "Symbol," and it's basically a simple constant, like an immutable string. It's very useful for those cases where you would normally use an unchanging string, such as for hash keys, but you're too lazy to actually type two quotation marks. Actually, one of the reasons I like them so much is that Vim colorizes them quite differently from strings, making them easier to read.

Why would you use these indexing methods? A large number of my classes have a collection of parameters, and this makes it trivial to provide direct access to those parameters. Sometimes it makes sense to subclass the Hash class, but there are plenty of other times where you want to wrap one or more hashes, and these methods make that very easy. I also often define these methods on the classes themselves, so that I can retrieve instances by name:

```
class Yayness
        attr_accessor :name
        @instances = {}
        def Yayness.[](instname)
                @instances[instname]

        end

        def Yayness.[]=(instname, object)
                @instances[instname] = object

        end

        def initialize(myname)
                self.class[myname] = self
                @name = myname

        end

end

first = Yayness.new(:first)
second = Yayness.new(:second)

puts Yayness[:first].name
=> first
```

Here I've used attr_accessor (which is a method on the Module class, and thus available in all class definitions) to define getter/setter methods for name, and then some methods for storing and retrieving instances by name. I also use this frequently, possibly even too frequently. In Ruby, instance variables are denoted by prefixing them with an @ sigil; so, in this

case, calling the name method on a Yayness object will return the value of @name.

The initialize method, by the way, is kind of like a constructor in Ruby—I say "kind of" because it's actually called after the object exists and is only expected to (of course) initialize the object, not create it.

INTROSPECTION

Ruby is insanely introspective. We've already seen how you can ask any object what type of object it is (using the class method), and there are a bunch of complementary methods for asking things like whether an object is an instance of a given class, but you can also ask what methods are available for an object, or even how many arguments a given method expects:

```
class Funtest
        def foo(one, two)
                puts "Got %s and %s" % [one, two]
        end

end

class Yaytest
        def foo(one)
                puts "Only got %s" % one
        end

end

[Funtest.new, Yaytest.new, "a string"].each { |obj|
        if obj.respond_to? :foo
                if obj.method(:foo).arity == 1
                        obj.foo("one argument")
                else
                        obj.foo("first argument", "second argument")
                end
        else
                puts "'%s' does not respond to :foo" % obj
        end
}

=> Got first argument and second argument
=> Only got one argument
=> 'a string' does not respond to :foo
```

Here we create two classes, each with a foo method but each accepting a different number of arguments. We then iterate over a list containing an instance of each of those classes, plus a string (which does not respond to the foo method); if the object responds to the method we're looking for, we retrieve the method (yes, we get an actual Method object) and ask that method how many arguments it expects (called its arity).

You can see here that I'm using a Symbol for the method name during the introspection; this is common practice in the Ruby world, even though I could have used a string.

ITERATION

You've already seen the each method on arrays (it also works on hashes), and you probably thought, "Oh, well, my language has that and it's called

'map,' " or something similar. Well, Ruby goes a bit further. Ruby attempts to duck the multiple inheritance problem by supporting only single inheritance but allowing you to mix in Modules. I'll leave it to the documentation to cover all of the details, but Ruby ships with a few modules that are especially useful, and the Enumerable module is at the top of the list:

```
class Funtest
        include Enumerable
        def each
                @params.each { |key, value|
                        yield key, value
                }
        end
        def initialize(hash)
                @params = hash
        end
end
f = Funtest.new(:first => "foo", :second => 59, :third => :symbol)
f.each { |key, value|
        puts "Value %s is %s of type %s" % [key, value, value.class]
}
puts f.find { |key, value| value.is_a?(Symbol) }.join(" => ")
puts f.collect { |key, value| value.to_s }.join("---")
```

```
=> Value second is 59 of type Fixnum
=> Value first is foo of type String
=> Value third is symbol of type Symbol
=> third => symbol
=> 59---foo---symbol
```

Here we define a simple class that accepts a hash as an argument and then an each method that yields each key/value pair in turn (you'll have to hit the docs for more info on how yield works—it took me a while to understand it, but it was worth it). By itself our class isn't so useful, but when we include the Enumerable module, we get a bunch of other methods for free. I've shown two useful methods: find (which finds the first key/value pair for which the test is true, and returns the pair as an array) and collect (which collects the output of the iterative code and returns it as a new array).

You can see that our each code did exactly as we expected, but we also easily found the first Symbol in the list (find_all will return an array containing all matching elements). In addition, we used collect to create an array of strings (just about every object in Ruby accepts the to_s method to convert it to a string, although you generally have to define the method yourself on your own classes for it to be meaningful).

The great thing here is that we just defined one simple method and got a bunch of other powerful iterative methods. Another useful module is Comparable; if you define the comparison method <=> and include this module, then you get a bunch of other comparison methods for free (e.g., >, <=, and ==).

BLOCKS

On the one hand, I feel as though I should talk about blocks, because they really are one of the most powerful parts of Ruby; on the other hand, I know that I am not up to adequately explaining in such a short article how

they work and why you'd use them. I'm going to take a shot at such an explanation, but I fear that I'll only confuse you; please blame any confusion on me, and not on Ruby. As you use Ruby, you'll naturally invest more in using and understanding blocks, but you can survive in Ruby just fine without worrying about them. It's worth noting, though, that the iteration examples above all use blocks—each, find, etc., are all called with blocks.

Blocks just keep looking more powerful the more I use them. They're relatively simple in concept, and many languages have something somewhat similar—they're just anonymous subroutines, really—but the way Ruby uses them goes far beyond what I've seen in most places. As a simple example, many objects accept blocks as an argument and will behave differently—files will automatically close at the end of a block, for instance—if a block is provided:

```
File.open("/etc/passwd") { |f|
        puts f.read
}
```

Once you get used to blocks automatically cleaning up after you, it becomes quite addictive. I often find myself creating simple methods that do some setup, execute the block, and then clean up. For instance, here's a simple method for executing code as a different user (the method is significantly simplified from what I actually use):

```
def asuser(name)
        require 'etc'
        uid = Etc.getpwnam(nam).uid
        Process.euid = uid
        yield
        Process.euid = Process.uid
end

asuser("luke") {
        File.unlink("/home/luke/.rhosts")
}
```

We convert the name to a number using the Etc module (which is basically just an interface to POSIX methods), and then we set the effective user ID to the specified user's. We use yield to give control back to the calling code (which just executes the associated block), and then reset the EUID to the normal UID.

This is a very simple example; there are a huge number of methods in Ruby that accept blocks, and there are many ways of using them to make your life easier. I recently refactored the whole structure of Puppet around using blocks where I hadn't previously, and the result was a huge increase in clarity and, thus, productivity. Here's a hint: if you find yourself dynamically creating modules or classes, note that you can use a block when doing so:

```
myclass = Class.new {
        def foo
                puts "called foo"
        end
}
a = myclass.new
a.foo
=> called foo
```

Here I'm defining a class at runtime, rather than at compile-time, and using a block to define a method on that class.

This specific example is no different from just using the class keyword to define the class, but at least for me it provided much more power and flexibility in how I created new classes. This ends up being critical in Puppet, which is composed almost entirely of classes containing classes; the relationships between those classes is one of the most complicated parts of the code—making that easier had a huge payoff.

Conclusion

I hope I've at least interested you in learning more about Ruby. I know that I was short on my descriptions, but I've tried to focus more on why you'd use it than how. I highly recommend looking into some of the discussion around Ruby on Rails [8]; a lot of Java refugees have taken up Ruby as a means of getting more done with less effort, and their discussions on why are very informative.

Even if you don't use Ruby, though, learn more languages, assess them critically, and demand more from them. Your computer should be working for you, and the language you choose for interacting with your computer determines a lot about how you work.

REFERENCES

[1] http://Perl.org

[2] http://ruby-lang.org

[3] http://reductivelabs.com/projects/puppet

[4] http://www.pragmaticprogrammer.com/index.html

[5] http://www.pragmaticprogrammer.com/titles/ruby/index.html

[6] http://paulgraham.com/power.html

[7] http://paulgraham.com/avg.html

[8] http://rubyonrails.org