NAME

perlfaq6 - Regular Expressions ($Revision: 1.38 $, $Date: 2005/12/31 00:54:37 $)

DESCRIPTION

This section is surprisingly small because the rest of the FAQ is littered with answers involving regular expressions. For example, decoding a URL and checking whether something is a number are handled with regular expressions, but those answers are found elsewhere in this document (in *perlfaq9*: "How do I decode or create those %-encodings on the web" and *perlfaq4*: "How do I determine whether a scalar is a number/whole/integer/float", to be precise).

How can I hope to use regular expressions without creating illegible and unmaintainable code?

Three techniques can make regular expressions maintainable and understandable.

Comments Outside the Regex

Describe what you're doing and how you're doing it, using normal Perl comments.

```
# turn the line into the first word, a colon, and the
# number of characters on the rest of the line
s/^(\w+)(.*)/ lc($1) . ":" . length($2) /meg;
```

Comments Inside the Regex

The `/x` modifier causes whitespace to be ignored in a regex pattern (except in a character class), and also allows you to use normal comments there, too. As you can imagine, whitespace and comments help a lot.

`/x` lets you turn this:

```
s{<(?:[^>'"]*|".*?"|'.*?')+>}{}gs;
```

into this:

```
s{ <                    # opening angle bracket
    (?:                 # Non-backreffing grouping paren
         [^>'"] *       # 0 or more things that are neither > nor
 ' nor "
            |           #    or else
         ".*?"          # a section between double quotes (stingy
 match)
            |           #    or else
         '.*?'          # a section between single quotes (stingy
 match)
       ) +              #   all occurring one or more times
      >                 # closing angle bracket
   }{}gsx;              # replace with nothing, i.e. delete
```

It's still not quite so clear as prose, but it is very useful for describing the meaning of each part of the pattern.

Different Delimiters

While we normally think of patterns as being delimited with `/` characters, they can be delimited by almost any character. *perlre* describes this. For example, the `s///` above uses braces as delimiters. Selecting another delimiter can avoid quoting the delimiter within the pattern:

```
s/\/usr\/local/\/usr\/share/g; # bad delimiter choice
s#/usr/local#/usr/share#g;  # better
```

I'm having trouble matching over more than one line. What's wrong?

Either you don't have more than one line in the string you're looking at (probably), or else you aren't using the correct modifier(s) on your pattern (possibly).

There are many ways to get multiline data into a string. If you want it to happen automatically while reading input, you'll want to set $/ (probably to '' for paragraphs or `undef` for the whole file) to allow you to read more than one line at a time.

Read *perlre* to help you decide which of `/s` and `/m` (or both) you might want to use: `/s` allows dot to include newline, and `/m` allows caret and dollar to match next to a newline, not just at the end of the string. You do need to make sure that you've actually got a multiline string in there.

For example, this program detects duplicate words, even when they span line breaks (but not paragraph ones). For this example, we don't need `/s` because we aren't using dot in a regular expression that we want to cross line boundaries. Neither do we need `/m` because we aren't wanting caret or dollar to match at any point inside the record next to newlines. But it's imperative that $/ be set to something other than the default, or else we won't actually ever have a multiline record read in.

```
    $/ = '';    # read in more whole paragraph, not just one line
    while ( <> ) {
while ( /\b([\w'-]+)(\s+\1)+\b/gi ) {   # word starts alpha
    print "Duplicate $1 at paragraph $.\n";
}
    }
```

Here's code that finds sentences that begin with "From " (which would be mangled by many mailers):

```
    $/ = '';    # read in more whole paragraph, not just one line
    while ( <> ) {
while ( /^From /gm ) { # /m makes ^ match next to \n
    print "leading from in paragraph $.\n";
}
    }
```

Here's code that finds everything between START and END in a paragraph:

```
    undef $/;    # read in whole file, not just one line or paragraph
    while ( <> ) {
while ( /START(.*?)END/sgm ) { # /s makes . cross line boundaries
    print "$1\n";
}
    }
```

How can I pull out lines between two patterns that are themselves on different lines?

You can use Perl's somewhat exotic `..` operator (documented in *perlop*):

```
    perl -ne 'print if /START/ .. /END/' file1 file2 ...
```

If you wanted text and not lines, you would use

```
    perl -0777 -ne 'print "$1\n" while /START(.*?)END/gs' file1 file2 ...
```

But if you want nested occurrences of START through END, you'll run up against the problem described in the question in this section on matching balanced text.

Here's another example of using `..`:

```
    while (<>) {
        $in_header =   1  .. /^$/;
        $in_body   = /^$/ .. eof();
# now choose between them
    } continue {
reset if eof();  # fix $.
    }
```

I put a regular expression into $/ but it didn't work. What's wrong?

Up to Perl 5.8.0, $/ has to be a string. This may change in 5.10, but don't get your hopes up. Until then, you can use these examples if you really need to do this.

If you have File::Stream, this is easy.

```
    use File::Stream;
             my $stream = File::Stream->new(
                  $filehandle,
                  separator => qr/\s*,\s*/,
                  );

    print "$_\n" while <$stream>;
```

If you don't have File::Stream, you have to do a little more work.

You can use the four argument form of sysread to continually add to a buffer. After you add to the buffer, you check if you have a complete line (using your regular expression).

```
       local $_ = "";
       while( sysread FH, $_, 8192, length ) {
          while( s/^((?s).*?)your_pattern/ ) {
             my $record = $1;
             # do stuff here.
          }
       }
```

```
You can do the same thing with foreach and a match using the
c flag and the \G anchor, if you do not mind your entire file
being in memory at the end.
```

```
       local $_ = "";
       while( sysread FH, $_, 8192, length ) {
          foreach my $record ( m/\G((?s).*?)your_pattern/gc ) {
             # do stuff here.
          }
          substr( $_, 0, pos ) = "" if pos;
       }
```

How do I substitute case insensitively on the LHS while preserving case on the RHS?

Here's a lovely Perlish solution by Larry Rosler. It exploits properties of bitwise xor on ASCII strings.

```
    $_= "this is a TEsT case";

    $old = 'test';
    $new = 'success';
```

```
    s{(\Q$old\E)}
     { uc $new | (uc $1 ^ $1) .
(uc(substr $1, -1) ^ substr $1, -1) x
     (length($new) - length $1)
     }egi;

    print;
```

And here it is as a subroutine, modeled after the above:

```
    sub preserve_case($$) {
my ($old, $new) = @_;
my $mask = uc $old ^ $old;

uc $new | $mask .
     substr($mask, -1) x (length($new) - length($old))
    }

    $a = "this is a TEsT case";
    $a =~ s/(test)/preserve_case($1, "success")/egi;
    print "$a\n";
```

This prints:

```
    this is a SUcCESS case
```

As an alternative, to keep the case of the replacement word if it is longer than the original, you can use this code, by Jeff Pinyan:

```
  sub preserve_case {
    my ($from, $to) = @_;
    my ($lf, $lt) = map length, @_;

    if ($lt < $lf) { $from = substr $from, 0, $lt }
    else { $from .= substr $to, $lf }

    return uc $to | ($from ^ uc $from);
  }
```

This changes the sentence to "this is a SUcCess case."

Just to show that C programmers can write C in any programming language, if you prefer a more C-like solution, the following script makes the substitution have the same case, letter by letter, as the original. (It also happens to run about 240% slower than the Perlish solution runs.) If the substitution has more characters than the string being substituted, the case of the last character is used for the rest of the substitution.

```
    # Original by Nathan Torkington, massaged by Jeffrey Friedl
    #
    sub preserve_case($$)
    {
        my ($old, $new) = @_;
        my ($state) = 0; # 0 = no change; 1 = lc; 2 = uc
        my ($i, $oldlen, $newlen, $c) = (0, length($old), length($new));
        my ($len) = $oldlen < $newlen ? $oldlen : $newlen;
```

```
            for ($i = 0; $i < $len; $i++) {
                if ($c = substr($old, $i, 1), $c =~ /[\W\d_]/) {
                    $state = 0;
                } elsif (lc $c eq $c) {
                    substr($new, $i, 1) = lc(substr($new, $i, 1));
                    $state = 1;
                } else {
                    substr($new, $i, 1) = uc(substr($new, $i, 1));
                    $state = 2;
                }
            }
            # finish up with any remaining new (for when new is longer than
        old)
            if ($newlen > $oldlen) {
                if ($state == 1) {
                    substr($new, $oldlen) = lc(substr($new, $oldlen));
                } elsif ($state == 2) {
                    substr($new, $oldlen) = uc(substr($new, $oldlen));
                }
            }
            return $new;
        }
```

How can I make \w match national character sets?

Put `use locale;` in your script. The \w character class is taken from the current locale.

See *perllocale* for details.

How can I match a locale-smart version of /[a-zA-Z]/?

You can use the POSIX character class syntax `/[[:alpha:]]/` documented in *perlre*.

No matter which locale you are in, the alphabetic characters are the characters in \w without the digits and the underscore. As a regex, that looks like `/[^\W\d_]/`. Its complement, the non-alphabetics, is then everything in \W along with the digits and the underscore, or `/[\W\d_]/`.

How can I quote a variable to use in a regex?

The Perl parser will expand $variable and @variable references in regular expressions unless the delimiter is a single quote. Remember, too, that the right-hand side of a `s///` substitution is considered a double-quoted string (see *perlop* for more details). Remember also that any regex special characters will be acted on unless you precede the substitution with \Q. Here's an example:

```
    $string = "Placido P. Octopus";
    $regex  = "P.";

    $string =~ s/$regex/Polyp/;
    # $string is now "Polypacido P. Octopus"
```

Because `.` is special in regular expressions, and can match any single character, the regex `P.` here has matched the <Pl> in the original string.

To escape the special meaning of `.`, we use `\Q`:

```
    $string = "Placido P. Octopus";
    $regex  = "P.";

    $string =~ s/\Q$regex/Polyp/;
```

```
    # $string is now "Placido Polyp Octopus"
```

The use of `\Q` causes the <.> in the regex to be treated as a regular character, so that `P.` matches a `P` followed by a dot.

What is /o really for?

Using a variable in a regular expression match forces a re-evaluation (and perhaps recompilation) each time the regular expression is encountered. The `/o` modifier locks in the regex the first time it's used. This always happens in a constant regular expression, and in fact, the pattern was compiled into the internal format at the same time your entire program was.

Use of `/o` is irrelevant unless variable interpolation is used in the pattern, and if so, the regex engine will neither know nor care whether the variables change after the pattern is evaluated the *very first* time.

`/o` is often used to gain an extra measure of efficiency by not performing subsequent evaluations when you know it won't matter (because you know the variables won't change), or more rarely, when you don't want the regex to notice if they do.

For example, here's a "paragrep" program:

```
    $/ = '';  # paragraph mode
    $pat = shift;
    while (<>) {
        print if /$pat/o;
    }
```

How do I use a regular expression to strip C style comments from a file?

While this actually can be done, it's much harder than you'd think. For example, this one-liner

```
    perl -0777 -pe 's{/\*.*?\*/}{}gs' foo.c
```

will work in many but not all cases. You see, it's too simple-minded for certain kinds of C programs, in particular, those with what appear to be comments in quoted strings. For that, you'd need something like this, created by Jeffrey Friedl and later modified by Fred Curtis.

```
    $/ = undef;
    $_ = <>;

s#/\*[^*]*\*+([^/*][^*]*\*+)*/|("(\\.|[^"\\])*"|'(\\.|[^'\\])*'|.[^/"'\\]*)
#defined $2 ? $2 : ""#gse;
    print;
```

This could, of course, be more legibly written with the `/x` modifier, adding whitespace and comments. Here it is expanded, courtesy of Fred Curtis.

```
    s{
       /\*         ##  Start of /* ... */ comment
       [^*]*\*+    ##  Non-* followed by 1-or-more *'s
       (
         [^/*][^*]*\*+
       )*          ##  0-or-more things which don't start with /
                   ##    but do end with '*'
       /           ##  End of /* ... */ comment

     |         ##     OR  various things which aren't comments:
```

```
        (
          "           ##  Start of " ... " string
          (
            \\.           ##  Escaped char
          |               ##    OR
            [^"\\]        ##  Non "\
          )*
          "           ##  End of " ... " string

        |         ##     OR

          '           ##  Start of ' ... ' string
          (
            \\.           ##  Escaped char
          |               ##    OR
            [^'\\]        ##  Non '\
          )*
          '           ##  End of ' ... ' string

        |         ##     OR

          .           ##  Anything other char
          [^/"'\\]*   ##  Chars which doesn't start a comment, string or
escape
        )
     }{defined $2 ? $2 : ""}gxse;
```

A slight modification also removes C++ comments:

```
s#/\*[^*]*\*+([^/*][^*]*\*+)*/|//[^\n]*|("(\\.|[^"\\])*"|'(\\.|[^'\\])*'|.[
^/"'\\]*)#defined $2 ? $2 : ""#gse;
```

Can I use Perl regular expressions to match balanced text?

Historically, Perl regular expressions were not capable of matching balanced text. As of more recent versions of perl including 5.6.1 experimental features have been added that make it possible to do this. Look at the documentation for the (??{ }) construct in recent perlre manual pages to see an example of matching balanced parentheses. Be sure to take special notice of the warnings present in the manual before making use of this feature.

CPAN contains many modules that can be useful for matching text depending on the context. Damian Conway provides some useful patterns in Regexp::Common. The module Text::Balanced provides a general solution to this problem.

One of the common applications of balanced text matching is working with XML and HTML. There are many modules available that support these needs. Two examples are HTML::Parser and XML::Parser. There are many others.

An elaborate subroutine (for 7-bit ASCII only) to pull out balanced and possibly nested single chars, like ` and ', { and }, or (and) can be found in http://www.cpan.org/authors/id/TOMC/scripts/pull_quotes.gz .

The C::Scan module from CPAN also contains such subs for internal use, but they are undocumented.

What does it mean that regexes are greedy? How can I get around it?

Most people mean that greedy regexes match as much as they can. Technically speaking, it's actually the quantifiers (?, *, +, {}) that are greedy rather than the whole pattern; Perl prefers local greed and immediate gratification to overall greed. To get non-greedy versions of the same quantifiers, use (??, *?, +?, {}?).

An example:

```
$s1 = $s2 = "I am very very cold";
$s1 =~ s/ve.*y //;      # I am cold
$s2 =~ s/ve.*?y //;     # I am very cold
```

Notice how the second substitution stopped matching as soon as it encountered "y ". The *? quantifier effectively tells the regular expression engine to find a match as quickly as possible and pass control on to whatever is next in line, like you would if you were playing hot potato.

How do I process each word on each line?

Use the split function:

```
   while (<>) {
foreach $word ( split ) {
    # do something with $word here
}
   }
```

Note that this isn't really a word in the English sense; it's just chunks of consecutive non-whitespace characters.

To work with only alphanumeric sequences (including underscores), you might consider

```
   while (<>) {
foreach $word (m/(\w+)/g) {
    # do something with $word here
}
   }
```

How can I print out a word-frequency or line-frequency summary?

To do this, you have to parse out each word in the input stream. We'll pretend that by word you mean chunk of alphabetics, hyphens, or apostrophes, rather than the non-whitespace chunk idea of a word given in the previous question:

```
   while (<>) {
while ( /(\b[^\W_\d][\w'-]+\b)/g ) {   # misses "`sheep'"
    $seen{$1}++;
}
   }
   while ( ($word, $count) = each %seen ) {
print "$count $word\n";
   }
```

If you wanted to do the same thing for lines, you wouldn't need a regular expression:

```
   while (<>) {
$seen{$_}++;
   }
   while ( ($line, $count) = each %seen ) {
```

```
print "$count $line";
   }
```

If you want these output in a sorted order, see *perlfaq4*: "How do I sort a hash (optionally by value instead of key)?".

How can I do approximate matching?

See the module String::Approx available from CPAN.

How do I efficiently match many regular expressions at once?

(contributed by brian d foy)

Avoid asking Perl to compile a regular expression every time you want to match it. In this example, perl must recompile the regular expression for every iteration of the foreach() loop since it has no way to know what $pattern will be.

```
   @patterns = qw( foo bar baz );

   LINE: while( <> )
    {
 foreach $pattern ( @patterns )
  {
     print if /\b$pattern\b/i;
     next LINE;
     }
 }
```

The qr// operator showed up in perl 5.005. It compiles a regular expression, but doesn't apply it. When you use the pre-compiled version of the regex, perl does less work. In this example, I inserted a map() to turn each pattern into its pre-compiled form. The rest of the script is the same, but faster.

```
   @patterns = map { qr/\b$_\b/i } qw( foo bar baz );

   LINE: while( <> )
    {
 foreach $pattern ( @patterns )
  {
     print if /\b$pattern\b/i;
     next LINE;
     }
 }
```

In some cases, you may be able to make several patterns into a single regular expression. Beware of situations that require backtracking though.

```
$regex = join '|', qw( foo bar baz );

   LINE: while( <> )
    {
 print if /\b(?:$regex)\b/i;
 }
```

For more details on regular expression efficiency, see Mastering Regular Expressions by Jeffrey Freidl. He explains how regular expressions engine work and why some patterns are surprisingly inefficient. Once you understand how perl applies regular expressions, you can tune them for

individual situations.

Why don't word-boundary searches with \b work for me?

(contributed by brian d foy)

Ensure that you know what \b really does: it's the boundary between a word character, \w, and something that isn't a word character. That thing that isn't a word character might be \W, but it can also be the start or end of the string.

It's not (not!) the boundary between whitespace and non-whitespace, and it's not the stuff between words we use to create sentences.

In regex speak, a word boundary (\b) is a "zero width assertion", meaning that it doesn't represent a character in the string, but a condition at a certain position.

For the regular expression, /\bPerl\b/, there has to be a word boundary before the "P" and after the "l". As long as something other than a word character precedes the "P" and succeeds the "l", the pattern will match. These strings match /\bPerl\b/.

```
"Perl"    # no word char before P or after l
"Perl "   # same as previous (space is not a word char)
"'Perl'"  # the ' char is not a word char
"Perl's"  # no word char before P, non-word char after "l"
```

These strings do not match /\bPerl\b/.

```
"Perl_"   # _ is a word char!
"Perler"  # no word char before P, but one after l
```

You don't have to use \b to match words though. You can look for non-word characters surrounded by word characters. These strings match the pattern /\b'\b/.

```
"don't"   # the ' char is surrounded by "n" and "t"
"qep'a'"  # the ' char is surrounded by "p" and "a"
```

These strings do not match /\b'\b/.

```
"foo'"    # there is no word char after non-word '
```

You can also use the complement of \b, \B, to specify that there should not be a word boundary.

In the pattern /\Bam\B/, there must be a word character before the "a" and after the "m". These patterns match /\Bam\B/:

```
"llama"   # "am" surrounded by word chars
"Samuel"  # same
```

These strings do not match /\Bam\B/

```
"Sam"      # no word boundary before "a", but one after "m"
"I am Sam" # "am" surrounded by non-word chars
```

Why does using $&, $`, or $' slow my program down?

(contributed by Anno Siegel)

Once Perl sees that you need one of these variables anywhere in the program, it provides them on each and every pattern match. That means that on every pattern match the entire string will be copied, part of it to $`, part to $&, and part to $'. Thus the penalty is most severe with long strings and

patterns that match often. Avoid $&, $', and $` if you can, but if you can't, once you've used them at all, use them at will because you've already paid the price. Remember that some algorithms really appreciate them. As of the 5.005 release, the $& variable is no longer "expensive" the way the other two are.

Since Perl 5.6.1 the special variables @- and @+ can functionally replace $`, $& and $'. These arrays contain pointers to the beginning and end of each match (see perlvar for the full story), so they give you essentially the same information, but without the risk of excessive string copying.

What good is \G in a regular expression?

You use the `\G` anchor to start the next match on the same string where the last match left off. The regular expression engine cannot skip over any characters to find the next match with this anchor, so `\G` is similar to the beginning of string anchor, ^. The `\G` anchor is typically used with the g flag. It uses the value of pos() as the position to start the next match. As the match operator makes successive matches, it updates pos() with the position of the next character past the last match (or the first character of the next match, depending on how you like to look at it). Each string has its own pos() value.

Suppose you want to match all of consective pairs of digits in a string like "1122a44" and stop matching when you encounter non-digits. You want to match 11 and 22 but the letter <a> shows up between 22 and 44 and you want to stop at a. Simply matching pairs of digits skips over the a and still matches 44.

```
$_ = "1122a44";
my @pairs = m/(\d\d)/g;   # qw( 11 22 44 )
```

If you use the \G anchor, you force the match after 22 to start with the a. The regular expression cannot match there since it does not find a digit, so the next match fails and the match operator returns the pairs it already found.

```
$_ = "1122a44";
my @pairs = m/\G(\d\d)/g; # qw( 11 22 )
```

You can also use the `\G` anchor in scalar context. You still need the g flag.

```
$_ = "1122a44";
while( m/\G(\d\d)/g )
 {
 print "Found $1\n";
 }
```

After the match fails at the letter a, perl resets pos() and the next match on the same string starts at the beginning.

```
$_ = "1122a44";
while( m/\G(\d\d)/g )
 {
 print "Found $1\n";
 }

print "Found $1 after while" if m/(\d\d)/g; # finds "11"
```

You can disable pos() resets on fail with the c flag. Subsequent matches start where the last successful match ended (the value of pos()) even if a match on the same string as failed in the meantime. In this case, the match after the while() loop starts at the a (where the last match stopped), and since it does not use any anchor it can skip over the a to find "44".

```
$_ = "1122a44";
while( m/\G(\d\d)/gc )
 {
 print "Found $1\n";
 }

print "Found $1 after while" if m/(\d\d)/g; # finds "44"
```

Typically you use the `\G` anchor with the c flag when you want to try a different match if one fails, such as in a tokenizer. Jeffrey Friedl offers this example which works in 5.004 or later.

```
while (<>) {
  chomp;
  PARSER: {
       m/ \G( \d+\b    )/gcx   && do { print "number: $1\n";  redo; };
       m/ \G( \w+      )/gcx   && do { print "word:   $1\n";  redo; };
       m/ \G( \s+      )/gcx   && do { print "space:  $1\n";  redo; };
       m/ \G( [^\w\d]+ )/gcx   && do { print "other:  $1\n";  redo; };
  }
}
```

For each line, the PARSER loop first tries to match a series of digits followed by a word boundary. This match has to start at the place the last match left off (or the beginning of the string on the first match). Since `m/ \G( \d+\b )/gcx` uses the c flag, if the string does not match that regular expression, perl does not reset pos() and the next match starts at the same position to try a different pattern.

Are Perl regexes DFAs or NFAs? Are they POSIX compliant?

While it's true that Perl's regular expressions resemble the DFAs (deterministic finite automata) of the egrep(1) program, they are in fact implemented as NFAs (non-deterministic finite automata) to allow backtracking and backreferencing. And they aren't POSIX-style either, because those guarantee worst-case behavior for all cases. (It seems that some people prefer guarantees of consistency, even when what's guaranteed is slowness.) See the book "Mastering Regular Expressions" (from O'Reilly) by Jeffrey Friedl for all the details you could ever hope to know on these matters (a full citation appears in *perlfaq2*).

What's wrong with using grep in a void context?

The problem is that grep builds a return list, regardless of the context. This means you're making Perl go to the trouble of building a list that you then just throw away. If the list is large, you waste both time and space. If your intent is to iterate over the list, then use a for loop for this purpose.

In perls older than 5.8.1, map suffers from this problem as well. But since 5.8.1, this has been fixed, and map is context aware - in void context, no lists are constructed.

How can I match strings with multibyte characters?

Starting from Perl 5.6 Perl has had some level of multibyte character support. Perl 5.8 or later is recommended. Supported multibyte character repertoires include Unicode, and legacy encodings through the Encode module. See *perluniintro*, *perlunicode*, and *Encode*.

If you are stuck with older Perls, you can do Unicode with the `Unicode::String` module, and character conversions using the `Unicode::Map8` and `Unicode::Map` modules. If you are using Japanese encodings, you might try using the jperl 5.005_03.

Finally, the following set of approaches was offered by Jeffrey Friedl, whose article in issue #5 of The Perl Journal talks about this very matter.

Let's suppose you have some weird Martian encoding where pairs of ASCII uppercase letters encode

single Martian letters (i.e. the two bytes "CV" make a single Martian letter, as do the two bytes "SG", "VS", "XX", etc.). Other bytes represent single characters, just like ASCII.

So, the string of Martian "I am CVSGXX!" uses 12 bytes to encode the nine characters 'I', ' ', 'a', 'm', ' ', 'CV', 'SG', 'XX', '!'.

Now, say you want to search for the single character `/GX/`. Perl doesn't know about Martian, so it'll find the two bytes "GX" in the "I am CVSGXX!" string, even though that character isn't there: it just looks like it is because "SG" is next to "XX", but there's no real "GX". This is a big problem.

Here are a few ways, all painful, to deal with it:

```
$martian =~ s/([A-Z][A-Z])/ $1 /g; # Make sure adjacent "martian"
                                   # bytes are no longer adjacent.
print "found GX!\n" if $martian =~ /GX/;
```

Or like this:

```
@chars = $martian =~ m/([A-Z][A-Z]|[^A-Z])/g;
# above is conceptually similar to:     @chars = $text =~ m/(.)/g;
#
foreach $char (@chars) {
    print "found GX!\n", last if $char eq 'GX';
}
```

Or like this:

```
while ($martian =~ m/\G([A-Z][A-Z]|.)/gs) {  # \G probably unneeded
    print "found GX!\n", last if $1 eq 'GX';
}
```

Here's another, slightly less painful, way to do it from Benjamin Goldberg, who uses a zero-width negative look-behind assertion.

```
print "found GX!\n" if $martian =~ m/
    (?<![A-Z])
    (?:[A-Z][A-Z])*?
    GX
 /x;
```

This succeeds if the "martian" character GX is in the string, and fails otherwise. If you don't like using (?<!), a zero-width negative look-behind assertion, you can replace (?<![A-Z]) with (?:^|[^A-Z]).

It does have the drawback of putting the wrong thing in $-[0] and $+[0], but this usually can be worked around.

How do I match a pattern that is supplied by the user?

Well, if it's really a pattern, then just use

```
chomp($pattern = <STDIN>);
if ($line =~ /$pattern/) { }
```

Alternatively, since you have no guarantee that your user entered a valid regular expression, trap the exception this way:

```
if (eval { $line =~ /$pattern/ }) { }
```

If all you really want is to search for a string, not a pattern, then you should either use the index() function, which is made for string searching, or, if you can't be disabused of using a pattern match on a non-pattern, then be sure to use \Q...\E, documented in *perlre*.

```
    $pattern = <STDIN>;

    open (FILE, $input) or die "Couldn't open input $input: $!; aborting";
    while (<FILE>) {
 print if /\Q$pattern\E/;
    }
    close FILE;
```

AUTHOR AND COPYRIGHT

Copyright (c) 1997-2006 Tom Christiansen, Nathan Torkington, and other authors as noted. All rights reserved.

This documentation is free; you can redistribute it and/or modify it under the same terms as Perl itself.

Irrespective of its distribution, all code examples in this file are hereby placed into the public domain. You are permitted and encouraged to use this code in your own programs for fun or for profit as you see fit. A simple comment in the code giving credit would be courteous but is not required.