Step 1 - Regular expressions basics

PHP regular expressions seems to be a quite complicated area especially if you are not an experienced Unix user. Historically regular expressions were originally designed to help working with strings under Unix systems.

Using regular expressions you can easy find a pattern in a string and/or replace it if you want. This is a very powerful tool in your hand, but be careful as it is slower than the standard string manipulation functions.

Regular expression types
There are 2 types of regular expressions:

- POSIX Extended
- Perl Compatible

The ereg, eregi, ... are the POSIX versions and preg_match, preg_replace, ... are the Perl version. It is important that using Perl compatible regular expressions the expression should be enclosed in the delimiters, a forward slash (/), for example. However this version is more powerful and faster as well than the POSIX one.

The regular expressions basic syntax
To use regular expressions first you need to learn the syntax of the patterns. We can group the characters inside a pattern like this:

- Normal characters which match themselves like hello
- Start and end indicators as ^ and $
- Count indicators like +,*,?
- Logical operator like |
- Grouping with {},(),[]

An example pattern to check valid emails looks like this:

Code: ^[a-zA-Z0-9._-]+@[a-zA-Z0-9-]+\.[a-zA-Z.]{2,5}$
The code to check the email using Perl compatible regular expression looks like this:

Code:
```
$pattern = "/^[a-zA-Z0-9._-]+@[a-zA-Z0-9-]+\.[a-zA-Z.]{2,5}$/";
$email   = "jim@demo.com";
if (preg_match($pattern,$email)) echo "Match";
else echo "Not match";
```

And very similar in case of POSIX extended regular expressions:

Code:
```
$pattern = "^[a-zA-Z0-9._-]+@[a-zA-Z0-9-]+\.[a-zA-Z.]{2,5}$";
$email   = "jim@demo.com";
if (eregi($pattern,$email)) echo "Match";
else echo "Not match";
```

Now let's see a detailed pattern syntax reference:

Regular expression (pattern)	**Match (subject)**	**Not match (subject)**	**Comment**
world	Hello world	Hello Jim	Match if the pattern is present anywhere in the subject
^world	world class	Hello world	Match if the pattern is present at the beginning of the subject

world$	Hello world	world class	Match if the pattern is present at the end of the subject
world/i	This WoRLd	Hello Jim	Makes a search in case insensitive mode
^world$	world	Hello world	The string contains only the "world"
world*	worl, world, worlddd	wor	There is 0 or more "d" after "worl"
world+	world, worlddd	worl	There is at least 1 "d" after "worl"
world?	worl, world, worly	wor, wory	There is 0 or 1 "d" after "worl"
world{1}	world	worly	There is 1 "d" after "worl"
world{1,}	world, worlddd	worly	There is 1 ore more "d" after "worl"
world{2,3}	worldd, worlddd	world	There are 2 or 3 "d" after "worl"
wo(rld)*	wo, world, worldold	wa	There is 0 or more "rld" after "wo"
earth|world	earth, world	sun	The string contains the "earth" or the "world"
w.rld	world, wwrld	wrld	Any character in place of the dot.
^.{5}$	world, earth	sun	A string with exactly 5 characters
[abc]	abc, bbaccc	sun	There is an "a" or "b" or "c" in the string
[a-z]	world	WORLD	There are any lowercase letter in the string
[a-zA-Z]	world, WORLD, Worl12	123	There are any lower- or uppercase letter in the string
[^wW]	earth	w, W	The actual character can not be a "w" or "W"

Step 2 - Complex regular expression examples

Now as you know the theory and basic syntax of PHP regular expressions it's time to create and analyze some more complex cases.

User name check with regular expression
First start with a user name check. In case of a registration form you may want to control available user names a bit. Let's suppose you don't want to allow any special character in the name except "_.-" and of course letters and numbers. Besides this you may want to control the length of the user name to be between 4 and 20.

First we need to define the available characters. This can be realised with the following code:

[a-zA-Z0-9_.-]

After that we need to limit the number of characters with the following code: {4,20}

At least we need to put it together: ^[a-zA-Z-0-9_.-]{4,20}$

In case of Perl compatible regular expression surround it with '/'. At the end the PHP code looks like this:

Code:
```
$pattern  = '/^[a-zA-Z0-9_.-]{4,20}$/';
$username = "this.is.a-demo_-";
if (preg_match($pattern,$username)) echo "Match";
else echo "Not match";
```

Check hexadecimal color codes with regular expression
A hexadecimal color code looks like this: #5A332C or you can use a short form like #C5F. In both case it starts with a # and follows with exactly 3 or 6 numbers or letters from a-f.

So the first it starts as: ^#

the following character range is: [a-fA-F0-9]

and the length can be 3 or 6. The complete pattern is the following:

^#(([a-fA-F0-9]{3}$)|([a-fA-F0-9]{6}$))

Here we use an or statement first check the #123 form and then the #123456 form. At the end the PHP code looks like this:

Code:
```
$pattern = '/^#(([a-fA-F0-9]{3}$)|([a-fA-F0-9]{6}$))/';
$color   = "#1AA";
if (preg_match($pattern,$color)) echo "Match";
else echo "Not match";
```

Email check with regular expression
At least let's see how we can check an email address with regular expressions. First take a careful look at the following example emails:

- john.demo@demo.com
- john@demo.us
- john_123.demo_.name@demo.info

What we can see is that the @ is a mandatory element in an email. Besides this there must be some character before and some after it. More precisely there must be a valid domain name after the @.

So the first part must be a string with letters a numbers or some special characters like _-. In pattern we can write it as follows:

^[a-zA-Z0-9_.-]+

The domain name always have a let's say name and tld. The tld is the .com, .us. .info and the name can be any string with valid characters. It means that the domain pattern looks like this:

[a-zA-Z0-9-]+\.[a-zA-Z.]{2,4}$

Now we only need to put together the 2 parts with the @ and get the complete pattern:

^[a-zA-Z0-9._-]+@[a-zA-Z0-9-]+\.[a-zA-Z.]{2,5}$

The PHP code looks like this:

Code:
```
$pattern = '/^[a-zA-Z0-9._-]+@[a-zA-Z0-9-]+\.[a-zA-Z.]{2,5}$/';
$email   = "john123.demo_.name@demo.info";
if (preg_match($pattern,$email)) echo "Match";
else echo "Not match";
```

MySQL Regular Expressions Cheat Sheet

MySQL Regular Expressions
Regular Expressions in MySQL are used within the REGEXP and RLIKE sections of WHERE clauses in the selection of records for display, update or deletion.

Operator Type	Examples	Description	
Literal Characters Match a character exactly	a A y 6 % @	Letters, digits and many special characters match exactly	
		\$ \^ \+ \\ \?	Prec othe spec char with canc rege spec mea
		\n \t \r	Liter line, retu
		\cJ \cG	Cont code
		\xa3	Hex for a char
Anchors and assertions	^	Field starts with	
		$	Field with
		[[:<:]]	Wor with
		[[:>:]]	Wor with
Character groups any 1 character from the group	[aAeEiou]	any character listed from [to]	
		[^aAeEiou]	any char exce aAeI
		[a-fA-F0-9]	any char (0 to to f)
		.	any char all
		[[:space:]]	any

			char (spa or \t
		[[:alnum:]]	any alph c ch (lett digit
Counts apply to previous element	+	1 or more ("some")	
		*	0 or ("pe som
		?	0 or ("pe a")
		{4}	exac
		{4,}	4 or
		{4,8}	betw and
		Add a ? after any count to turn it sparse (match as few as possible) rather than have it default to greedy	
Alternation	|	either, or	
Grouping	( )	group for count and save to variable	

Introduction to Regular Expressions in PHP
Introduction to Regular Expressions in PHP

Regular expressions were created by an American mathematician named Stephen Kleene. PHP supports two different types of regular expressions: POSIX-extended and Perl-Compatible Regular Expressions (PCRE). The PCRE functions are more powerful than the POSIX ones, and faster too, so we will concentrate on them.

Some Important Terms
Let Me Start With an example *.txt [Find all files with extension txt]

Metacharacter
A metacharacter is a special character that the regex engine will use to apply "rules" for

Eg: *.txt

Literal text
Literal text is actual "text" that you are using to be matched in your regular expression.
Eg: *.txt

Character Class []
A character class is something that lets you tell the regex engine what characters (literal text) that you would like to allow at that point in the regular expression
Eg: [Jj]ohn

Anchor
An anchor is actually a 'metacharacter', but It doesn't actually match text, only the position of text.
Eg: /^[Jj]ohn$/

Whitespace -
Whitespace is actually "literal text", but it is empty space. a string of:
Eg: $str = " "; is comprised of 'whitespace'.

Common Metacharacters and Anchors

Caret symbol (^)
A caret (^) character at the beginning of a regular expression indicates that it must match the beginning of the string.
Eg: ^z searches for a part that begins with z.

Dollar Symbol($)
A dollar sign ($) is used to match strings that end with the given pattern
Eg: z$ searches for a part that ends with z.

Dot (.)
A Dot metacharacter matches any single character except newline (\).
Eg: pattern h.t matches hat, hothit, hut, h7t, etc

The vertical pipe (|)
The vertical pipe (|) metacharacter is used for alternatives in a regular expression. It behaves much like a logical OR operator and you should use it if you want to construct a pattern that matches more than one set of characters. For instance, the pattern Utah|Idaho|Nevada matches strings that contain "Utah" or "Idaho" or "st="on"Nevada". Parentheses give us a way to group sequences. For example, (Nant|b)ucket matches "Nantucket" or "bucket". Using parentheses to group together characters for alternation is called grouping.

Other Meta Characters.
The metacharacters +, *, ?, and {} affect the number of times a pattern should be matched.

Plus (+)
Match one or more of the preceding expression

The + (plus) matches the previous character 1 or more times, for example, tre+ will find tree and tread but not trough.

Asterisk/ Star(*)
Match zero or more of the preceding expression

The * (asterisk or star) matches the preceding character 0 or more times, for example, tre* will find tree and tread and trough.

Question Mark(?)
Match zero or one of the preceding expression

The ? (question mark) matches the preceding character 0 or 1 times only, for example, colou?r will find both color and colour.

Curly braces {}
{1} means "match exactly 1 occurrences of the preceding expression", with one
{1,} means "match 1 or more occurrences of the preceding expression",
{1,5} means "match the previous character if it occurs at least 1 times, but no more than 5 times".

{n}

Matches the preceding character n times exactly, for example, to find a local phone number we could use [0-9]{3}-[0-9]{4} which would find any number of the form 123-4567.

Note: The - (dash) in this case, because it is outside the square brackets, is a **literal**. Value is enclosed in braces (curly brackets).

{n,m}

Matches the preceding character at least n times but not more than m times, for example,

'ba{2,3}b' will find 'baab' and 'baaab' but NOT 'bab' or 'baaaab'. Values are enclosed in braces (curly brackets).

Note: Using these metacharacters and a pair of (parentheses) you can create a number of different and complex search patterns. Here are some examples of different search patterns :

abc{3}	searches for abccc
(abc){3}	searches for abcabcabc
on|off	searches for onff or ooff
(on)|(off)	searches for on or off

Quick Reference

^z	searches for a part that begins with z.
z$	searches for a part that ends with z.
z+	searches for at least one z in a row.
z?	searches for zero or one z.
(yz)	searches for yz grouped together.
y|z	searches for y or z.
z{3}	searches for zzz.
z{1,}	searches for z or zz or zzz and so on...
z{1,3}	searches for z or zz or zzz only.

Other metacharacter type searches include...

.	searches for ANY character or letter.
[a-z]	searches for any lowercase letter.
[A-Z]	searches for any uppercase letter.
[0-9]	searches for any digit 0 to 9.
\	escapes the next character.
\n	new line.
\t	tab.

Note: If you want to match a literal metacharacter in a pattern, you have to escape it with a backslash

ereg_replace — Replace regular expression

Example#1

```
$str= "This is a test";
echo str_replace(" is", " was", $str);
echo ereg_replace("( )is", "\\1was", $str);
echo ereg_replace("(( )is)", "\\2was", $str);
//prints "This was a test" three times:
```

Example#2

```
$s = "Coding PHP is fun.";
$pattern = "(.*)PHP(.*)";
$replacement = " They say \\1other languages\\2";
print ereg_replace($pattern, $replacement, $s);
?>
```

	Output: They say Coding other languages is fun. **Explanation:** "PHP" is replaced with "other languages", and the sentence is changed a little, using \1 and \2 to access the parts within parentheses.
Example#3 **Replace URLs with links** $text = ereg_replace("[[:alpha:]]+://[^<>[:space:]]+[[:alnum:]/]", "\\0", $text);	

eregi — Case insensitive regular expression match

Example#1

```
= 'XYZ';
if (eregi('z', $string)) {
echo "'$string' contains a 'z' or 'Z'!";
}
```

ereg — Regular expression match

Example#1
The following code snippet takes a date in ISO format (YYYY-MM-DD) and prints it in DD.MM.YYYY format:
if (ereg
("([0-9]{4})-([0-9]{1,2})-([0-9]{1,2})",
$date, $regs)) { echo
"$regs[3].$regs[2].$regs[1]";
} else { echo "Invalid date format: $date"; }
	Example#2
	Check if string only contains letters and numbers.
	if (ereg("[^A-Za-z0-9]", $string)) {
	echo "Error: String can only contain letters and numbers!";
	exit();
	}

Example#3
This is intended to validate fully specified (international) phone numbers without forcing the user to use the full international format and giving them maximum reasonable flexibility including an optional extension number.
Allows numbers plus any of: space():.ext,+-
Example: +44(0)113 249-0442 ext:1234
	Example#3
	The code matches any combination of the allowed character set.
	$phoneNumber="+44(0)113 249-0442 ext:1234";
	$regex="[0-9 ():.ext,+-]{".strlen($phoneNumber)."}";
	if(ereg($regex,$phoneNumber)){
	echo "ok";
	}else {
	echo "invalid phone number";
	}

PHP Regex Cheat Sheet

Special Sequences • \w - Any "word" character (a-z 0-9 _) • \W - Any non "word" character • \s - Whitespace (space, tab CRLF) • \S - Any non whitepsace character • \d - Digits (0-9) • \D - Any non digit character	**Meta Characters** • ^ - Start of subject (or line in multiline mode) • $ - End of subject (or line in multiline mode) • [- Start character class definition •] - End character class definition

• . - (Period) – Any character except newline	• | - Alternates, eg (a|b) matches a or b • (- Start subpattern •) - End subpattern • \ - Escape character
Quantifiers • n* - Zero or more of n • n+ - One or more of n • n ? - Zero or one occurrences of n • {n} - n occurrences exactly • {n,} - At least n occurrences • {,m} - At most m occurrences • {n,m} - Between n and m occurrences (inclusive)	**Pattern Modifiers** • i - Case Insensitive • m - Multiline mode - ^ and $ match start and end of lines • s - Dotall - . class includes newline • x - Extended– comments and whitespace • e - preg_replace only – enables evaluation of replacement as PHP code • S - Extra analysis of pattern • U - Pattern is ungreedy • u - Pattern is treated as UTF-8
Point based assertions • \b - Word boundary • \B - Not a word boundary • \A - Start of subject • \Z - End of subject or newline at end • \z - End of subject • \G - First matching position in subject	**Assertions** • (?=) - Positive look ahead assertion foo(?=bar) matches foo when followed by bar • (?!) - Negative look ahead assertion foo(?!bar) matches foo when not followed by bar • (?<=) - Positive look behind assertion (?<=foo)bar matches bar when preceded by foo • (? - Negative look behind assertion (? • (?>) - Once-only subpatterns (?>\d+)bar Performance enhancing when bar not present • (?(x)) - Conditional subpatterns • (?(3)foo|fu)bar - Matches foo if 3rd subpattern has matched, fu if not • (?#) - Comment (?# Pattern does x y or z)