The Perl Review

Volume 0 Issue 6 November 1, 2002

Like this issue? Support *The Perl Review* with a donation! http://www.ThePerlReview.com/

Web Access http://www.ThePerlReview.com/ **Email** letters@theperlreview.com **Publisher** brian d foy **Editor** Andy Lester **Technical Editors** Kurt Starsinic, Adam Turoff **Copy Editors** Beth Linker, Glenn Maciag, Chris Nandor **Contributors** David H. Adler, Paul Barry, Neil Bauman, brian d foy, Andy Lester, Mike Stok, Betsy Waliszewski

Delightful Languages: Ruby

Mike Stok, mike@stok.co.uk

Abstract

This is a brief recounting of my initial impressions of and experience with the Ruby programming language and its community. In many ways Ruby strikes the same chord in me that Perl did a decade or more ago. I show Ruby from a Perl perspective.

1 Introduction to Ruby

When I first encountered Perl, I found the language to be a little strange coming from a C background. From time to time I would use Perl to write things I would have written in C or shell, and soon Perl was my tool of choice for many tasks.

Ruby is having a similar effect on me. Sometimes I prototype Perl code in Ruby, sometimes I just use Ruby for the sake of seeing if I arrive at a different solution using a different language. Like Perl, Ruby makes programming fun, but in a different way.

2 Rewriting Soundex

Ruby borrows features from many languages, and one of those is Perl. I can simply translate Perl code into Ruby if I want. I will use the Soundex function as an example, as the algorithm is simple and my Perl implementation will reveal something about my abilities as a programmer.

The Soundex algorithm is a simple hashing of the letters of a word to a four character code which brings similar sounding words to the same code. In 1994 I posted a routine, shown in code listing 1, to comp.lang.perl which shows both the simplicity of the Soundex algorithm and my Perl style at its worst (or best).

Code Listing 1: Original Perl soundex function

```
sub Soundex
{
  local ($_, $f) = shift;

  y;a-z;A-Z;;y;A-Z;;cd;$_ ne q???do{($f)=m;^(.);;s;$f+;;;
  y;AEHIOUWYBFPVCGJKQSXZDTLMNR;00000000111122222222334556;;
  y;;;cs;y;0;;d;s;^;$f;;s;$;0000;;m;(^.{4});;$1;}:q;;;
}
```

This code survives, with a bug fix and some reformatting as the Text::Soundex module in Perl 5.8.0[1], shown in code listing 2. The major difference between the routines is that the newer code makes use of Perl's subroutine call context to decide whether to return a single scalar or a list of scalars. This is a legacy from Perl 4 days when Perl did not have map. These days I would leave it to the routine's user to do the work even if it means a few extra subroutine calls.

Code Listing 2: Current Perl soundex function

```
sub soundex
{
  local (@s, $f, $fc, $_) = @_;

  push @s, '' unless @s;        # handle no args as a single empty string

  foreach (@s)
  {
        $_ = uc $_;
        tr/A-Z//cd;

        if ($_ eq '')
        {
          $_ = $soundex_nocode;
        }
        else
        {
          ($f) = /^(.)/;
          tr/AEHIOUWYBFPVCGJKQSXZDTLMNR/00000000111122222222334556/;
          ($fc) = /^(.)/;
          s/^$fc+//;
          tr///cs;
          tr/0//d;
          $_ = $f . $_ . '000';
          s/^(.{4}).*/$1/;
        }
  }

  wantarray ? @s : shift @s;
}
```

2.1 Ruby Soundex code

I did a simple translation of the Perl code into Ruby[2], shown in code listing 3. As the wantarray is a Perlish idiom I left it out.

[1] Serious Text::Soundex users should grab Mark Mielke's faster version from CPAN. I do not know why the current version has survived so long in the official distribution.

[2] There is already a Soundex module in the Ruby Application Archive. Its author, Michael Neumann, took a more conventional approach to its implementation than mine.

Code Listing 3: Soundex in Ruby

```
def soundex(string, nocode=nil)
        copy = string.upcase.tr '^A-Z', ''
        return nocode if copy.empty?
        first_letter = copy[0, 1]
        copy.tr_s! 'AEHIOUWYBFPVCGJKQSXZDTLMNR',
                   '00000000111122222222334556'
        copy.sub!(/^(.)\1*/, '').gsub!(/0/, '')
        "#{first_letter}#{copy}000"[0 .. 3]
end
```

2.1.1 Perl and Ruby differences

No Semicolons

I did not use any semicolons in code listing 3. Ruby can use semicolons to separate expressions, but most Ruby code uses line ends to indicate the end of an expression or statement. When a line ends in the middle of an expression, Ruby realizes that it will continue on the next line. For example, I can split `x = 2 + 2` over two lines.

```
x = 2 +
    2
```

Ruby sees a single expression as it expects something after the `+`.

If I wanted to put multiple statements on a line then I could use semicolons.

```
x = 2 + 2; y = x + 1
```

No Sigils

Ruby does not use sigils (the leading `$`, `@`, `%` character used in Perl) to indicate the type of a variable. When retrieving an element from a Hash or an Array, the same operator, `[]`, is used. The type of the container I am accessing determines what I should put inside the brackets.

Ruby does use `$`, `@`, and `@@` as prefixes, but I do not cover that in this article.

Named Parameters

Ruby's method definitions let me name parameters and specify default values for optional parameters. The string parameter is required, and the nocode parameter is optional. If I do not specify nocode, then Ruby gives it the value nil.

Ruby checks the arity when I call a method to make sure that I required all parameters. If I try to call soundex with the wrong number of arguments then Ruby raises an ArgumentError.

```
soundex()           # => ArgumentError: wrong number of arguments(0 for 1)
```

No declarations

There are no equivalents of my or local. If I use a variable name then Ruby checks if there is already a variable of that name in scope; if there is, Ruby reuses it, otherwise it creates a variable in the current scope.

String Interpolation Using #{ ... }

Inside double quotes Ruby uses `#{ ... }` to interpolate any expression. I used double quotes to concatenate first_character, copy, and the literal `000`, but I can put any expression between the braces and Ruby interpolates the result into the string being generated by the double quotes.

Methods everywhere

All the subroutine calls are methods. Most of the methods, upcase, tr, length, are members of the String class. As in Perl a method is a subroutine called in the context of an object.

Predicate method `empty?`

The `empty?` method ends with a question mark which is part of the method name. In Ruby this conventionally means that the method is a predicate. This convention is not enforced by the interpreter.

In Perl I can say `$string or return $soundex_nocode;` if I know the string cannot contain a solitary 0. Perl's notion of truth means that the empty string or a string containing 0 are considered false. In Ruby the only false values are nil and false, so I have to test string.

I could have used `string.length == 0`, but the name of the `empty?` method expresses my intent more clearly.

Method chaining

`string.upcase.tr '^A-Z', ''` shows how I can chain methods in Ruby. The upcase method takes the value referenced by string, converts it to upper case, then the tr method deletes the non-alphabetic characters. The tr and upcase methods do not affect the value string refers to. Ruby's convention is that a destructive method name ends with a `!`. This is only a convention, so the Ruby interpreter does not enforce it.

Copy data in String

As all data are objects in Ruby, string is a reference to an object. I was careful not to change the value of the original string passed to the subroutine.

When I operate on the data referenced by copy, many of the methods end with a `!`. The tr! method modifies the object it is called on. The tr method always gives you a new string as a result, but tr! returns the modified, original string or nil.

Getting the First Character of a String

Ruby strings are sequences of integers, not arrays of characters.

The first time I tried to get a character from a string in Ruby, I used something like `char = string[0]`. This gave me the ASCII value of the first character of the string. Consulting the section on the String class in *Programming Ruby*, I discovered that I need to use either a Range or a couple of numbers in the brackets to get a substring.

2.2 A Brief Diversion - irb

I use the irb program which comes with Ruby as a test bed for Ruby code, much as I use `perl -de 1` for Perl. irb prints the result of each expression it evaluates, as in code listing 4. The `irb(main):001:0>` is the irb prompt. When I enter an expression, the result irb prints before the next prompt.

Code Listing 4: irb session

```
[mike@ratdog tmp]$ irb
irb(main):001:0> string = "Mike"
"Mike"
irb(main):002:0> string.class
String
irb(main):003:0> string[0]
77
irb(main):004:0> string[0].class
Fixnum
irb(main):005:0> string[0,1].class
String
irb(main):006:0> string[0,1]
"M"
```

2.3 Making soundex More Rubyesque

The Ruby soundex code presented in code listing 3 is fragile. When I use it on strings, all is well, but should I accidentally use it on a variable containing a number, as in code listing 5, something else happens. Perl automatically morphs the contents of scalars between numbers and strings, but Ruby expects you to be explicit about this. If an object does not does not respond to a method, Ruby complains.

Code Listing 5: Passing Ruby's soundex a number

```
irb(main):018:0> soundex(2)
NoMethodError: undefined method `upcase' for 2:Fixnum
                 from (irb):8:in `soundex'
                 from (irb):18
                 from :0
```

The things I can do to mitigate this include:

Argument Checking

I can test the class of the argument using the class method available to all Ruby objects.

Turn the Argument into a String

I can use the to_s method which turns Ruby objects into Strings.

Make soundex a String Method

I can put soundex into the String class and make it look more like a builtin.

I prefer to make soundex act more like a builtin String method. Ruby allows me to add methods to classes at any time. I can add soundex to the String class so that it is available to Strings and classes derived from them for the duration of the program.

When I use soundex as a String method call, the thing being encoded is available through a variable self or I can use implicitly as the default object. In the code listing 6 there is an implied self in the `copy = upcase.tr '^A-Z', ''` which I could write as the equivalent `copy = self.upcase.tr '^A-Z', ''`.

Code Listing 6: Adding soundex to String

```
class String
        def soundex(nocode=nil)
                copy = upcase.tr '^A-Z', ''
                return nocode if copy.empty?
                first_letter = copy[0, 1]
                copy.tr_s! 'AEHIOUWYBFPVCGJKQSXZDTLMNR',
                           '00000000111122222222334556'
                copy.sub!(/^(.)\1*/, '').gsub!(/0/, '')
                "#{first_letter}#{copy}000"[0 .. 3]
        end
end
```

I saved code listing 6 in soundex.rb, and running irb in the same directory as soundex.rb I could make soundex available to all Strings and objects in subclasses of String. In code listing 7, I ran an irb session to try it. The soundex method is not available until I load the file containing String#soundex, soundex.rb, with require.

Code Listing 7: Importing the soundex method

```
irb(main):001:0> str = "Mike"
"Mike"
irb(main):002:0> str.soundex
NoMethodError: undefined method `soundex' for "Mike":String
                from (irb):2
irb(main):003:0> require 'soundex'
true
irb(main):004:0> str.soundex
"M200"
```

In code listing 8 I create a Surname subclass of String and check that soundex is available. The < in the class line means that Surname is a subclass of String. In the last couple of lines I show that surname really is a Surname object, and that I can tell that surname's class is derived from String. In Perl I would use something like `$s->isa('String')` to do this.

Code Listing 8: Creating the Surname subclass

```
irb(main):005:0> class Surname < String
irb(main):006:1> end
nil
irb(main):007:0> surname = Surname.new('Stok')
"Stok"
irb(main):008:0> surname.soundex
"S320"
irb(main):009:0> surname.class
Surname
irb(main):010:0> surname.kind_of? String
true
```

2.4 Testing the Code

The soundex code seems to work, but I do not feel happy until I have a basic set of tests I can run to make sure that innocuous changes do not break things.

Ruby's equivalent to CPAN is the Ruby Application Archive (RAA). I use Nathaniel Talbott's Test::Unit package which makes writing and running test cases a breeze.

To use Test::Unit, I just have to put all my test cases in a class derived from Test::Unit::TestCase, and Test::Unit will find all the methods and run named test_* in that class. Test::Unit has a number of assertion methods I can use, but my test cases are so simple that they only use a few.

Ruby has the __FILE__ token, so I can add a line to my module to see if the module (soundex.rb) is the program being run or whether it is being included from another file. If it is being included Ruby should not run the tests, and if I run the file directly, it should run the tests.

To add unit tests to the code presented above I add them to the bottom of the file, as in code listing 9.

Code Listing 9: Testing soundex

```
if __FILE__ == $0
        require 'test/unit'

        class TC_Soundex < Test::Unit::TestCase
                def test_knuth
                        [ %w(Euler       Ellery    E460),
                          %w(Gauss       Ghosh     G200),
                          %w(Hilbert     Heilbron  H416),
                          %w(Knuth       Kant      K530),
                          %w(Lloyd       Ladd      L300),
                          %w(Lukasiewicz Lissajous L222),
                        ].each do |test|
                                assert_equal(test[0].soundex, test[-1])
                                assert_equal(test[0].soundex, test[1].soundex)
                        end
                end

                def test_empty
                        assert_nil(''.soundex)
                end

                def test_non_alpha
                        assert_nil('2+2=4'.soundex)
                end

                def test_mike
                        assert_equal('Mike'.soundex, 'M200')
                        assert_equal('Stok'.soundex, 'S320')
                end

                def test_czarkowska
                        # in the old perl version this was a bug which caused a
                        # discrepancy between Oracle's soundex and mine.  Spotted
                        # by Rich Pinder

```

```
36                      assert_equal('CZARKOWSKA'.soundex, 'C622')
37              end
38
39              def test_nocode
40                      assert_equal(''.soundex('?'), '?')
41              end
42      end
43  end
```

I run the tests by executing the module directly.

```
[mike@ratdog ruby-soundex]$ ruby soundex.rb
Loaded suite soundex
Started
.......
Finished in 0.041179 seconds.
6 tests, 18 assertions, 0 failures, 0 errors
```

2.5 Packaging the Code

When I get a Perl module from CPAN, I know that I can install it with the familiar `perl Makefile.PL`, `make`, `make test` and `make install` ritual.

The Ruby module world has not yet settled into a predictable pattern for module installation. If I want to use code from a library file, then the file has to be in one of the directories mentioned in the `$:` variable which is like Perl's @INC (Ruby does not have built-in formats).

3 Impressions of Ruby

I really like Ruby. I thought I would like it when I saw it first more than a year ago, and I still like it. I still use Perl, and I think that Ruby has improved my Perl.

I find Ruby's clean class and method definitions make me much more inclined to make new classes for different types of object in my code. I can use various helper modules like base in Perl, but I prefer the look of Ruby.

I try to adopt the Extreme Programming "test first" strategy when developing Ruby code. This means that I am less likely to get caught out by the lack of explicit scoping commands. I like the control that my gives me in Perl. While giving me more control, my does clutter up the code and lets me get away with long rambling subroutines. Ruby encourages me to write small routines and think about minimizing the scope of variables.

The abundance of unit tests provided with most of the code on the Ruby Application Archive makes me much more comfortable when I am trying to fix flaws in code. When I needed to fix a couple of problems in Sean Russell's amazing Ruby Electric XML (REXML) package, the unit tests allowed me to see how much damage I was doing.

I have avoided going over areas which are well covered in other articles which introduce Ruby. I found the *Doctor Dobb's* article by Dave Thomas and Andy Hunt a good introduction to Ruby's features. I found that Ruby has enough features which are different enough from those in Perl that it made me think differently about problems; if I had to pick one to begin with then it would be the yield method.

3.1 A yield Digression

I can use yield to build generators and iterators. I can do this in Perl and Python too, but Ruby's libraries make so much use of iterators that they do not seem at all unusual.

The Ruby Enumerable module class uses yield in the `select` which selects items from a collection by calling a user specified block of code on each element of the collection. This is much like Perl's `grep BLOCK LIST`, where grep calls BLOCK on each element of LIST. I use this to select all the elements smaller than a pivot from an array in Ruby.

```
lesser  = list.select { |e| e <  pivot }
```

The block is called on each element of C¡list¿, if the element is less than the pivot then that element is included in the list which is returned. In Perl I perform the selection using this code:

```
my @lesser = grep { $_ < $pivot} @list;
```

In Perl the code block comes between grep and LIST, and in the Ruby the block comes at the end. I prefer the Ruby style, especially if I need to split the code across multiple lines.

When I want to allow the user to pass a block of code to a method I just call yield with the argments I want to pass into the block. Python recently acquired yield, so the simple Ruby transliteration of Paul Prescod's fib_gen2 routine from *The Perl Review (0,2)* might look like code listing 10. The fib_gen2 method yields control to a user supplied block of code each time around the count.times loop.

Code Listing 10: Ruby fibonacci number generator

```
1  def fib_gen2(count)
2          this_number, next_number = 1, 1
3          count.times do
4                  yield this_number, next_number
5                  this_number, next_number = next_number, this_number + next_number
6          end
7  end
8
9  fib_gen2(100) { |n1, n2|
10         puts "#{n1} #{n2}"
11 }
```

4 Ruby Resources

The Ruby community is active and varied. There are many local Ruby User Groups, and there is an annual Ruby Conference.

One of the things which attracted me to Ruby was its community. The spirit reminds me of the Perl community in the early 1990s.

There are many Ruby resources on the net. Over the past couple of years Ruby resources have started to be available in more languages than Japanese and English. There are links to the entire text of *Programming Ruby.*

The comp.lang.ruby Newsgroup

Ruby's author and many active contributors participate in the news group. I find the traffic moderate and the signal-to-noise ratio high.

#ruby-lang on irc.openprojects.net

There is a Ruby IRC channel. I don't use IRC.

Ruby Language and Ruby Garden Web Sites

The main Ruby language web site is at http://www.ruby-lang.org . This site contains pretty much everything you need to explore Ruby and get started. It has links to other introductory articles.

The community web site for Ruby is http://www.rubygarden.com . This has all kinds of useful resources including a Ruby wiki and a summary of the past week's activity on the newsgroup.

Ruby Application Archive (RAA)

The Ruby Application Archive is Ruby's equivalent to CPAN. There is a link to it on the Ruby language web site mentioned above.

If I wanted to use Perl-style formats then a search of the RAA would get me to Paul Rubel's FormatR package (billed as "just like Perl's plus some and the ability to go in reverse.")

Ruby Books

The most popular English language book about Ruby seems to be *Programming Ruby* by David Thomas and Andrew Hunt. This book had the same effect on my appreciation of Ruby as the first edition of *Programming Perl* had on my interest in Perl. My copy has become quite dog-eared—always a good sign that I find a book useful.

Programming Ruby is available on-line at http://www.rubycentral.com/book/index.html . This is a perfect complement to the print version, and the introduction to Ruby is a good read.

Hal Fulton's *The Ruby Way* is an excellent source of all kinds of techniques you can use in Ruby. It is task-oriented and includes an appendix on coming to Ruby from Perl.

5 Conclusion

I like Ruby because it seems to have struck the right balance between simplicity and power. The language is different enough from Perl to make me think about things in a fresh light. The Ruby community is active and helpful. The texture of Ruby encourages me to write code which seems to make sense months later.

I feel like the time I have spent with Ruby has been as rewarding as the time I spent with Perl. Admittedly the Ruby Application Archive is not as big as CPAN, and some of Ruby's features might appear in Perl 6, but I still think it is worth giving Ruby a try.

6 References

The Ruby Language - http://www.ruby-lang.org/

"Programming in Ruby", Dave Thomas and Andy Hunt, Dr. Dobbs, January 2001,http://www.ddj.com/documents/s=871/ddj0101b/0101b.htm

http://www.germane-software.com/software/rexml/ - the REXML module. The tutorial for this package might give you a better feel for Ruby's character.

One way of insinuating Ruby code into a Perl world is the devious use of Brian Ingerson's Inline::Ruby module from CPAN.

7 About the Author

I work for Exegenix in Toronto, writing Perl to transform documents into other documents. I enjoy life with my wife and daughter, Perl and Ruby coding, hot air ballooning, beer and chocolate (even white chocolate, as long as it's a Toblerone).