Regular Expression Recipes: A Problem-Solution Approach

NATHAN A. GOOD

Apress®

Regular Expression Recipes: A Problem-Solution Approach

ISBN (pbk): 1-59059-441-X

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Chris Mills
Technical Reviewer: Bill Johnson
Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis, Jason Gilmore, Chris Mills, Dominic Shakeshaft, Jim Sumser
Project Manager: Tracy Brown Collins
Copy Edit Manager: Nicole LeClerc
Copy Editor: Kim Wimpsett
Production Manager: Kari Brooks-Copony
Production Editor: Ellie Fountain
Compositor: Dina Quan
Proofreader: Linda Marousek
Cover Designer: Kurt Krames
Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013, and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States: phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders@springer-ny.com`, or visit `http://www.springer-ny.com`. Outside the United States: fax +49 6221 345229, e-mail `orders@springer.de`, or visit `http://www.springer.de`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Downloads section.

CHAPTER 1

Words and Text

This chapter includes recipes for performing some basic tasks of regular expressions, such as finding and replacing words and certain special characters such as tabs and high-ASCII characters.

Although this book isn't organized in levels of difficulty, this chapter includes many basic concepts that will make the rest of the book easier to follow. You won't have to go through this chapter to understand later ones, but it may help if you're new to regular expressions to make sure all the recipes in this chapter are easy to understand.

1-1. Finding Blank Lines

You can use this recipe for identifying blank lines in a file. Blank lines can contain spaces or tabs, or they can contain a combination of spaces and tabs. Variations on these expressions can be useful for stripping blank lines from a file.

Perl

```
#!/usr/bin/perl -w
use strict;

open( FILE, $ARGV[0] ) || die "Cannot open file!";

my $i = 0;

while ( <FILE> )
{
    $i++;
    next unless /^\s*$/;
    print "Found blank line at line $i\n";
}

close( FILE );
```

How It Works

The previous `\s` sequence represents a character class. A character class can match an entire category of characters, which `\s` does here even though it's a category that contains only tabs and spaces. Other character classes match far more examples, such as `\d`, which matches digits in most regular expression flavors. Character classes are shorthand ways of describing sets of characters.

Here's the expression broken down into parts:

`^` starts the beginning of the line, followed by . . .

`\s` any whitespace character (a tab or space) . . .

`*` zero or more times, followed by . . .

`$` the end of the line.

Shell Scripting

```
# grep '^[[:space:]]*$' filename
```

How It Works

The whitespace character class in `grep` is `[[:space:]]`. The `grep` expression, which is a Portable Operating System Interface (POSIX) expression, breaks down into the following:

`^`	at the beginning of the line, followed by . . .
`[[:space:]]`	any whitespace character (a tab or space) . . .
`*`	more than one time, but not required, followed by . . .
`$`	the end of the line.

Vim

```
/^\s*$
```

How It Works

To see how the Vim expression breaks down, refer to "How It Works" under the Perl example.

Variations

You can alter the `grep` recipe by adding the `-v` parameter to `grep`, which tells `grep` to print the lines that don't match the expression. The Perl and vi recipes have the advantage of containing a different character class that means "nonwhitespace." That character class is `\S`. Changing the expression in the Perl recipe to `^\S*$` will have the opposite effect as `^\s*$`.

1-2. Finding Words

You can use this recipe for finding single words in a block of text. The expression will find only complete words surrounded by spaces.

Perl

```
#!/usr/bin/perl -w
use strict;

open( FILE, $ARGV[0] ) || die "Cannot open file!";

my $i = 0;

while ( <FILE> )
{
    $i++;
    next unless /\bword\b/;
    print "Found word at line $i\n";
}

close( FILE );
```

How It Works

A special character class in Perl, `\b`, allows you to easily search for whole words. This is an advantage because without doing a whole bunch of extra work you can make sure that a search for `word`, for example, doesn't yield unexpected matches such as `sword`.

You can easily break the regular expression shown here into the following:

- `\b` a word boundary (a space or beginning of a line, or punctuation) . . .
- `w` a *w* followed by . . .
- `o` an *o*, followed by . . .
- `r` an *r*, then . . .
- `d` a *d*, and finally . . .
- `\b` a word boundary at the end of the word.

PHP

```
<html>
<head><title>1-2 Finding words</title></head>
<body>
<form action="recipe1-2.php" method="post">
```

```
<input type="text" name="str"
        value="<?php print $_POST['str'];?>" /><br />
<input type="submit" value="Find word" /><br /><br />
<?php
if ( $_SERVER['REQUEST_METHOD'] == "POST" )
{
    $str = $_POST['str'];
    if ( preg_match( "/\bword\b/", $str ) )
    {
        print "<b>Heh heh.  You said 'word'</b>";
    }
    else
    {
        print "<b>Nope.  Didn't find it.</b>";
    }
}
?>
</form>
</body>
</html>
```

How It Works

See "How It Works" under the Perl example for a walk-through of this expression.

Python

```
import re
r = re.compile( r'\bword\b', re.M )
if r.search( open( 'sample.txt' ).read( ) ) :
    print "I finally found what I'm looking for.",
else:
    print "\"word\"s not here, man.",
```

How It Works

Refer to "How It Works" under the Perl example to see how this expression breaks down.

Shell Scripting

```
# grep '\<word\>' filename
```

How It Works

The word boundary character class \b found in Perl-Compatible Regular Expressions (PCREs) doesn't exist in extended POSIX expressions. Instead, you can use the character class \< to refer to the left word boundary and the character class \> to refer to the word boundary at the end of the word. For example:

- \< a word boundary at the beginning of the word . . .
- w a *w* followed by . . .
- o an *o*, followed by . . .
- r an *r*, then. . .
- d a *d*, and finally . . .
- \> a word boundary at the end of the word.

Vim

```
/\<word\>
```

How It Works

See "How It Works" under the shell scripting example for an explanation of this expression.

1-3. Finding Multiple Words with One Search

You can use this recipe for finding more than one word in a block of text. This recipe assumes both words are whole words surrounded by whitespace.

Perl

```
#!/usr/bin/perl -w
use strict;

open( FILE, $ARGV[0] ) || die "Cannot open file!";

my $i = 0;

while ( <FILE> )
{
    $i++;
    next unless /\s+((moo)|(oink))\s+/;
    print "Found farm animal sound at line $i\n";
}

close( FILE );
```

How It Works

Starting outside working in, this expression searches for something that's surrounded by whitespace. For example:

- `\s` whitespace . . .
- `+` found one or more times . . .
- `(...)` followed by *something* . . .
- `\s` followed by whitespace . . .
- `+` that occurs one or more times.

The something here is another expression, `(moo)|(oink)`. This expression is as follows:

- `(` a group that contains . . .
- `m` an *m*, followed by . . .
- `o` an *o*, then . . .
- `o` an *o* . . .
- `)` the end of the group . . .
- `|` or . . .

(	a group that contains . . .
o	an *o*, followed by . . .
i	an *i*, then . . .
n	an *n*, followed by . . .
k	a *k* . . .
)	the end of a group.

PHP

```
<html>
<head><title>1-3 Finding multiple words with one search</title></head>
<body>
<form action="recipe1-3.php" method=”post”>
<input type="text" name="str"
    value="<?php print $_POST['str'];?>" /><br />
<input type="submit" value="Find words" /><br /><br />
<?php
if ( $_SERVER['REQUEST_METHOD'] == "POST" )
{
    $str = $_POST['str'];
    if ( preg_match( "/\s+((moo)|(oink))\s+/", $str ) )
    {
        print "<b>Found match:  '" . $str . "'</b><br/>";
    }
    else
    {
        print "<b>Did NOT find match: '" . $str . "'</b><br/>";
    }
}
?>
</form>
</body>
</html>
```

How It Works

See “How It Works” under the Perl example in this recipe.

Python

```
import re
r = re.compile( r'\s+((moo)|(oink))\s+', re.M )
if r.search( open( 'sample.txt' ).read( ) ):
```

```
    print "I spy a cow or pig.",
else:
    print "Ah, there's no cow or pig here.",
```

How It Works

See "How It Works" under the Perl example.

Shell Scripting

```
$ grep '[[:space:]]\+\(\(oink\)\|\(moo\)\)[[:space:]]\+' filename
```

How It Works

The POSIX expression uses the `[[:space:]]` character class to identify whitespace in the expression. Starting from the outside in, the expression is as follows:

`[[:space:]]`	whitespace . . .
`\+`	found one or more times . . .
`\(...\)`	a group that contains something . . .
`[[:space:]]`	whitespace . . .
`\+`	found one or more times.

The something here is the expression `\(oink\)\|\(moo\)`. This expression joins two groups of characters found inside `\(` and `\)` with the or operator `\|` to say, "The group of characters *o*, *i*, *n*, and *k* or the group of characters containing *m*, *o*, and *o*."

Vim

```
/\s\+\(\(oink\)\|\(moo\)\)\s\+
```

How It Works

See "How It Works" under the Perl example. Remember to escape the characters +, (,), and |.

1-4. Finding Variations on Words (John, Jon, Jonathan)

You can use this recipe for finding variations on a word with one search. This particular recipe searches for the strings `Jon Doe`, `John Doe`, or `Jonathan Doe`.

Perl

```
#!/usr/bin/perl -w
use strict;

my $mystr = $ARGV[0] || die "You must supply a parameter!\n";
if ( $mystr =~ /Joh?n(athan)? Doe/)
{
    print "Hello!  I've been looking for you!\n"
}
```

How It Works

This expression works by finding the common and optional parts of a word and searching based on them. `John`, `Jon`, and `Jonathan` are all similar. They start with *Jo* and have an *n* in them. The rest is the *h* in `John` or the *athan* ending in `Jonathan`. For example:

`J`	followed by . . .
`o`	then . . .
`h`	that is . . .
`?`	optional, followed by . . .
`n`	followed by . . .
`(...)`	a group of characters . . .
`?`	that may appear once, but isn't required, followed by . . .
<space>	a space, followed by . . .
`D`	then . . .
`o`	and finally . . .
`e`	at the end.

This group of characters is `athan`, which will let the expression match `Jonathan`. It may or may not appear as a whole part, so that's why it's grouped with parentheses and followed by `?`.

PHP

```
<html>
<head><title>1-4 Finding variations on words (John, Jon, Jonathan)</title></head>
<body>
<form action="recipe1-4.php" method="post">
<input type="text" name="str"
    value="<?php print $_POST['str'];?>" /><br />
<input type="submit" value="Where is John" /><br /><br />
<?php
if ( $_SERVER['REQUEST_METHOD'] == "POST" )
{
    $str = $_POST['str'];
    if ( preg_match( "/Joh?n(athan)? Doe/", $str ) )
    {
        print "<b>Found him:  '" . $str . "'</b><br/>";
    } else {
        print "<b>Nothing:   '" . $str . "'</b><br/>";
    }
}
?>
</form>
</body>
</html>
```

How It Works

See "How It Works" under the Perl example for a detailed explanation of this expression.

Python

```
#!/usr/bin/python
import re;
import sys;

nargs = len(sys.argv);

if nargs > 1:
    mystr = sys.argv[1];

    r = re.compile(r'Joh?n(athan)? Doe', re.M)
    if r.search(mystr):
        print 'Here\'s Johnny!';
    else:
        print 'Who?';

else:
    print 'I came here for an argument!';
```

How It Works

See "How It Works" under the Perl example for a breakdown of this recipe.

Shell Scripting

```
$ grep 'Joh\?n\(athan\)\? Doe' filename
```

How It Works

See "How It Works" under the Perl example, and remember to escape the characters ?, (, and).

Vim

```
/Joh\?n\(athan\)\? Doe
```

How It Works

See "How It Works" under the Perl expression. Remember to escape the characters ?, (, and); otherwise they will be taken literally instead of having their special meanings.

Variations

One variation on this recipe is using instead an expression such as that found in recipe 1-3, like `((Jon)|(John)|(Jonathan)) Doe`. Depending on the skills of your peers, this may be easy to use because it can be easier to read by someone else. Another variation on this is `((Jon(athan)?)|(John)) Doe`. Writing an elegant and fast regular expression is nice, but in these days processor cycles are often cheaper than labor. Make sure whatever path you choose will be the easiest to maintain by the people in your organization.

1-5. Finding Similar Words (Bat, Cat, Mat)

Slightly different from the previous recipe, this recipe focuses on using a character class to match a single letter and the word boundary character class.

Perl

```
#!/usr/bin/perl -w
use strict;

my $mystr = $ARGV[0] || die "You must supply a parameter!\n";
if ($mystr =~ /\b[bcm]at\b/)
{
    print "Found a word rhyming with hat\n";
}
else
{
    print "That's a negatory, big buddy.\n";
}
```

How It Works

The key to the expressions in these recipes finding whole words, even those that have no spaces around them and are on lines all by themselves, is the `\b` character class, which specifies a word boundary. A word boundary can be any whitespace that sets a word apart from others and can also include the beginning of a line and then end of a line. The other character class that's in this expression is `[bcm]`, which matches a single character that can be one of `b`, `c`, or `m`. So you can break the entire expression down like this:

`\b`	is a word boundary, followed by . . .
`[bcm]`	one of *b*, *c*, or *m*, followed by . . .
`a`	then . . .
`t`	and finally . . .
`\b`	a word boundary.

PHP

```
<html>
<head><title>1-5 Finding similar words (bat, cat, mat)</title></head>
<body>
<form action="recipe1-5.php" method="post">
<input type="text" name="value" value="<? print $_POST['value']; ?>"/><br/>
<input type="submit" value="Submit" /><br/><br/>
<?php
```

```
if ( $_SERVER[REQUEST_METHOD] == "POST" )
{
    $mystr = $_POST['value'];
    if ( preg_match( '/\b[bcm]at\b/', $mystr ) )
    {
        echo "Yes!<br/>";
    }
    else
    {
        echo "Uh, no.<br/>";
    }
}
?>
</form>
</body>
</html>
```

How It Works

Since this PHP example uses `preg_match()`, which uses PCRE, refer to "How It Works" under the Perl example to see this expression broken down.

Python

```
#!/usr/bin/python
import re;
import sys;

nargs = len(sys.argv);

if nargs > 1:
    mystr = sys.argv[1];

    r = re.compile(r'\b[bcm]at\b', re.M)
    if r.search(mystr):
        print 'I spy a bat, a cat, or a mat',
    else:
        print 'I don\'t spy nuttin\', honey',

else:
    print 'Come again?',
```

How It Works

Refer to "How It Works" under the Perl example to see how this expression breaks down into parts.

Shell Scripting

```
$ grep '\<[cbm]at\>' filename
```

How It Works

I explain this expression under "How It Works" in the Perl example, with the exception of the different word boundary. The PCRE word boundary character class `\b` is replaced with `\<` and `\>`.

Vim

```
/\<[cbm]at\>
```

How It Works

See "How It Works" under the shell scripting example.

Variations

A few variations on this expression exist, the most common of which is to use grouping and the or operator | instead of a character class to specify *b*, *c*, or *m*, as in `\b(b|c|m)at\b`.

1-6. Replacing Words

This recipe focuses on replacing complete words. It takes advantage of word anchors, which allow you to easily make sure you get an entire match.

Perl

```
#!/usr/bin/perl -w
use strict;

my $mystr = $ARGV[0] || die "You must supply a parameter!\n";
$mystr =~ s/\bfrick\b/frack/g;
print $mystr . "\n";
```

How It Works

This recipe has two expressions in it—the search expression and the replacement expression. Some extra characters in this expression make sure the match works on a whole word and not on a partial word. The word *frickster* will be left alone, for instance. Let's break down the search recipe:

- `\b` is a word boundary . . .
- `f` followed by . . .
- `r` then . . .
- `i` followed by . . .
- `c` followed by . . .
- `k` and finally . . .
- `\b` a word boundary.

Spelling out the word previously may seem a little redundant, but it really helps to distinguish a word from a sequence of characters that resembles a word. This is important to remember so you don't end up with matches and substitutions you aren't expecting.

PHP

```
<html>
<head><title>1-6 Replacing words</title></head>
<body>
<form action="recipe1-6.php" method="post">
<input type="text" name="value" value="<? print $_POST['value']; ?>" /><br />
<input type="submit" value="Replace word" /><br /><br />
<?php
if ( $_SERVER['REQUEST_METHOD'] == "POST" )
```

```
{
    $str = $_POST['value'];
    $newstr = preg_replace( "/\bfrick\b/", "frack", $str );
    print "<b>$newstr</b><br/>";
}
?>
</form>
</body>
</html>
```

How It Works

Refer to "How It Works" under the Perl example to see this search expression broken down.

Shell Scripting

```
$ sed 's/\<frick\>/frack/g' filename
```

How It Works

POSIX regular expressions support \< as a beginning word boundary and \> as a word boundary for the end of a word. Other than those differences, this expression is the same as the one explained in "How It Works" under the Perl example in this recipe.

Vim

```
:%s/\<frick\>/frack/g
```

How It Works

See "How It Works" under the shell scripting example for an explanation of this expression.

1-7. Replacing Everything Between Two Delimiters

This recipe replaces everything inside double quotes with a different string—in this case three asterisks. You can replace the double-quote delimiter with a different character to build expressions that will replace anything inside delimiters with another string.

Perl

```
#!/usr/bin/perl -w
use strict;

my $mystr = $ARGV[0] || die "You must supply a parameter!\n";
$mystr =~ s/\"[^\"]*\"/"***"/g;
print $mystr . "\n";
```

How It Works

This recipe shows off a simple version of the recipes later in Chapter 3. The expression is saying the following:

- `\"` is a quote, followed by . . .
- `[` a character class . . .
- `^` that isn't . . .
- `\"` another quote . . .
- `]` the end of the character class . . .
- `*` zero or more times . . .
- `\"` another quote appears.

Why not just use `\".*\"` and be done with it, you say? Well, that will work. Somewhat. The problem is that a quote (`"`) is matched by `.` (which matches anything). Say you have a string such as this:

```
my "string" is "water absorbent"
```

then you'll end up with this:

```
my "***"
```

PHP

```
<html>
<head><title>1-7 Replacing everything between two delimiters</title></head>
<body>
<form action="recipe1-7.php" method="post">
```

```
<input type="text" name="value" value="<? print $_POST['value']; ?>"/><br/>
<input type="submit" value="Submit" /><br/><br/>
<?php
if ( $_SERVER['REQUEST_METHOD'] == "POST" )
{
    $mystr = $_POST['value'];
    $mynewstr = ( ereg_replace( '"[^"]*"', '"***"', $mystr ) );
    print "<b>$mynewstr</b>";
}
?>
</form>
</body>
</html>
```

How It Works

Although this PHP example uses `ereg_replace`, which is POSIX and not PCRE, the expression is the same as the one explained in "How It Works" under the Perl example in this recipe.

One difference exists: the double quotes aren't escaped in this PHP example. The string is wrapped in single quotes, so you don't need to escape them. Quotes don't need to be escaped for regular expressions—they need to be escaped depending on the context in which they're used.

Shell Scripting

```
$ sed 's/\"[^\"]*\"/"***"/g' filename
```

How It Works

See "How It Works" under the Perl example in this recipe.

Vim

```
:%s/\"[^\"]*\"/"***"/g
```

How It Works

See "How It Works" under the Perl example in this recipe.

Variations

This recipe has plenty of room for getting fancy, depending on the flavor of regular expression you're using. Some, such as Perl, allow you to modify expressions as they're back referenced. The `\U` metacharacter, for instance, turns the back reference to uppercase. Here's an expression that will turn everything inside parentheses to uppercase using Perl: `s/\(([^\)]*)\)/(\U$1)/g`.

1-8. Replacing Tab Characters

This recipe is for replacing tab characters in a string with a different character. In this recipe, I use a pipe (|) to replace the tab.

Perl

```
#!/usr/bin/perl -w
use strict;

my $mystr = $ARGV[0] || die "You must supply a parameter!\n";

$mystr =~ s/\t/|/g;

print $mystr . "\n";
```

How It Works

Breaking the recipe down yields simply the following:

`\t` is a tab, replaced by . . .

`|` a pipe character.

PHP

```
<html>
<head><title>1-8 Replacing tab characters</title></head>
<body>
<form action="recipe1-8.php" method="post">
<input type="submit" value="Submit" /><br/><br/>
<?php
if ( $_SERVER['REQUEST_METHOD'] == "POST" )
{

    $myfile = @fopen( "recipe1-8.txt", "r" )
        or die ("Cannot open file $myfile");

    while ( $line = @fgets( $myfile, 1024 ) )
    {
        $mynewstr = ereg_replace( "\t", "|", $line);
        echo "$mynewstr<br/>";
    }
    fclose($myfile);
}
?>
```

```
</form>
</body>
</html>
```

How It Works

See "How It Works" under the Perl example in this recipe. The POSIX expression, shown here, supports the same character class for tab characters as the PCRE expressions.

Python

```
#!/usr/bin/python
import re;
import sys;

nargs = len(sys.argv);

if nargs > 1:
    mystr = sys.argv[1]
    r = re.compile(r'\t', re.M)

    returnstr = r.sub( '|', open( mystr ).read( ) );

    print returnstr,

else:
    print 'Come again?',
```

How It Works

See "How It Works" under the Perl example in this recipe for a description of this expression.

Shell Scripting

```
$ sed 's/\t/\|/g' filename
```

How It Works

You'll find this `sed` expression here broken down in "How It Works" under the Perl expression.

Vim

```
:%s/\t/|/g
```

How It Works

See "How It Works" under the Perl example for an explanation of the expression shown here. The g option will tell the Vim editor to replace all occurrences found on the line.

Variations

Since this is such a simple recipe, it has an extensive number of variations. You could replace the character class representing a tab with other character classes, especially the `\s` character class in Perl. The Perl variation `s/\s/,/g` would replace each instance of whitespace with a comma.

One variation on the previous recipe is to use a qualifier after the character class to replace more than one instance of a tab at once. For instance, if you want to replace two tabs with four spaces, you could use something such as `s/\t{2}/  /g` in the Perl recipe.

1-9. Testing Complexity of Passwords

This recipe tests a string to make sure it has a combination of letters and numbers in the string. This recipe heavily uses look-arounds, which aren't supported by every flavor of regular expressions. For instance, the POSIX regular expressions in PHP don't support them, so use PCRE in PHP (`preg_match`) to accomplish this task.

Perl

```
#!/usr/bin/perl -w
use strict;

my $mystr = $ARGV[0] || die "You must supply a parameter!\n";

if ( $mystr =~ /^(?=.*[A-Z])(?=.*[a-z])(?=.*[0-9]).{7,15}$/ )
{
    print "Good password!\n";
}
else
{
    print "Dude!  That's way too easy.\n";
}
```

How It Works

The look-arounds in the expression make it seem more complicated than it really is. At the heart of the expression, without the look-arounds, is the following:

`^`	at the beginning of the line, followed by . . .
`.`	any character
`{7,15}`	found anywhere 7 to 15 times, followed by . . .
`$`	the end of the line.

Now let's add the look-arounds, which are grouped by the expressions `(?=` and `)`. Three of them exist in this expression: `(?=.*[A-Z])`, `(?=.*[a-z])`, and `(?=.*[0-9])`. These look-arounds say, "This expression must appear somewhere to the right." In this case, that's to the right of `^`, which is the line anchor that anchors the beginning of the line. The first look-ahead matches anything followed by a capital letter (`[A-Z]`), the second matches anything followed by a lowercase letter (`[a-z]`), and the third matches anything followed by a number (`[0-9]`).

PHP

```
<html>
<head><title>1-9 Testing complexity of passwords</title></head>
<body>
<form action="recipe1-9.php" method="post">
<input type="text" name="str"
    value="<?php print htmlspecialchars($_POST['str']);?>" /><br />
<input type="submit" value="Test password" /><br /><br />
<?php
if ( $_SERVER['REQUEST_METHOD'] == "POST" )
{
    $str = $_POST['str'];
    if ( preg_match( "/^(?=.*[A-Z])(?=.*[a-z])(?=.*[0-9]).{7,15}$/", $str ) )
    {
        print "<b>Good password!:  '" .
            htmlspecialchars($str) . "'</b><br/>";
    } else {
        print "<b>That's not good enough:  '" .
            htmlspecialchars($str) . "'</b><br/>";
    }
}
?>
</form>
</body>
</html>
```

How It Works

See "How It Works" under the Perl example for an explanation of this expression.

Vim

```
/^\(.*[A-Z]\)\@=\(.*[a-z]\)\@=\(.*[0-9]\)\@=.\{7,15\}$
```

How It Works

Vim supports look-arounds, but their syntax is so different from those in Perl that it's worth explaining the first group here as an example.

- `^` at the beginning of the line . . .
- `\(` a group that contains . . .
- `.` any character . . .
- `*` found zero or more times, followed by . . .
- `[` a character class that contains . . .
- `A-Z` *A* through *Z* . . .
- `]` the end of the character class . . .
- `\)` the end of the group . . .
- `\@=` the preceding expression is a look-ahead . . .
- . . . and so on.

Variations

This one has many variations, but probably the most notable is to make the expression more complicated by adding a fourth look-ahead group that matches punctuation characters, such as `(?=.*[!@#$%^&*()])`.

Another variation is to use a different character class for the number, such as `\d` if the flavor of regular expressions you're using supports it.