How to use regular expressions in PHP

Validate user input, parse user input and file contents, and reformat strings

Skill Level: Intermediate

Nathan A. Good (mail@nathanagood.com)
Consultant
Alliance of Computer Professionals

10 Jan 2006

Regular expressions can provide a powerful way to work with text. Using regular expressions, you can do complex validation of user input, parse user input and file contents, and reformat strings. PHP provides simple methods that let you use POSIX and PCRE regular expressions. This tutorial discusses the differences between POSIX and PCRE, and how you can use regular expressions and PHP V5.

Section 1. Before you start

Learn what to expect from this tutorial, and how to get the most out of it.

About this tutorial

Regular expressions can provide a powerful way to work with text. Using regular expressions, you can do complex validation of user input, parse user input and file contents, and reformat strings.

Objectives

This tutorial will get you up and running with PHP's regular expressions by focussing on simple methods that let you use POSIX and PCRE regular expressions. We will discuss the differences between POSIX and PCRE, and how you can use regular expressions and PHP V5. You'll learn how, when, and why to use regular expressions.

System requirements

You'll be able to complete this tutorial on any Microsoft® Windows® or UNIX®-like system (including Mac OS X and Linux®) that has PHP installed. Because we're covering what's built into PHP, you won't need anything more than PHP installed on your system.

Section 2. Getting started

What are regular expressions?

A couple years ago, I was doing some interesting validation for an input box on a Web form. Users would enter a phone number on the form. The phone number, exactly as typed, would then be printed on the user's advertising. The requirements stated that a U.S. phone number could be entered several ways: (555) 555-5555 was OK, 555-555-5555 was also fine, but 555-5555 was not acceptable.

You may wonder why we didn't strip out all the non-numerical characters and count to see whether a total of 10 characters remained. That approach would have worked, but it wouldn't have stopped users from entering something like !555?333-3333.

From a Web developer's perspective, the situation presented an interesting challenge. I could have written routines that checked for each of the different formats, but I wanted a solution that would be flexible if the user later approved the format 555.555.5555.

This is where regular expressions (regexes) came in. I had cut and pasted them in applications before, but I'd never had a reason to understand their syntax. Regexes, it turns out, are a lot like mathematical expressions. When you look at an expression like **2x2=4**, you usually think "Two times two equals four." Regular expressions are very similar. By the end of this article, you'll see a regex like **^b$** and say to yourself, "The beginning of the line followed by b, then the end of the line." Not only that but you'll learn how easy it is to use regexes in PHP.

When to use regexes

You should use regexes to do searches and make replacements when you have rules to follow, but you don't necessarily have the exact characters that need to be found or replaced. For instance, in the earlier phone number example, the users defined rules that dictated the format of the phone number entered, but not the digits contained in the phone number. The same is true for a lot of user input. U.S. state abbreviations may be limited to two capital letters between A and Z. Or, using a

regular expression, you can easily limit text or user input on a form to alphabetical letters, regardless of case and length.

When not to use regexes

Regexes are powerful, but they have a couple of drawbacks. One is the skill set required to read and write expressions. If you decide to incorporate regular expressions into your application, you should comment them completely; then, if someone needs to change the expression, they can do so without breaking functionality. In addition, if you're new to regular expressions, you may find them difficult to debug.

To avoid these difficulties, don't use regexes where an easier built-in function will work just as well.

Section 3. POSIX vs. PCRE

PHP supports two implementations of regexes: Portable Operating System Implementation (POSIX) and Perl-Compatible Regular Expressions (PCREs). These implementations offer different features, but they're equally easy to use in PHP. The regex style you use will depend a lot on your previous experience with regexes and what you're comfortable using. There is some evidence that PCREs are a little faster than POSIX expressions, but the difference won't be noticeable in most applications.

The examples here include the syntax of each regex method in comments. In the function syntax, the regex is the `regex` argument, and the string being searched is `string`. Arguments that appear in brackets are optional; this primer won't discuss all of them.

Section 4. Regex syntax

Although POSIX and PCRE implementations differ in their support for some features and character classes, they share a syntax. Each regex is made up of one or more characters, special characters (sometimes called *metacharacters*), character classes, and groups of characters.

POSIX and PCRE use the same wildcard -- a way in regex to say, "anything goes." The wildcard character is a period or dot (`.`). To look for a literal period or dot, use the escape character `\`: `\.`. The same goes for the other special characters discussed in the following sections, such as line anchors and qualifiers. If a character has a special meaning in a regex, it must be escaped to take on its literal

meaning.

Line anchors are special metacharacters that match the beginning or end of a line, but don't capture any text (see Table 1). For example, if a line begins with the letter a, a line anchor in the expression ^a doesn't capture the letter a, but does match the beginning of the line.

Table 1. Line anchors

Anchor	Description
^	Matches the beginning of a line
$	Matches the end of a line

Qualifiers apply to the expressions immediately preceding them (see Table 2). Using qualifiers, you can specify how many times an expression can be found in a search. For instance, the expression a+ looks for the letter a found one or more times.

Table 2. Qualifiers

Qualifier	Description
?	The expression before the qualifier can be found optionally once
+	The expression before the qualifier can be found one or more times.
*	The expression before the qualifier can be found any number of times, including zero.
{n}	The expression before the qualifier can be found exactly n times.
{n,m}	The expression before the qualifier can be found between n and m times.

The ability to capture text and refer to it in replacements and searches is an extremely useful feature of regexes (see Table 3). Using captures, you can perform searches that look for doubled words and closing HTML and XML tags. If you use captures when you're replacing, you can put the text you found back into the replacement string. A later example shows how to replace e-mail addresses with hyperlinks.

Table 3: Grouping and capturing

Character Class	Description
( )	Groups characters and can capture text

POSIX character classes

POSIX regular expressions comply with standards that make them usable with many regex implementations (see Table 4). For instance, if you write a POSIX regex, you can use it in PHP, use it with the `grep` command, and use it with many editors that support regexes.

Table 4. POSIX character classes

Character	Description
`[:alpha:]`	Matches alphanumeric characters
`[:digit:]`	Matches any number
`[:space:]`	Matches any white space

Section 5. POSIX matches

Two functions search strings using POSIX regexes: `ereg()` and `eregi()`.

ereg()

The `ereg()` method searches a string for the given regex. If no match is found, it returns 0, so you can test it like this:

Listing 1. ereg() method

```
<?php
$phonenbr="555-555-5555";
// Syntax is ereg( regex, string [,
out_captures_array])
if (ereg("[-[:digit:]]{12}", $phonenbr)) {
    print("Found match!\n");
} else {
    print("No match found!\n");
}
?>
```

The regex `[-[:digit:]]{12}` looks for 12 characters that are digits or hyphens. It's a little sloppy for use with a phone number, and it can instead be written like this: `^[0-9]{3}-[0-9]{3}-[0-9]{4}$`. (In regex, `[0-9]` and `[:digit:]` are essentially the same thing; you may prefer `[0-9]` because it's shorter.) The alternative expression is much more exact. It looks for the beginning of the line (^), followed by a group of three digits (`[0-9]{3}`), a hyphen (`-`), another group of three digits, another hyphen, a group of four digits, and then the end of the line ($). When you're crafting an expression, it helps to be able to anticipate the type of data to be searched or replaced with the expression because then you know how far to go with the regex in terms of complexity.

eregi()

The `eregi()` method is similar to `ereg()`, except that it's case-insensitive. It returns an integer that contains the length of the match found, but you'll most likely use it inside conditional statements like the following:

Listing 2. eregi() method

```
<?php
$str="Hello World!";
// Syntax is ereg( regex, string [,
out_captures_array])
if (eregi("hello", $str)) {
    print("Found match!\n");
} else {
    print("No match found!\n");
}
?>
```

When you execute this example, it prints `Found match!` because *hello* is found as long as you do the search with case ignored. If you were using `ereg`, the match would fail.

Section 6. POSIX replacements

The methods `ereg_replace()` and `eregi_replace()` make replacements in text and feature POSIX regexes.

ereg_replace()

You can use the `ereg_replace()` method to make case-sensitive replacements in POSIX regex syntax. The following example demonstrates how to replace an e-mail address in a string with a hyperlink:

Listing 3. ereg_replace() method

```
<?php
$origstr = "My e-mail address is:
first.last@example.com";
// Syntax is: ereg_replace( regex,
replacestr, string )
$newstr = \
ereg_replace("([.[:alpha:][:digit:]]+@[.[:alpha:][:digit:]]+)",
    "<a href=\"mailto:\\1\">\\1</a>",
$origstr);
print("$newstr\n");
?>
```

This is an incomplete version of a regex to match e-mail addresses, but it shows that `ereg_replace()` is more powerful than a regular replacement function like `str_replace()`. When you use a regex, you define rules by which to do searches, instead of searching for literal characters.

eregi_replace()

With the exception of ignoring case, the `eregi_replace()` function is identical to `ereg_replace()`:

Listing 4. eregi_replace() function

```
<?php
$origstr = "1 BANANA, 2 banana, 3 Banana";
// Syntax is: eregi_replace( regex,
replacestr, string )
$newstr = eregi_replace("banana", "pear",
$origstr);
print("New string is: '$newstr'\n");
?>
```

This example replaces `banana` with `pear`, regardless of case.

Section 7. PCRE character classes

Because PCRE syntax supports shorter character classes and more features, it can be more powerful than POSIX syntax. Table 5 lists some of the character classes supported in PCREs, but can't be found in POSIX expressions.

Table 5. PCRE character classes

Character Class	Description
`\b`	A word boundary; finds the end or beginning of the word
`\d`	Matches any number
`\s`	Matches any white space, such as tabs or spaces
`\t`	Matches a tab character
`\w`	Matches alphanumeric characters

PCRE matches

The PCRE match functions in PHP are similar to the POSIX match functions, but one feature makes them tricky to use if you're used to POSIX expressions: The PCRE functions require the expression to begin and end with a delimiter. In most examples, the delimiter is a single `/` found at the beginning and end of the expression, inside the quotes. It's important to remember that this delimiter isn't part of the expression.

After the last delimiter in a PCRE, you can add a modifier that changes the behavior of the regex. The `i` modifier, for example, makes the regex case-insensitive. This is a key difference from the POSIX methods, where you call a different method based on your need for case-insensitivity.

preg_grep()

The `preg_grep()` method returns an array that contains all the items in another

array where a match was found with the regex. It can be useful if you have a large collection of values and wish to search them to find the matches. Here's an example:

Listing 5. preg_grep() method

```
<?php
$array = array( "1", "3", "ABC", "XYZ", "42" );
// Syntax is preg_grep( regex, inputarray );
$grep_array = preg_grep("/^\d+$/", $array);
print_r($grep_array);
?>
```

In this example, the regex `^\d+$` finds all elements in the array that contain one or more digits (`\d+`) between the beginning of the line (^) and the end of the line ($).

preg_match()

The `preg_match()` function finds matches in strings using PCREs. It requires two parameters: the regex and the string. Optionally, you can provide an array that will be filled with the matches, flags that let you tailor the behavior of the matching, and the position in the string from which to begin finding matches (`offset`). Here's an example:

Listing 6. offset method

```
<?php
$string = "abcdefgh";
$regex = "/^[a-z]+$/i";
// Syntax is preg_match(regex, string, [, out_matches [, flags [, offset]]]);
if (preg_match($regex, $string)) {
    printf("Pattern '%s' found in string '%s'\n", $regex, $string);
} else {
    printf("No match found in string '%s'!\n", $string);
}
?>
```

This example uses the regex `^[a-z]+$` to search for any letter a through z found one or more times (`[a-z]+`) between the beginning (^) and the end ($) of the line.

preg_match_all()

The `preg_match_all()` function builds an array of all the matches found in the string. The following example builds an array of all the words in a sentence:

Listing 7. preg_match_all() function

```
<?php
$string = "The quick red fox jumped over the lazy brown dog";
$re = "/\b\w+\b/";
// Syntax is preg_match_all( regex, string, return_array [, flags [, offset]])
preg_match_all($re, $string, $arrayout);
print_r($arrayout);
?>
```

This regex, `\b\w+\b`, looks for word characters found one or more times (`\w+`)

between word boundaries (\b). Each word is put into an array element in the output array $arrayout.

Section 8. PCRE replacements

Making PCRE replacements in PHP is similar to making POSIX replacements, with the exception that you use preg_replace(), instead of ereg_replace() and eregi_replace().

preg_replace()

The preg_replace() function makes replacements using PCREs. It requires as parameters the regex, the replacement expression, and the original string. Optionally, you can provide the maximum number or replacements you want to make and a variable that will be filled with the number of replacements made. Here's an example:

Listing 8. preg_replace() function

```
<?php
$orig_string = "5555555555";
printf("Original string is '%s'\n",
$orig_string);
$re = "/^(\d{3})(\d{3})(\d{4})$/";
// Syntax is preg_replace( regex,
replacement, string \
[, limit [, out_count]] );
$new_string = preg_replace($re, "(\\1)
\\2-\\3", $orig_string);
printf("New string is '%s'\n", $new_string);
?>
```

This example gives a quick demonstration of how to capture parts of text and use *back references*, such as \\1. These back references insert whatever text is matched by the groups in parentheses; in this case, \\1 matches the first group, (\d{3}).

You could slice and dice the phone number in the example using substr, but with a few changes to the string, it would become increasingly difficult to rely on substr to reliably capture the correct text.

If the string can be (555)5555555, you can modify the expression to ^(?(\d{3}))?(\d{3})(\d{4})$ to look for optional parentheses.

Section 9. Summary

PHP offers two syntaxes for regexes: POSIX and PCRE. This tutorial offered a high-level overview of the main functions used in PHP for POSIX and PCRE regex support.

Using regexes, you can define rules with which to do powerful searching and replacing -- above and beyond literal searches and replacements.

Resources

Learn

- Regular-Expressions.info provides information about regexes.
- PHP: Regular Expression Functions (Perl-Compatible) - Manual is the PHP online documentation that covers PCREs.
- Regular Expression Functions (POSIX Extended) is the PHP online documentation on POSIX regexes.
- Visit developerWorks' PHP project resources to learn more about PHP.
- For a series of developerWorks tutorials on learning to program with PHP, see "Learning PHP, Part 1," Part 2, and Part 3.
- Stay current with developerWorks technical events and webcasts.
- Visit the developerWorks Open source zone for extensive how-to information, tools, and project updates to help you develop with open source technologies and use them with IBM's products.

Get products and technologies

- Download the latest version of PHP from PHP.net.
- Regular Expression Library has a large repository of regexes.
- Order the SEK for Linux, a two-DVD set containing the latest IBM trial software for Linux from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.
- Innovate your next open source development project with IBM trial software, available for download or on DVD.

Discuss

- Get involved in the developerWorks community by participating in developerWorks blogs.

About the author

Nathan A. Good
Nathan A. Good is an author, software engineer, and system administrator in the Twin Cities in Minnesota. His books include *PHP 5 Recipes: A Problem-Solution Approach* by Lee Babin, et al (Apress, 2005), *Regular Expression Recipes for Windows Developers: A Problem-Solution Approach* (Apress, 2005), *Regular Expressions: A Problem-Solution Approach* (Apress, 2005), and *Professional Red Hat Enterprise Linux 3* by Kapil Sharma et al (Wrox, 2004).