PHP Regular Expressions

What is a Regular Expression?

A regular expression is a sequence of characters that forms a search pattern. When you search for data in a text, you can use this search pattern to describe what you are searching for.

A regular expression can be a single character, or a more complicated pattern.

Regular expressions can be used to perform all types of text search and text replace operations.

Syntax

In PHP, regular expressions are strings composed of delimiters, a pattern and optional modifiers.

$exp = "/w3schools/i";

In the example above, `/` is the **delimiter**, *w3schools* is the **pattern** that is being searched for, and `i` is a **modifier** that makes the search case-insensitive.

The delimiter can be any character that is not a letter, number, backslash or space. The most common delimiter is the forward slash (/), but when your pattern contains forward slashes it is convenient to choose other delimiters such as # or ~.

Regular Expression Functions

PHP provides a variety of functions that allow you to use regular expressions. The `preg_match()`, `preg_match_all()` and `preg_replace()` functions are some of the most commonly used ones:

Function	Description
preg_match()	Returns 1 if the pattern was found in the string and 0 if not
preg_match_all()	Returns the number of times the pattern was found in the string, which may also be 0
preg_replace()	Returns a new string where matched patterns have been replaced with another string

---

Using preg_match()

The `preg_match()` function will tell you whether a string contains matches of a pattern.

Example

Use a regular expression to do a case-insensitive search for "w3schools" in a string:

```php
<?php
$str = "Visit W3Schools";
$pattern = "/w3schools/i";
echo preg_match($pattern, $str); // Outputs 1
?>
```

Using preg_match_all()

The `preg_match_all()` function will tell you how many matches were found for a pattern in a string.

Example

Use a regular expression to do a case-insensitive count of the number of occurrences of "ain" in a string:

```php
<?php
$str = "The rain in SPAIN falls mainly on the plains.";
$pattern = "/ain/i";
echo preg_match_all($pattern, $str); // Outputs 4
?>
```

Using preg_replace()

The `preg_replace()` function will replace all of the matches of the pattern in a string with another string.

Example

Use a case-insensitive regular expression to replace Microsoft with W3Schools in a string:

```php
<?php
$str = "Visit Microsoft!";
$pattern = "/microsoft/i";
echo preg_replace($pattern, "W3Schools", $str); // Outputs "Visit W3Schools!"
?>
```

Regular Expression Modifiers

Modifiers can change how a search is performed.

Modifier	Description

i	Performs a case-insensitive search
m	Performs a multiline search (patterns that search for the beginning or end of a string will match the beginning or end of each line)
u	Enables correct matching of UTF-8 encoded patterns

Regular Expression Patterns

Brackets are used to find a range of characters:

Expression	Description
[abc]	Find one character from the options between the brackets
[^abc]	Find any character NOT between the brackets
[0-9]	Find one character from the range 0 to 9

Metacharacters

Metacharacters are characters with a special meaning:

Metacharacter	Description
|	Find a match for any one of the patterns separated by | as in: cat|dog|fish
.	Find just one instance of any character
^	Finds a match as the beginning of a string as in: ^Hello
$	Finds a match at the end of the string as in: World$
\d	Find a digit
\s	Find a whitespace character
\b	Find a match at the beginning of a word like this: \bWORD, or at the end of a word like this: WORD\b
\uxxxx	Find the Unicode character specified by the hexadecimal number xxxx

Quantifiers

Quantifiers define quantities:

Quantifier	Description
n+	Matches any string that contains at least one *n*
n*	Matches any string that contains zero or more occurrences of *n*
n?	Matches any string that contains zero or one occurrences of *n*
n{x}	Matches any string that contains a sequence of *X n*'s
n{x,y}	Matches any string that contains a sequence of X to Y *n*'s
n{x,}	Matches any string that contains a sequence of at least X *n*'s

Note: If your expression needs to search for one of the special characters you can use a backslash (\ ) to escape them. For example, to search for one or more question marks you can use the following expression: $pattern = '/\?+/';

Grouping

You can use parentheses ( ) to apply quantifiers to entire patterns. They also can be used to select parts of the pattern to be used as a match.

Example

Use grouping to search for the word "banana" by looking for *ba* followed by two instances of *na*:

```
<?php
$str = "Apples and bananas.";
$pattern = "/ba(na){2}/i";
echo preg_match($pattern, $str); // Outputs 1
?>
```

Complete RegExp Reference

For a complete reference, go to our Complete PHP Regular Expression Reference.

The reference contains descriptions and examples of all Regular Expression functions.

https://www.w3schools.com/php/php_regex.asp

PHP Regular Expressions

In this tutorial you will learn how regular expressions work, as well as how to use them to perform pattern matching in an efficient way in PHP.

What is Regular Expression

Regular Expressions, commonly known as "**regex**" or "**RegExp**", are a specially formatted text strings used to find patterns in text. Regular expressions are one of the most powerful tools available today for effective and efficient text processing and manipulations. For example, it can be used to verify whether the format of data i.e. name, email, phone number, etc. entered by the user was correct or not, find or replace matching string within text content, and so on.

PHP (version 5.3 and above) supports Perl style regular expressions via its `preg_` family of functions. Why Perl style regular expressions? Because Perl (*Practical Extraction and Report Language*) was the first mainstream programming language that provided integrated support for regular expressions and it is well known for its strong support of regular expressions and its extraordinary text processing and manipulation capabilities.

Let's begin with a brief overview of the commonly used PHP's built-in pattern-matching functions before delving deep into the world of regular expressions.

Function	What it Does
`preg_match()`	Perform a regular expression match.
`preg_match_all()`	Perform a global regular expression match.
`preg_replace()`	Perform a regular expression search and replace.
`preg_grep()`	Returns the elements of the input array that matched the pattern.
`preg_split()`	Splits up a string into substrings using a regular expression.
`preg_quote()`	Quote regular expression characters found within a string.

Note: The PHP `preg_match()` function stops searching after it finds the first match, whereas the `preg_match_all()` function continues searching until the end of the string and find all possible matches instead of stopping at the first match.

Regular Expression Syntax

Regular expression syntax includes the use of special characters (do not confuse with the HTML special characters). The characters that are given special meaning within a regular expression, are: `.` `*` `?` `+` `[` `]` `(` `)` `{` `}` `^` `$` `|` `\`. You will need to backslash these characters whenever you want to use them literally. For example, if you want to match ".", you'd have to write `\.`. All other characters automatically assume their literal meanings.

The following sections describe the various options available for formulating patterns:

Character Classes

Square brackets surrounding a pattern of characters are called a character class e.g. `[abc]`. A character class always matches a single character out of a list of specified characters that means the expression `[abc]` matches only a, b or c character.

Negated character classes can also be defined that match any character except those contained within the brackets. A negated character class is defined by placing a caret (`^`) symbol immediately after the opening bracket, like this `[^abc]`.

You can also define a range of characters by using the hyphen (`-`) character inside a character class, like `[0-9]`. Let's look at some examples of character classes:

RegExp	What it Does
`[abc]`	Matches any one of the characters a, b, or c.
`[^abc]`	Matches any one character other than a, b, or c.
`[a-z]`	Matches any one character from lowercase a to lowercase z.
`[A-Z]`	Matches any one character from uppercase a to uppercase z.
`[a-Z]`	Matches any one character from lowercase a to uppercase Z.
`[0-9]`	Matches a single digit between 0 and 9.
`[a-z0-9]`	Matches a single character between a and z or between 0 and 9.

The following example will show you how to find whether a pattern exists in a string or not using the regular expression and PHP `preg_match()` function:

Example

Run this code »

```
<?php
$pattern = "/ca[kf]e/";
$text = "He was eating cake in the cafe.";
if(preg_match($pattern, $text)){
    echo "Match found!";
} else{
    echo "Match not found.";
}
?>
```

Similarly, you can use the `preg_match_all()` function to find all matches within a string:

Example

Run this code »

```
<?php
$pattern = "/ca[kf]e/";
$text = "He was eating cake in the cafe.";
$matches = preg_match_all($pattern, $text, $array);
echo $matches . " matches were found.";
?>
```

Tip: Regular expressions aren't exclusive to PHP. Languages such as Java, Perl, Python, etc. use the same notation for finding patterns in text.

Predefined Character Classes

Some character classes such as digits, letters, and whitespaces are used so frequently that there are shortcut names for them. The following table lists those predefined character classes:

Shortcut	What it Does
`.`	Matches any single character except newline `\n`.
`\d`	matches any digit character. Same as `[0-9]`
`\D`	Matches any non-digit character. Same as `[^0-9]`
`\s`	Matches any whitespace character (space, tab, newline or carriage return character). Same as `[ \t\n\r]`
`\S`	Matches any non-whitespace character. Same as `[^ \t\n\r]`
`\w`	Matches any word character (definned as a to z, A to Z,0 to 9, and the underscore). Same as `[a-zA-Z_0-9]`
`\W`	Matches any non-word character. Same as `[^a-zA-Z_0-9]`

The following example will show you how to find and replace space with a hyphen character in a string using regular expression and PHP `preg_replace()` function:

Example

Run this code »

```
<?php
$pattern = "/\s/";
$replacement = "-";
$text = "Earth revolves around\nthe\tSun";
// Replace spaces, newlines and tabs
echo preg_replace($pattern, $replacement, $text);
echo "<br>";
// Replace only spaces
echo str_replace(" ", "-", $text);
?>
```

Repetition Quantifiers

In the previous section we've learnt how to match a single character in a variety of fashions. But what if you want to match on more than one character? For example, let's say you want to find out words containing one or more instances of the letter p, or words containing at least two p's, and so on. This is where quantifiers come into play. With quantifiers you can specify how many times a character in a regular expression should match.

The following table lists the various ways to quantify a particular pattern:

RegExp	What it Does
`p+`	Matches one or more occurrences of the letter p.
`p*`	Matches zero or more occurrences of the letter p.
`p?`	Matches zero or one occurrences of the letter p.
`p{2}`	Matches exactly two occurrences of the letter p.
`p{2,3}`	Matches at least two occurrences of the letter p, but not more than three occurrences of

RegExp	What it Does
	the letter p.
p{2,}	Matches two or more occurrences of the letter p.
p{,3}	Matches at most three occurrences of the letter p

The regular expression in the following example will splits the string at comma, sequence of commas, whitespace, or combination thereof using the PHP `preg_split()` function:

Example

Run this code »

```
<?php
$pattern = "/[\s,]+/";
$text = "My favourite colors are red, green and blue";
$parts = preg_split($pattern, $text);

// Loop through parts array and display substrings
foreach($parts as $part){
    echo $part . "<br>";
}
?>
```

Position Anchors

There are certain situations where you want to match at the beginning or end of a line, word, or string. To do this you can use anchors. Two common anchors are caret (^) which represent the start of the string, and the dollar ($) sign which represent the end of the string.

RegExp	What it Does
^p	Matches the letter p at the beginning of a line.
p$	Matches the letter p at the end of a line.

The regular expression in the following example will display only those names from the names array which start with the letter "J" using the PHP `preg_grep()` function:

Example

Run this code »

```
<?php
$pattern = "/^J/";
$names = array("Jhon Carter", "Clark Kent", "John Rambo");
$matches = preg_grep($pattern, $names);

// Loop through matches array and display matched names
foreach($matches as $match){
    echo $match . "<br>";
}
?>
```

Pattern Modifiers

A pattern modifier allows you to control the way a pattern match is handled. Pattern modifiers are placed directly after the regular expression, for example, if you want to search for a pattern in a case-insensitive manner, you can use the `i` modifier, like this: `/pattern/i`. The following table lists some of the most commonly used pattern modifiers.

Modifier	What it Does
`i`	Makes the match case-insensitive manner.
`m`	Changes the behavior of `^` and `$` to match against a newline boundary (i.e. start or end of each line within a multiline string), instead of a string boundary.
`g`	Perform a global match i.e. finds all occurrences.
`o`	Evaluates the expression only once.
`s`	Changes the behavior of `.` (dot) to match all characters, including newlines.
`x`	Allows you to use whitespace and comments within a regular expression for clarity.

The following example will show you how to perform a global case-insensitive search using the `i` modifier and the PHP `preg_match_all()` function.

Example

Run this code »

```
<?php
$pattern = "/color/i";
$text = "Color red is more visible than color blue in daylight.";
$matches = preg_match_all($pattern, $text, $array);
echo $matches . " matches were found.";
?>
```

Similarly, the following example shows how to match at the beginning of every line in a multi-line string using `^` anchor and `m` modifier with PHP `preg_match_all()` function.

Example

Run this code »

```
<?php
$pattern = "/^color/im";
$text = "Color red is more visible than \ncolor blue in daylight.";
$matches = preg_match_all($pattern, $text, $array);
echo $matches . " matches were found.";
?>
```

Word Boundaries

A word boundary character (`\b`) helps you search for the words that begins and/or ends with a pattern. For example, the regexp `/\bcar/` matches the words beginning with the pattern car, and would match cart, carrot, or cartoon, but would not match oscar.

Similarly, the regexp `/car\b/` matches the words ending with the pattern car, and would match scar, oscar, or supercar, but would not match cart. Likewise, the `/\bcar\b/` matches the words beginning and ending with the pattern car, and would match only the word car.

The following example will highlight the words beginning with car in bold:

Example

Run this code »

```
<?php
$pattern = '/\bcar\w*/';
$replacement = '<b>$0</b>';
$text = 'Words begining with car: cart, carrot, cartoon. Words ending with car:
scar, oscar, supercar.';
echo preg_replace($pattern, $replacement, $text);
?>
```

We hope you have understood the basics of regular expression. To learn how to validate form data using regular expression, please check out the tutorial on PHP Form Validation.

https://www.tutorialrepublic.com/php-tutorial/php-regular-expressions.php

Regex Introduction

Regex is a text-processing language! They are fast and efficient!

What is Regex?

Regex (or RegExp) stands for **Regular Expressions**, which is fast and efficient way to match patterns inside a string. In this tutorial, we will learn about how to create regular expressions and how to use them in PHP functions.

Regex can be used for processes like **text search**, **text search and replace**, **input validation**, etc.

Regex can be a simple character or a complicated pattern. All of these are defined under certain rules.

Regex in PHP

PHP supports the widely used syntax for regex: PCRE (Perl Compatible Regular Expressions) by default.

In PHP, PCRE functions are prefixed by *preg_*.

PHP Regex Replace Example

```
<?php
$str = 'Hello World';
$regex = '/\s/';
echo preg_replace($regex, '', $str);
```

In this example, the first white space in "Hello World" is removed. So, it will output "HelloWorld". Let's see what *$regex* and *preg_replace()* does.

- *preg_replace()* searches for a string (using regex patterns) and replaces it with another string.
- *$regex* helps to search the string.
- */* signs at the beginning and ending of *$regex* indicates the start and the end of the regex. They are called **delimiters**.
- *\s* is a single expression. It matches any white space characters. They are called **character types**.
- Then, the match is replaced with ″. So, the whitespace gets removed.

We will learn more about PCRE syntax in the next chapter.

Regex PCRE Syntax

Delimiters

When using PCRE functions in PHP, it is required to enclose the pattern by **delimiters**.

A delimiter can be any character except following.

- alphanumeric characters (letters and numbers)
- backslash \
- whitespace

Often used delimiters: /, #, ~

```
/hello world/
#hello world#
~hello world~
```

You can also use brackets as delimiters where the opening and closing brackets represents the beginning and ending respectively.

Valid bracket style delimiters: *()*, *[]*, *{}*, <>

```
(hello world)
[hello world]
{hello world}
<hello world>
```

If the delimiter need to be matched inside the pattern, you should escape the delimiter by \ (back slash). If the delimiter appears several times in a pattern, it is better to use another delimiter.

- bad: */http:\/\//*
- good: *#http://#*

But, brackets style delimiters do not need to be escaped.
Note: You will learn more about escaping with \ in next sub topics of this chapter.

Meta Characters

Meta characters are helpers for complicated patterns. They do not stand for themselves, but empower the regex to do more.

There are two types of meta characters because of their different behavior when they are inside and outside **square brackets** *[]*.

Outside square brackets

Meta Character	Description	Example	Explained
\	Used for escape the next character	.\.	. matches any character while \.matches dot. The full regex will match any character followed by a dot.
^	Matches the start of a string (or line, in multi-line mode)	*^hello*	Matches the word **hello** which is in the beginning of a string.
$	Matches the end of a string (or end of line, in multi-line mode)	*hello$*	Matches the word **hello** which is in the ending of a string.

Meta Character	Description	Example	Explained
.	Matches any character except line break	*h.llo*	Matches **h**, then, any character and then, **llo** (hello, hollo, hallo, ... are matched)
|	OR conditional operator	*hello| hallo*	Matches either **hello** or **hallo**
()	Defines a sub-pattern	*hel(lo|p)*	Matches either **hello** or **help**
[]	Defines a character class	*hel[a-z]o*	Matches **hel**, then, any lowercase letter and then, **o** (hello, helko, helpo)
{}	Defines a repetition pattern	*hel{1,3}o*	Matches **he**, then, 1 to 3 **l** characters and then, **o** (helo, hello, helllo)
?	Matches a pattern 0 or 1 time	*hell?o*	Matches **hel**, then, either **l** or nothing, then **o** (hello, helo)
*	Matches a pattern 0 or more times	*hel*o*	Matches **he**, then 0 or more **l** characters and then, **o**. (heo, helo, hello, helllo)
+	Matches a pattern 1 or more times	*hel+o*	Matches **he**, then 1 or more **l** characters and then, **o**. (helo, hello, helllo)

A part of pattern that is inside square brackets is called a **character class**. In a character class there are only three meta characters.

Inside square brackets

Meta Character	Description	Example	Explained
^	Negates the class, if it is the first character	*hel[^abc]o*	Matches **hel**, then, any character except **a**, **b** and **c** and finally, **o**. (hello, helko, helwo)
\	Used for escape the next character	*[\^abc]*	Matches any of **a**, **b**, **c** or ^. Here, ^ has no special meaning as it is escaped.
-	Matches a range of characters	*hel[a-z]o*	Matches **hel**, then, any a to z character (lowercase letter) and finally, **o**. (hello, helko, helwo).

Escape Sequences

If you match meta characters in a pattern, you should **escape** them with \

- * - Works as a meta character (Matches the previous character 0 or more times)
- * - Matches *

Backslash \ is the basic escape character in PHP. It has several uses:

- Backslash can convert a **meta character** to a **normal character**.
- Backslash can convert a **normal character** to a **special character**.
- Backslash can convert a **normal character** to a **generic character type**.

Escaping Meta Characters

As we learned previously, * is a meta character. If we need to match * in a string, we have to escape it.

In this example, * characters are replaced with *$*.

Escaping Meta Characters Example

```
<?php
$str = 'H*H*H*H*';
$regex = '/\*/';
echo preg_replace($regex, '$', $str); // returns H$H$H$H$
```

Tip: \ is also a meta character. To match \, use \\!

Escaping Normal Characters

Escaping normal characters can make two results, special characters and generic character types.

1. Special Characters

Among plenty of special characters, we will use following characters frequently.

Special Character	Meaning
\t	Tab Space
\n	New Line
\r	Carriage Return

In the following example, *\n* (New line) is replaced with - .

Special Characters Example

```
<?php
$str =
"Hello
World"
;
$regex = '/\n/';
echo preg_replace($regex, ' - ', $str); // returns H$H$H$H$
```

2. Generic Character Types

A Generic character type is a set of characters which is really helpful in regex patterns.

Generic character type	Description
\d	Any decimal digit
\D	Any character except decimal digits
\w	Any word character
\W	Any non-word character
\s	Any whitespace character
\S	Any non-whitespace character

Generic Character Types Example

```
<?php
$str = '06:45:12';
$regex = '/\d/';
echo preg_replace($regex, 'n', $str); // returns "nn:nn:nn"
```

```
echo "<br>";

$str = 'How are you?';
$regex = '/\s/';
echo preg_replace($regex, '', $str); // returns "Howareyou?"
```

There are two more useful escape sequences which we will use in later tutorials.

Escape Sequence	Description
\b	Word boundary
\B	Not word boundary

Modifiers

Modifiers can modify the way that the pattern is working. As an example, if you match "hello", "Hello" will not match. To perform a case-insensitive regex, we can use the *i* modifier.

Modifier	Description
i	Makes the match case-insensitive.
m	Activates the multi-line mode.
s	Allows . to match all the characters, including new lines.

Modifiers Example

```
<?php
$str = 'Hello World';

// outputs "Hello World"
echo preg_replace('/hello/', 'Hi', $str);

echo '<br>';

// outputs "Hi World"
echo preg_replace('/hello/i', 'Hi', $str);
```

In PHP, modifiers are defined after the ending delimiter of the pattern.

PHP PREG Functions

PHP has several functions to work with PCRE regular expressions we learned in the last chapter.

Among them, we have already learned about *preg_replace()* function in PHP. Now we are going to discuss about two of *preg_* functions which we will use in PHP forms chapter.

- preg_match
- preg_replace

preg_match

preg_match() function performs a regular expression match.

preg_match() Syntax

preg_match(pattern, input string, variable to save results)

Let's see some examples.

1. Find the word "dogs" in a string

```
<?php
$str = 'She had stood between the pack of wild dogs and what they wanted';
if (preg_match('/\bdogs\b/i', $str)) {
        echo 'Word dogs was found';
} else {
        echo 'Word dogs was not found';
}

echo "<br>"; // line break

// hotdogs will not match for dogs, because regex matches for word boundaries (\
b)
$str = 'Linda returned with two hotdogs and handed one to each of them';
if (preg_match('/\bdogs\b/i', $str)) {
        echo 'Word dogs was found';
} else {
        echo 'Word dogs was not found';
}
```

\b matches a word boundary and the *i* modifier makes the match case-insensitive.

2. Getting the domain name out of a URL

```
<?php
$url = 'https://developer.hyvor.com/tutorials';
preg_match('#^(?:\w+://)?([^/]+)#i', $url, $matches);

echo $matches[1];
```

Note: Any match inside parentheses is captured. It will be saved in *$matches* from the index 1. (*$matches[1]* is the first capture). To avoid capturing of a parenthesized sub pattern, *?:* should be appended to the parentheses. (*(?:\w+://)*)

How this regular expressions works:

- We use # as delimiters.
- ^ matches for the beginning of the string.
- *(?:\w+://)* is a non-capturing parenthesized sub pattern, which matches 1 or more *\w* (word characters) and *://*. (It matches the protocol, which we do not need to capture to get the domain.)
- *?* after the non-capturing sub pattern tells that main pattern should match even the protocol is not defined (0 or 1).
- *([^/]+)* is a capturing sub pattern which matches 1 or more non-front-slash (/) characters, which is the domain.
- So, any character after a \ is not matched.

Note: *$matches[0]* saves the full match.

3. Using named sub patterns

In this example, we will use named sub patterns. (*?P<book>* to name the sub pattern as book)

```
<?php
$str = 'The Mother: Maxim Gorky - 1906';
preg_match('/(?P<book>[\w\s]+): (?P<author>[\w\s]+) - (?P<year>\d+)/', $str,
$matches);

echo 'Book: ' . $matches['book'] . '<br>';
echo 'Author: ' . $matches['author'] . '<br>';
echo 'Year: ' . $matches['year'];
```

preg_replace

preg_replace() function performs a regular expression match and replace.

preg_replace Syntax

preg_replace(pattern, replace with, input string)

In the following example, all the digits are replaced with *$* signs.

preg_replace Example

```
<?php
$str = 'October 22, 2000';
echo preg_replace('/\d/', '$', $str);
```