1 Paul's Ruby Regex Primer

Welcome to *Paul's Ruby Regex Primer*[1]. Work through this to gain a thorough grounding in, and introduction to, the regular expression technology built into Ruby. The assumption is that you have ready access to `irb` and to a command-line Ruby.

2 Credit When Credit is Due

The material in this document is based on very similar material from Chapter 7 of my second book: *Bioinformatics, Biocomputing and Perl*, as published by Wiley in 2004. There's also some code and examples "borrowed" from the second edition of *The PickAxe*.

3 Regular Expressions and Ruby

In addition to Ruby, many other programs and tools utilise regular expressions in interesting ways. One such tool is `grep`, the "generalized regular expression parser", which can be used to search for a pattern within any selection of disk-files. Read the manual page for `grep` to learn how to use it, then use `grep` to search for the existence of arbitrary sequences in the your disk-files. [Be advised that upon success and by default, `grep` prints the matching line to `STDOUT`. Check the options, documented in the manual page, to learn how to change this default behaviour.]

3.1 What is a regular expression?

A regular expression is, first and foremost, a *pattern*. The pattern tells `ruby` to look for something, and this "something" can be any sequence of characters. This pattern looks for the word "even":

```
/even/
```

Typically, the pattern that makes up a regular expression is enclosed within two forward-leaning slash characters, as is the case above. It is also possible to enclose Ruby patterns within single quotes when creating a regular expression object with `Regexp.new`. Alternative delimiters can also be used (more on this later).

It is important to realize that the pattern is just a sequence of characters to `ruby`. Even though "even" is a word (for us), it is four individual characters (for `ruby`). Specifically, the pattern `/even/` looks for the character "`e`", followed by the character "`v`", followed by the character "`e`", followed by the character "`n`". When the pattern is compared against something (such as an input stream or string), it is said to *match* if this sequence of four characters appears. Here are some successful matches to the `/even/` pattern:

[1] Version 1.01, September 2006

```
eleven                # matches at end of word
eventually            # matches at start of word
even Stevens          # matches twice: an entire word and within a word
```

And here are some unsuccessful matches (or non-matches):

```
heaven                # 'a' breaks the pattern
Even                  # uppercase 'E' breaks the pattern
EVEN                  # all uppercase breaks the pattern
eveN                  # uppercase 'N' breaks the pattern
leave                 # not even close!
Steve not here        # space between 'Steve' and 'not' breaks the pattern
```

Most regular expression technologies (and Ruby's is no exception) are extended by a collection of *special* characters, referred as a *metacharacters*. Metacharacters influence how the pattern is matched, and are described later in this document.

> **Technical Commentary**: The term "regular expression" is often shortened to "regex", and is pronounced "reg", as in "beg", and "ex" as in ... well, "x".

3.2 What makes regular expressions so special?

Let's answer this question with a demonstration. Imagine the requirement to write a subroutine to find the first occurrence of a pattern, such as "even", within a given string. A reasonable strategy is to approach the problem in the following way (note that *before* any processing occurs, `ruby` starts from the beginning of the string to search and has yet to read any characters from it):

1. Examine the next character of the string.
2. If the character under consideration is *not* "e", return to step 1.
3. If the character under consideration is "e", consider the next character of the string.
4. If the character under consideration is *not* "v", go back one character (that is, back to the found "e"), and return to step 1.
5. If the character under consideration is "v", consider the next character of the string.
6. If the character under consideration is *not* "e", go back two characters (that is, back to the first found "e"), and return to step 1.
7. If the character under consideration is "e", consider the next character of the string.
8. If the character under consideration is *not* "n", go back three characters (that is, back to the first found "e"), and return to step 1.

9. If the character under consideration is "n" - rejoice! - a match has been found.

Using a pencil and some paper, use this strategy to search for the pattern "even" in the strings "Steven", "heaven" and "eleven", convincing yourself that it does indeed work[2].

Now, imagine further that a subroutine, which is based on the above strategy and called `find_it`, searches a given string for a given pattern, returning "true" upon success. The subroutine could be invoked like this:

```
pattern = "even"
string = "do the words heaven and eleven match?"

if find_it( pattern, string )
    puts "A match was found."
else
    puts "No match was found."
end # of if.
```

Assuming, of course, that the subroutine did indeed exist (which is does not). The reason it does not exist is that no Ruby programmer, even the most masochistic, would ever dream of creating a subroutine like `find_it`. Writing such a subroutine is tedious, tricky and totally unnecessary. The Ruby programmer uses a regular expression, and writes the above code like this:

```
string = "do the words heaven and eleven match?"

if string =~ /even/
    puts "A match was found."
else
    puts "No match was found."
end # of if.
```

And then the Ruby programmer promptly gets on with whatever else needs doing! The requirement to write a subroutine to perform the searching is nullified. After running the above code through `irb`, try the following commands:

```
puts $`
puts $"
puts $&
```

The key point of all of this is that by using a regular expression, Ruby programmers are able to specify what it is they are interested in finding, *without* having to spell out how it should be found. The "how" is left to `ruby`, which performs the search based on the specified regular expression. So, you use a regular expression to specify what you want to find, not how to find it.

At first glance, many think that this is not such an important thing. However, finding things in other things is such a common occurrence that any programming technology that makes it quick and easy is to be welcomed.

[2] Even though it is not the most efficient strategy. Can you think of an improvement?

Simple patterns, like "even", are known as *concatenations*. To concatenate is to *link together* or *form a sequence of*. So, any sequence of characters is a pattern, specifically a concatenation pattern. Of course, there are other types of patterns. Unlike concatenations, the other types of patterns are associated with a particular *pattern metacharacter*.

4 Introducing The Pattern Metacharacters

In addition to concatenations, patterns can represent *repetitions* and *alternations*. It is also possible to state that a pattern may or may not be there, in that it is *optional*.

4.1 The + repetition metacharacter

The `+` metacharacter is read as *one or more of*. The following regular expression matches one or more occurrence of the letter "`T`":

```
/T+/
```

Which matches any of the following:

```
T
TTTTTT
TT
```

But, does not match any of these:

```
t
this and that
hello
ttttttttt
```

This is a good place to note that it is possible to use the command-line driven `regex.rb` program[3] to play with regular expressions. The `regex.rb` program takes two arguments: a *string* to match against and a *pattern*. Note that the *pattern* does not need to include the slash delimiters. If a match occurs, the output marks where the match occurred using "chevrons", otherwise a "no match found" message is displayed. Here's an example interactive session that shows `regex.rb` in action:

```
$ ruby regex.rb 'and the Tower appeared on the horizon' 'T+'
and the >>T<<ower appeared on the horizon
$ ruby regex.rb 'and the tower appeared on the horizon' 'q+'
no match found
$ ruby regex.rb 'and the tower appeared on the horizon' 'pear'
and the tower ap>>pear<<ed on the horizon
```

Repetitions can be combined with concatenations. This next pattern matches "`e`", followed by "`l`", followed by *one or more* occurrences of "`a`":

[3] Refer to the end of this document for the `regex.rb` source code.

```
/ela+/
```

In the above example, the repetition is said to *bind more closely* than the concatenation, in that only the letter immediately proceeding the `+` symbol is repeated. So, these strings successfully match the pattern:

```
elation
elaaaaaaaa
```

If a requirement exists to bind the repetition to more than one character (i.e., to a concatenation), use parentheses to indicate how many characters to repeat. Consider this regular expression:

```
/(ela)+/
```

Now, if the combination of "`e`", followed by "`l`", followed by "`a`" occurs one or more times, there's a match, as with these strings:

```
elaelaelaela
ela
```

This means that the "`(`" and "`)`" characters are also metacharacters, which is fine until a requirement exists to match either of these characters (or any other metacharacter, for that matter). When such a requirement exists, a metacharacter can have its special meaning *switched off* by the use of the \ character (which is known as *escaping*). Consider this regular expression:

```
/\(ela\)+/
```

which now matches an opening parenthesis, "`(`", followed by "`e`", followed by "`l`", followed by "`a`", followed by one or more occurrences of the closing parenthesis, "`)`". So, this string matches (`ruby` returns `0`):

```
(ela)))))
```

and this does not (`ruby` returns `nil`):

```
(ela(ela(ela
```

4.2 The | alternation metacharacter

Another important metacharacter is the vertical bar, `|`, which indicates *alternation*. Alternation offers choice. Here's an example, which matches any of the digit characters:

```
/0|1|2|3|4|5|6|7|8|9/
```

That is, the digit 0 or, alternatively, the digit 1 or, alternatively, the digit 2 or, alternatively, the digit 3, and so on, up to and including the digit 9. So, if a single digit occurs *anywhere in the string*, there's a match. All of these strings match (as they all contain at least one digit):

```
0123456789
there's a 0 in here somewhere
My telephone number is: 212-555-1029
```

As can be imagined, looking for any digit is a common requirement, as is trying to match any single lowercase or uppercase letter. It is possible to match any lowercase letter with this regular expression:

```
/a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z/
```

Just as it is possible to use this regular expressions to match any single uppercase letter:

```
/A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z/
```

Both seem like an awful lot of work just to match a single character. And they are. Ruby's regular expression shorthand to the rescue!

4.3 Metacharacter shorthand and character classes

In order to reduce the amount of work required, Ruby provides the *character class*, which is a shorthand notation for a long list of alternatives. Rather than using this regular expression to match any digit:

```
/0|1|2|3|4|5|6|7|8|9/
```

it is possible to define a character class, which means the same thing, as follows:

```
/[0123456789]/
```

That is, place the digits (or letters, or whatever) between the "`[`" and "`]`" characters, to indicate a series of alternations. This regular expression:

```
/[aeiou]/
```

is exactly the same as this one:

```
/a|e|i|o|u/
```

Most Ruby programmers prefer the character class version of the regular expression. When the first character of a character class is the `^` symbol (known as *hat*), the character class is *inverted*. This regular expression:

```
/[^aeiou]/
```

matches any single character that is *not* one of the five vowels. The `^` character can be included within a character class as a *literal character* by positioning it anywhere but the first position. Ranges can also be specified within character classes using the `-` symbol. This character class:

```
/[0123456789]/
```

can also be written as:

```
/[0-9]/
```

Which is shorter, more convenient and less prone to a typing error[4]. As the letters are also ranges, the long "any letter" regular expressions from earlier in this section can be rewritten as:

```
/[a-z]/
```

which matches any single lowercase letter, and like this:

```
/[A-Z]/
```

to match any single uppercase letter. If a requirement exists to match a literal "`-`" character, position the dash at the start of the character class:

```
/[-A-Z]/
```

The above regular expression now matches for any single uppercase letter or the dash. Combining character classes defines very specific concatenations. Consider this regular expression:

```
/[BCFHST][aeiou][mty]/
```

which matches any three letter word that starts with an uppercase letter from the first character class, has a vowel in the middle (the second character class) and ends in either the letter "`m`", "`t`" or "`y`" (the third character class). Each of the following words match this regular expression:

```
Bat
Hit
Tot
Cut
Say
```

while these words do not:

```
Hog
Can
May
bat
```

Note the last word, "bat", which *almost* matches, but does not as regular expressions are *case-sensitive* by default. To match words that start with either an uppercase or lowercase word, rewrite the regular expression like this:

```
/[BbCcFfHhSsTt][aeiou][mty]/
```

which now allows for both "bat" and "Bat" to match.

[4] Although I never make any of thsoe ... em, eh, sorry ... *those*.

4.4 More metacharacter shorthand

The character classes that match any single digit and any single letter (either lowercase or uppercase) are so common that Ruby provides further convenient shorthand related to them. Rather than using this character class to match any single digit:

```
/[0-9]/
```

Ruby provides the *slash-d* shorthand:

```
/\d/
```

So, `\d` means the same as `[0-9]`, and it is easy to remember, as "`d`" is short for *digit*. When it comes to lowercase and uppercase letters, Ruby groups these together with the digits and the underscore character to form the word character class. Instead of having to specify this character class:

```
/[a-zA-Z0-9_]/
```

all that's required is Ruby's *slash-w* shorthand:

```
/\w/
```

Again, this is easy to remember, as "`w`" is short for *word*. Another special character class is the *slash-s* shorthand, where "`s`" is short for *space*. This regular expression:

```
/\s/
```

is short for this regular expression:

```
/[^ \t\n\r\f]/
```

These characters are generally referred to as the space (or whitespace) characters. Each of these special character classes (sometimes referred to as the *classic* character classes) has an inverted form. To match any single character that is *not* a digit, use this regular expression:

```
/\D/
```

That is, the "`\D`" regular expression is "`\d`" inverted. Likewise, "`\W`" is "`\w`" inverted and "`\S`" is "`\s`" inverted.

The beauty of these special shorthands becomes clear when they are seen in action. Consider a regular expression that must match a digit, followed by any whitespace character, followed by two word characters and then any other character that is not a digit. Without the *specials*, the following regular expression does the trick[5]:

```
/[0-9][^ \t\n\r\f][a-zA-Z0-9_][a-zA-Z0-9_][^0-9]/
```

Here's the above regular expression rewritten to use "the specials":

```
/\d\s\w\w\D/
```

Note: less typing, less chance of error and more convenience. So, use regular expression shorthand to reduce the risk of error.

[5] I think. There are an awful lot of places that I could make a mistake typing that regex!

4.5 More repetition

Character class shorthand can be combined with the repetition metacharacter to great effect. This regular expression matches a word of any length:

```
/\w+/
```

and is read as: *one or more word characters*. Knowing this, the regular expression from the last section *could* be rewritten as:

```
/\d\s\w+\D/
```

However, this matches any number of word characters, not *exactly* two as was the requirement. Ruby provides a facility to match a specific number of occurrences of something. The { and } metacharacters are used to specify the number of occurrences to match. Here's the above regular expression rewritten to match exactly two word characters, as required:

```
/\d\s\w{2}\D/
```

If a requirement exists to match two but not more than four word characters, use this regular expression:

```
/\d\s\w{2,4}\D/
```

And, finally, if the requirement is to match at least two characters with no upper limit on the number of word characters to match, use this:

```
/\d\s\w{2,}\D/
```

4.6 The ? and * optional metacharacters

The *optional metacharacters* are used to specify that some part of a regular expression may or may not be there. Consider this example:

```
/[Bb]art?/
```

which matches any of the following words:

```
bar
Bar
bart
Bart
```

That is, the letter "`t`" is optional. More correctly, Ruby programmers read the `?` metacharacter as: *match zero or one time*. In other words, it is either there or it is not there, it's *optional*.

The `*` metacharacter matches *zero or more times*. Rewriting the above regular expression as follows has the effect of matching any number of occurrences of the letter "`t`", including *not matching it at all*:

```
/[Bb]art*/
```

Any of the following now match this regular expression:

```
bar
Bart
barttt
Bartttttttttttttttttttt!!!
```

Note that even though the last example appends three exclamation marks, there's still a match, as regular expressions match *anywhere in a string*. Care is needed when using the `*` metacharacter. Consider this regular expression, which *always* matches successfully:

```
/p*/
```

When applied against any string, the `p*` regular expression always matches, as the pattern is looking for zero or more occurrences of the letter "`p`". If the string matched against contains a "`p`", there's a match. Equally, if the string does not contain a "`p`", there is also a match! Remember: the `*` matches zero or more times, and something - whether it is the letter "`p`" or anything else, for that matter - is always *not there*. In this next example, the "`q*`" always matches:

```
$ ruby regex.rb "and the tower appeared on the horizon" 'q*'
>><<and the tower appeared on the horizon
```

4.7 The any character metacharacter

There is often a requirement to match any character, regardless of whether it is a word, digit or whitespace character. The `.` metacharacter does just that:

```
/[Bb]ar./
```

The use of the *any character metacharacter* allows the above pattern to successfully match any of these strings[6]:

```
barb
bark
barking
embarking
barn
Bart
Barry
```

Appending the `?` optional metacharacter to the pattern, thus:

```
/[Bb]ar.?/
```

allows words such as "bar" and "Bar" to match also.

[6] The temptation to use the letter "`f`" in this example was strong, but you'll be glad to know I resisted.

5 Anchors

The last example from the last section highlights, once again, the fact that the match is successful if the pattern is found *anywhere* in the string under consideration. Note that "bark", "barking" and "embarking" are all successful matches. This can often result in patterns matching when they were not expected to, which can sometimes be a surprise. But, what if a requirement exists to match an entire word, such as "bark", but not match "barking" and "embarking" (as the word "bark" is embedded in them)?

The *word boundary* metacharacters allow a regular expression to be *anchored* at a word boundary - that is the space between a word and something else, which is defined as the position between "\w" and "\W".

5.1 The \b word boundary metacharacter

To match an entire word, surround the word to be matched with the \b word boundary metacharacter, as follows:

```
/\bbark\b/
```

This string now successfully matches:

```
That dog sure has a loud bark, doesn't it?
```

as the word "bark" is surrounded by word boundaries, whereas this string does not match:

```
That dog's barking is driving me crazy!
```

The \b metacharacter has an inverse in \B, which matches at any position that is *not* a word boundary. Note that this regular expression:

```
/\Bbark\B/
```

matches "embarking" but not "bark" or "barking".

5.2 The ^ start-of-line metacharacter

To anchor the regular expression to the start of a string (or line), use the ^ metacharacter:

```
/^Bioinformatics/
```

which states that a successful match to a string must begin with the word "Bioinformatics", as follows:

```
Bioinformatics, Biocomputing and Perl is a great book.
```

The next string does not match, as the match cannot be made at the start of the string:

```
For a great introduction to Bioinformatics, see Moorhouse & Barry (2004).
```

5.3 The $ end-of-line metacharacter

To anchor the regular expression to the end of a string (or line), use the $ metacharacter:

```
/Ruby$/
```

which matches successfully with this string:

```
One of my favourite programming languages is Ruby
```

but not this one:

```
Is Ruby your favourite programming language?
```

A common regular expression to match against a blank line is:

```
/^$/
```

That is, the line has a start, an end, and nothing in-between: it's blank.

6 The Binding Operators

Consider this simple program, called `simplepat.rb`:

```
#! /usr/bin/ruby -w
#
# The 'simplepat.rb' program.

while line = gets
    puts "Got a blank line."            if line =~ /^$/
    puts "Line has a curly brace."      if line =~ /[]{]/
    puts "Line contains 'program'."     if line =~ /\bprogram\b/
end # of while.
```

The `simplepat.rb` program keeps reading lines of input from `STDIN` until there are no more lines to read. The line read in is assigned to the `line`. Three `puts` method calls form the body of the loop, with each statement qualified with an `if` conditional statement. Each of the `if` statements tries to match to a specific regular expression against the current contents of the line[7]. The *binding operator*, written as `=~` is used to tell `ruby` that a regular expression is to be applied (or *bound*) to a named variable. For example, this statement:

```
if line =~ /^$/
```

checks to see if the `line` variable contains a blank line. In addition to `=~`, there's also a *not binding operator*, `!~`, which is the logical negation of `=~`. This statement:

```
if line !~ /^$/
```

checks to see if `line` contains anything other than a blank line. The binding operators are very useful, but really come into their own when combined with *grouping parentheses*.

[7] The meaning of each should be clear. If they are not, you are advised to go back to the start of this document and start again. Sigh.

7 Remembering What Was Matched

The grouping parentheses were introduced earlier, when they were used to group a number of letters together so that they could be repeated:

```
/(ela)+/
```

It wasn't mentioned then, but when the parentheses are used to group in this way, `ruby` remembers the value which matched that part of the regular expression, often referred to as a subpattern. For each set of parentheses, `ruby` creates a special variable to hold what matched. These special variables, often referred to as the *after-match variables*, are numbered upward from 1.

Here's a small program, called `grouping.rb`, which demonstrates how the after-match variables are used:

```
#! /usr/bin/ruby -w
#
# The 'grouping.rb' program.

while line = gets
    line =~ /\w+ (\w+) \w+ (\w+)/

    puts "Second word: '#{$1}' on line #{$.}." if defined? $1
    puts "Fourth word: '#{$2}' on line #{$.}." if defined? $2
end # of while.
```

Each line read into this program is assigned to `line`, which is then bound against a regular expression. The pattern looks for a word, `\w+`, a space, another word which is to be remembered (note the use of parentheses), another space, another word, another space and another remembered word[8]. After a successful pattern match the two remembered values are automatically assigned by `ruby` to the special variables, `$1` and `$2`. The `puts` statement displays what was found (assuming it was, note the use of `defined?`). Note, too, the use of the "`$.`" variable, another internal Ruby variable, which contains the current line number of the input file being processed. Try the `grouping.rb` program against the `ruby.data.txt` data-file. It is also possible to *nest* parentheses. Consider the following regex:

```
line =~ /\w+ ((\w+) \w+ (\w+))/;
```

Can you work out what will match? As an exercise, amend the `grouping.rb` program to use the above regex and re-execute against the `ruby.data.txt` data-file. Did the results agree with what you expected?[9]

When working with nested parentheses, count the opening parentheses, starting with the leftmost, to determine which parts of the pattern are assigned to which of the after-match variables.

[8] As you can see, it is often easier to write a regular expression using shorthand than it is to actually describe it *in words*.

[9] Remember to display the results for the `$3`.

8 Greedy By Default

Consider this regular expression:

```
/(.+), Bart/
```

matched against this string:

```
Get over here, now, Bart!  Do you hear me, Bart?
```

The pattern matches one or more of any character, `.+`, a literal comma, a space character, then the word "Bart". The parentheses ensure that anything matched by `.+` is remembered in the `$1` after-match variable. After performing the match, `$1` contains this string[10]:

```
Get over here, now, Bart!  Do you hear me
```

This may come as a bit of a surprise, as it would be reasonable to think that the match succeeds when the first "Bart" is encountered, not the second. A reasonable assumption indeed, but incorrect. By default, `ruby` performs *greedy matching*, in that an attempt is always made to match *as much of the string as possible*, that is, the longest possible match. To specify that non-greedy (or *lazy*) matching should be applied to part of the regular expression (or *subpattern*), qualify it with the `?` character:

```
/(.+?), Bart/
```

Note that the `?` character when used in this way does not mean *optional*. It means *non-greedy*. Rather than match as much as possible, this part of the regular expression now matches *as little as possible*. When matched against the string from earlier, this non-greedy regular expression remembers the following value in the `$1` after-match variable:

```
Get over here, now
```

In addition to the use of the `?` non-greedy qualifier with the `+` metacharacter, it can also be used with the `*` metacharacter. It can also be used with the `{x}`, `{x,y}` and `{x,}` repetition specifiers (where "`x`" and "`y`" specify the minimum and maximum number of matches, respectively). Being able to control when `ruby` is and is not greedy is important.

9 Alternative Pattern Delimiters

The use of the `/` character as a regular expression delimiter suffices for most needs. However, consider writing a regular expression to match against a string like this:

[10] Try this is `irb` to confirm this behaviour.

```
/usr/bin/ruby
```

It is *not* possible to write the regular expression as follows:

```
//\w+/\w+/\w+/
```

as `ruby` will treat the second `/` character as the end of the pattern and ignore the `\w+/\w+/\w+/` bit. Whoops! It is possible to *escape* the `/` characters that are part of the pattern:

```
/\/\w+\/\w+\/\w+/
```

to ensure that the leftmost and rightmost `/` characters are treated as pattern delimiters. Unfortunately, the pattern is now harder to read and understand, and it gets worse when each of the matched words is remembered:

```
/\/(\w+)\/(\w+)\/(\w+)/
```

In situations such as this, Ruby allows alternative delimiters to be specified. To use the alternative delimiters, prefix the regular expression with "`%r{`" and postfix it with "`}`". The above *escaped* example regular expression can be rewritten as:

```
%r{/\w+/\w+/\w+}
```

or, if the matched words are to be remembered, like so:

```
%r{/(\w+)/(\w+)/(\w+)}
```

There is now no confusion as to the inclusion of the `/` characters within the regular expression: they are to be treated literally, *not* as delimiters.

10 And There's More ...

Ruby regular expression technology includes much more including pattern-based substitution, blackslash sequences and object-oriented regexes. Refer to Chapter 5 of *PickAxe* for more details.

An excellent resource on regexes is *Jeffrey E. F. Friedl's* book *Mastering Regular Expressions, 2nd Edition*. Although it only contains passing references to Ruby, it is the undisputed guide to all things regex. Highly recommended reading.

11 The `regex.rb` Source Code

Here's the complete source code to the `regex.rb` program:

```
#! /usr/bin/ruby -w

#
# == Synopsis
# A small ruby program which shows the user where a regex matches.
#
# == Usage
# There are two command-line arguments:
#  "target": the string to match against.
#  "regex": the pattern/regular-expression.
#
# == Author
# Paul Barry, The Institute of Technology, Carlow, Ireland.
#
# == Copyright
# Copyright (c) 2006, Paul Barry.
# Licensed under GPL v2 .
#

def show_regexp( target, regex )
    #
    # Based on the code from page 69, Programming Ruby, 2nd Edition by Dave Thomas.
    #
    if target =~ regex
        # Highlight with chevrons where the successful match occurs.
        "#{$`}>>#{$&}<<#{$'}"
    else
        "no match found"
    end # of if.
end # of show_regexp.

target, regex = ARGV.collect { |arg| arg }  # Doing things the Ruby way ...
regex = Regexp.new( regex )                 # Convert to regex.
puts show_regexp( target, regex )           # Show the results.
```