Regular Expressions (Perl-Compatible)

Introduction

The syntax for patterns used in these functions closely resembles Perl. The expression must be enclosed in the delimiters, a forward slash (/), for example. Any character can be used for delimiter as long as it's not alphanumeric or backslash (\). If the delimiter character has to be used in the expression itself, it needs to be escaped by backslash. Since PHP 4.0.4, you can also use Perl-style (), {}, [], and <> matching delimiters. See Pattern Syntax for detailed explanation.

The ending delimiter may be followed by various modifiers that affect the matching. See Pattern Modifiers.

PHP also supports regular expressions using a POSIX-extended syntax using the POSIX-extended regex functions.

Note
This extension maintains a global per-thread cache of compiled regular expressions (up to 4096).

Warning
You should be aware of some limitations of PCRE. Read » http://www.pcre.org/pcre.txt for more info.

Installing/Configuring

Requirements

No external libraries are needed to build this extension.

Installation

Beginning with PHP 4.2.0 these functions are enabled by default. You can disable the pcre functions with *--without-pcre-regex*. Use *--with-pcre-regex=DIR* to specify DIR where PCRE's include and library files are located, if not using bundled library. For older versions you have to configure and compile PHP with *--with-pcre-regex[=DIR]* in order to use these functions.

The Windows version of PHP has built-in support for this extension. You do not need to load any additional extensions in order to use these functions.

Note
As of PHP 5.3.0 this extension cannot be disabled and is therefore always present. It is still possible to build against an external PCRE library by using *--with-pcre-regex=DIR*

Runtime Configuration

The behaviour of these functions is affected by settings in *php.ini*.

PCRE Configuration Options

Name	Default	Changeable	Changelog
pcre.backtrack_limit	"100000"	PHP_INI_ALL	Available since PHP 5.2.0.
pcre.recursion_limit	"100000"	PHP_INI_ALL	Available since PHP 5.2.0.

For further details and definitions of the PHP_INI_* constants, see the php.ini directives.

Here's a short explanation of the configuration directives.

pcre.backtrack_limit integer
PCRE's backtracking limit.

pcre.recursion_limit integer
PCRE's recursion limit. Please note that if you set this value to a high number you may consume all the available process stack and eventually crash PHP (due to reaching the stack size limit imposed by the Operating System).

Resource Types

This extension has no resource types defined.

Predefined Constants

The constants below are defined by this extension, and will only be available when the extension has either been compiled into PHP or dynamically loaded at runtime.

PREG constants

constant	description
PREG_PATTERN_ORDER	Orders results so that $matches[0] is an array of full pattern matches, $matches[1] is an array of strings matched by the first parenthesized subpattern, and so on. This flag is only used with preg_match_all().
PREG_SET_ORDER	Orders results so that $matches[0] is an array of first set of matches, $matches[1] is an array of second set of matches, and so on. This flag is only used with preg_match_all().
PREG_OFFSET_CAPTURE	See the description of **PREG_SPLIT_OFFSET_CAPTURE**. This flag is available since PHP 4.3.0.
PREG_SPLIT_NO_EMPTY	This flag tells preg_split() to return only non-empty pieces.
PREG_SPLIT_DELIM_CAPTURE	This flag tells preg_split() to capture parenthesized expression in the delimiter pattern as well. This flag is available since PHP 4.0.5.
PREG_SPLIT_OFFSET_CAPTURE	If this flag is set, for every occurring match the appendant string offset will also be returned. Note that this changes the return values in an array where every element is an array consisting of the matched string at offset 0 and its string offset within subject at offset 1. This flag is available since PHP 4.3.0 and is only used for preg_split().
PREG_NO_ERROR	Returned by preg_last_error() if there were no errors. Available since PHP 5.2.0.
PREG_INTERNAL_ERROR	Returned by preg_last_error() if there was an internal PCRE error. Available since PHP 5.2.0.

PREG_BACKTRACK_LIMIT_ERROR	Returned by preg_last_error() if backtrack limit was exhausted. Available since PHP 5.2.0.
PREG_RECURSION_LIMIT_ERROR	Returned by preg_last_error() if recursion limit was exhausted. Available since PHP 5.2.0.
PREG_BAD_UTF8_ERROR	Returned by preg_last_error() if the last error was caused by malformed UTF-8 data (only when running a regex in UTF-8 mode). Available since PHP 5.2.0.
PREG_BAD_UTF8_OFFSET_ERROR	Returned by preg_last_error() if the offset didn't correspond to the begin of a valid UTF-8 code point (only when running a regex in UTF-8 mode). Available since PHP 5.3.0.
PCRE_VERSION	PCRE version and release date (e.g. "7.0 18-Dec-2006"). Available since PHP 5.2.4.

Examples

Example #1 - Examples of valid patterns

- */<\/\w+>/*
- *|(\d{3})-\d+|Sm*
- */^(?i)php[34]/*
- *{^\s+(\s+)?$}*

Example #2 - Examples of invalid patterns

- */href='(.*)'* - missing ending delimiter
- */\w+\s*\w+/J* - unknown modifier 'J'
- *1-\d3-\d3-\d4|* - missing starting delimiter

PCRE Patterns

Pattern Modifiers

The current possible PCRE modifiers are listed below. The names in parentheses refer to internal PCRE names for these modifiers. Spaces and newlines are ignored in modifiers, other characters cause error.

i (PCRE_CASELESS)
If this modifier is set, letters in the pattern match both upper and lower case letters.

m (PCRE_MULTILINE)
By default, PCRE treats the subject string as consisting of a single "line" of characters (even if it actually contains several newlines). The "start of line" metacharacter (^) matches only at the start of the string, while the "end of line" metacharacter ($) matches only at the end of the string, or before a terminating newline (unless *D* modifier is set). This is the same as Perl. When this modifier is set, the "start of line" and "end of line" constructs match immediately following or immediately before any newline in the subject string, respectively, as well as at the very start and end. This is equivalent to Perl's /m modifier. If there are no "\n" characters in a subject string, or no occurrences of ^ or $ in a pattern, setting this modifier has no effect.

s (PCRE_DOTALL)
If this modifier is set, a dot metacharacter in the pattern matches all characters, including newlines. Without it, newlines are excluded. This modifier is equivalent to Perl's /s modifier. A negative class such as [^a] always matches a newline character, independent of the setting of this modifier.

x (PCRE_EXTENDED)
If this modifier is set, whitespace data characters in the pattern are totally ignored except when escaped or inside a character class, and characters between an unescaped # outside a character class and the next newline character, inclusive, are also ignored. This is equivalent to Perl's /x modifier, and makes it possible to include comments inside complicated patterns. Note, however, that this applies only to data characters. Whitespace characters may never appear within special character sequences in a pattern, for example within the sequence (?(which introduces a conditional subpattern.

e (PREG_REPLACE_EVAL)
If this modifier is set, preg_replace() does normal substitution of backreferences in the replacement string, evaluates it as PHP code, and uses the result for replacing the search string. Single quotes, double quotes, backslashes and NULL chars will be escaped by backslashes in substituted backreferences. Only preg_replace() uses this modifier; it is ignored by other PCRE functions.

A (PCRE_ANCHORED)
If this modifier is set, the pattern is forced to be "anchored", that is, it is constrained to match only at the start of the string which is being searched (the "subject string"). This effect can also be achieved by appropriate constructs in the pattern itself, which is the only way to do it in Perl.

D (PCRE_DOLLAR_ENDONLY)
If this modifier is set, a dollar metacharacter in the pattern matches only at the end of the subject string. Without this modifier, a dollar also matches immediately before the final character if it is a newline (but not before any other newlines). This modifier is ignored if *m* modifier is set. There is no equivalent to this modifier in Perl.

S
When a pattern is going to be used several times, it is worth spending more time analyzing it in order to speed up the time taken for matching. If this modifier is set, then this extra analysis is performed. At present, studying a pattern is useful only for non-anchored patterns that do not have a single fixed starting character.

U (PCRE_UNGREEDY)
This modifier inverts the "greediness" of the quantifiers so that they are not greedy by default, but become greedy if followed by "?". It is not compatible with Perl. It can also be set by a (?U) modifier setting within the pattern or by a question mark behind a quantifier (e.g..*?).

X (PCRE_EXTRA)
This modifier turns on additional functionality of PCRE that is incompatible with Perl. Any backslash in a pattern that is followed by a letter that has no special meaning causes an error, thus reserving these combinations for future expansion. By default, as in Perl, a backslash followed by a letter with no special meaning is treated as a literal. There are at present no other features controlled by this modifier.

J (PCRE_INFO_JCHANGED)
The (?J) internal option setting changes the local PCRE_DUPNAMES option. Allow duplicate names for subpatterns.

u (PCRE_UTF8)
This modifier turns on additional functionality of PCRE that is incompatible with Perl. Pattern strings are treated as UTF-8. This modifier is available from PHP 4.1.0 or greater on Unix and from PHP 4.2.3 on win32. UTF-8 validity of the pattern is checked since PHP 4.3.5.

Pattern Syntax

Description

The PCRE library is a set of functions that implement regular expression pattern matching using the same syntax and semantics as Perl 5, with just a few differences (see below). The current implementation corresponds to Perl 5.005.

Differences From Perl

The differences described here are with respect to Perl 5.005.

- By default, a whitespace character is any character that the C library function isspace() recognizes, though it is possible to compile PCRE with alternative character type tables. Normally isspace() matches space, formfeed, newline, carriage return, horizontal tab, and vertical tab. Perl 5 no longer includes vertical tab in its set of whitespace characters. The \v escape that was in the Perl documentation for a long time was never in fact recognized. However, the character itself was treated as whitespace at least up to 5.002. In 5.004 and 5.005 it does not match \s.
- PCRE does not allow repeat quantifiers on lookahead assertions. Perl permits them, but they do not mean what you might think. For example, (?!a){3} does not assert that the next three characters are not "a". It just asserts that the next character is not "a" three times.
- Capturing subpatterns that occur inside negative lookahead assertions are counted, but their entries in the offsets vector are never set. Perl sets its numerical variables from any such patterns that are matched before the assertion fails to match something (thereby succeeding), but only if the negative lookahead assertion contains just one branch.
- Though binary zero characters are supported in the subject string, they are not allowed in a pattern string because it is passed as a normal C string, terminated by zero. The escape sequence "\x00" can be used in the pattern to represent a binary zero.
- The following Perl escape sequences are not supported: \l, \u, \L, \U. In fact these are implemented by Perl's general string-handling and are not part of its pattern matching engine.
- The Perl \G assertion is not supported as it is not relevant to single pattern matches.
- Fairly obviously, PCRE does not support the (?{code}) and (??{code}) construction. However, there is support for recursive patterns.
- There are at the time of writing some oddities in Perl 5.005_02 concerned with the settings of captured strings when part of a pattern is repeated. For example, matching "aba" against the pattern /^(a(b)?)+$/ sets $2 to the value "b", but matching "aabbaa" against /^(aa(bb)?)+$/ leaves $2 unset. However, if the pattern is changed to /^(aa(b(b))?)+$/ then $2 (and $3) get set. In Perl 5.004 $2 is set in both cases, and that is also **TRUE** of PCRE. If in the future Perl changes to a consistent state that is

different, PCRE may change to follow.

- Another as yet unresolved discrepancy is that in Perl 5.005_02 the pattern /^(a)?(?(1)a|b)+$/ matches the string "a", whereas in PCRE it does not. However, in both Perl and PCRE /^(a)?a/ matched against "a" leaves $1 unset.
- PCRE provides some extensions to the Perl regular expression facilities:
  - Although lookbehind assertions must match fixed length strings, each alternative branch of a lookbehind assertion can match a different length of string. Perl 5.005 requires them all to have the same length.
  - If PCRE_DOLLAR_ENDONLY is set and PCRE_MULTILINE is not set, the $ meta-character matches only at the very end of the string.
  - If PCRE_EXTRA is set, a backslash followed by a letter with no special meaning is faulted.
  - If PCRE_UNGREEDY is set, the greediness of the repetition quantifiers is inverted, that is, by default they are not greedy, but if followed by a question mark they are.

Regular Expression Details

Introduction

The syntax and semantics of the regular expressions supported by PCRE are described below. Regular expressions are also described in the Perl documentation and in a number of other books, some of which have copious examples. Jeffrey Friedl's "Mastering Regular Expressions", published by O'Reilly (ISBN 1-56592-257-3), covers them in great detail. The description here is intended as reference documentation.

A regular expression is a pattern that is matched against a subject string from left to right. Most characters stand for themselves in a pattern, and match the corresponding characters in the subject. As a trivial example, the pattern *The quick brown fox* matches a portion of a subject string that is identical to itself.

Meta-characters

The power of regular expressions comes from the ability to include alternatives and repetitions in the pattern. These are encoded in the pattern by the use of *meta-characters*, which do not stand for themselves but instead are interpreted in some special way.

There are two different sets of meta-characters: those that are recognized anywhere in the pattern except within square brackets, and those that are recognized in square brackets. Outside square brackets, the meta-characters are as follows:

\
    general escape character with several uses

^
: assert start of subject (or line, in multiline mode)

$
: assert end of subject (or line, in multiline mode)

.
: match any character except newline (by default)

[
: start character class definition

]
: end character class definition

|
: start of alternative branch

(
: start subpattern

)
: end subpattern

?
: extends the meaning of (, also 0 or 1 quantifier, also quantifier minimizer

*
: 0 or more quantifier

+
: 1 or more quantifier

{
: start min/max quantifier

}
: end min/max quantifier

Part of a pattern that is in square brackets is called a "character class". In a character class the only meta-characters are:

\
: general escape character

^
: negate the class, but only if the first character

-
: indicates character range

]
: terminates the character class

The following sections describe the use of each of the meta-characters.

Backslash

The backslash character has several uses. Firstly, if it is followed by a non-alphanumeric character, it takes away any special meaning that character may have. This use of backslash as an escape character applies both inside and outside character classes.

For example, if you want to match a "*" character, you write "*" in the pattern. This applies whether or not the following character would otherwise be interpreted as a meta-character, so it is always safe to precede a non-alphanumeric with "\" to specify that it stands for itself. In particular, if you want to match a backslash, you write "\\".

Note
Single and double quoted PHP strings have special meaning of backslash. Thus if \ has to be matched with a regular expression \\, then "\\\\" or '\\\\' must be used in PHP code.

If a pattern is compiled with the PCRE_EXTENDED option, whitespace in the pattern (other than in a character class) and characters between a "#" outside a character class and the next newline character are ignored. An escaping backslash can be used to include a whitespace or "#" character as part of the pattern.

A second use of backslash provides a way of encoding non-printing characters in patterns in a visible manner. There is no restriction on the appearance of non-printing characters, apart from the binary zero that terminates a pattern, but when a pattern is being prepared by text editing, it is usually easier to use one of the following escape sequences than the binary character it represents:

\a
: alarm, that is, the BEL character (hex 07)

\cx
: "control-x", where x is any character

\e
: escape (hex 1B)

\f
: formfeed (hex 0C)

\n
: newline (hex 0A)

\r
: carriage return (hex 0D)

\t
: tab (hex 09)

xhh
character with hex code hh

ddd
character with octal code ddd, or backreference

The precise effect of " *cx* " is as follows: if " *x* " is a lower case letter, it is converted to upper case. Then bit 6 of the character (hex 40) is inverted. Thus " *cz* " becomes hex 1A, but " *c{* " becomes hex 3B, while " *c;* " becomes hex 7B.

After " *x* ", up to two hexadecimal digits are read (letters can be in upper or lower case). In *UTF-8 mode*, " *x{...}* " is allowed, where the contents of the braces is a string of hexadecimal digits. It is interpreted as a UTF-8 character whose code number is the given hexadecimal number. The original hexadecimal escape sequence, *xhh*, matches a two-byte UTF-8 character if the value is greater than 127.

After " *0* " up to two further octal digits are read. In both cases, if there are fewer than two digits, just those that are present are used. Thus the sequence " *\0\x\07* " specifies two binary zeros followed by a BEL character. Make sure you supply two digits after the initial zero if the character that follows is itself an octal digit.

The handling of a backslash followed by a digit other than 0 is complicated. Outside a character class, PCRE reads it and any following digits as a decimal number. If the number is less than 10, or if there have been at least that many previous capturing left parentheses in the expression, the entire sequence is taken as a *back reference*. A description of how this works is given later, following the discussion of parenthesized subpatterns.

Inside a character class, or if the decimal number is greater than 9 and there have not been that many capturing subpatterns, PCRE re-reads up to three octal digits following the backslash, and generates a single byte from the least significant 8 bits of the value. Any subsequent digits stand for themselves. For example:

040
is another way of writing a space

40
is the same, provided there are fewer than 40 previous capturing subpatterns

7
is always a back reference

11
might be a back reference, or another way of writing a tab

011
is always a tab

0113
is a tab followed by the character "3"

113
: is the character with octal code 113 (since there can be no more than 99 back references)

377
: is a byte consisting entirely of 1 bits

81
: is either a back reference, or a binary zero followed by the two characters "8" and "1"

Note that octal values of 100 or greater must not be introduced by a leading zero, because no more than three octal digits are ever read.

All the sequences that define a single byte value can be used both inside and outside character classes. In addition, inside a character class, the sequence " *b* " is interpreted as the backspace character (hex 08). Outside a character class it has a different meaning (see below).

The third use of backslash is for specifying generic character types:

d
: any decimal digit

D
: any character that is not a decimal digit

h
: any horizontal whitespace character (since PHP 5.2.4)

H
: any character that is not a horizontal whitespace character (since PHP 5.2.4)

s
: any whitespace character

S
: any character that is not a whitespace character

v
: any vertical whitespace character (since PHP 5.2.4)

V
: any character that is not a vertical whitespace character (since PHP 5.2.4)

w
: any "word" character

W
: any "non-word" character

Each pair of escape sequences partitions the complete set of characters into two disjoint sets. Any given character matches one, and only one, of each pair.

A "word" character is any letter or digit or the underscore character, that is, any character which can be part of a Perl " *word* ". The definition of letters and digits is controlled by PCRE's character tables, and may vary if locale-specific matching is taking place. For example, in the "fr" (French) locale, some character codes greater than 128 are used for accented letters, and these are matched by *w*.

These character type sequences can appear both inside and outside character classes. They each match one character of the appropriate type. If the current matching point is at the end of the subject string, all of them fail, since there is no character to match.

The fourth use of backslash is for certain simple assertions. An assertion specifies a condition that has to be met at a particular point in a match, without consuming any characters from the subject string. The use of subpatterns for more complicated assertions is described below. The backslashed assertions are

b
: word boundary

B
: not a word boundary

A
: start of subject (independent of multiline mode)

Z
: end of subject or newline at end (independent of multiline mode)

z
: end of subject (independent of multiline mode)

G
: first matching position in subject

These assertions may not appear in character classes (but note that " *b* " has a different meaning, namely the backspace character, inside a character class).

A word boundary is a position in the subject string where the current character and the previous character do not both match *w* or *W* (i.e. one matches *w* and the other matches *W*), or the start or end of the string if the first or last character matches *w*, respectively.

The *A*, *Z*, and *z* assertions differ from the traditional circumflex and dollar (described below) in that they only ever match at the very start and end of the subject string, whatever options are set. They are not affected by the PCRE_MULTILINE or PCRE_DOLLAR_ENDONLY options. The difference between *Z* and *z* is that *Z* matches before a newline that is the last character of the string as well as at the end of the string, whereas *z* matches only at the end.

The \G assertion is true only when the current matching position is at the start point of the match, as specified by the `offset` argument of preg_match(). It differs from \A when the value of `offset` is non-zero. It is available since PHP 4.3.3.

\Q and \E can be used to ignore regexp metacharacters in the pattern since PHP 4.3.3. For example: \w+\Q.$.\E$will match one or more word characters, followed by literals.$. and anchored at the end of the string.

\K can be used to reset the match start since PHP 5.2.4. For example, the pattern *foo\Kbar* matches "foobar", but reports that it has matched "bar". The use of \K does not interfere with the setting of captured substrings. For example, when the pattern *(foo)\Kbar* matches "foobar", the first substring is still set to "foo".

Unicode character properties

Since PHP 4.4.0 and 5.1.0, three additional escape sequences to match generic character types are available when *UTF-8 mode* is selected. They are:

\p{xx}
: a character with the xx property

\P{xx}
: a character without the xx property

\X
: an extended Unicode sequence

The property names represented by *xx* above are limited to the Unicode general category properties. Each character has exactly one such property, specified by a two-letter abbreviation. For compatibility with Perl, negation can be specified by including a circumflex between the opening brace and the property name. For example, \p{^Lu} is the same as \P{Lu}.

If only one letter is specified with \p or \P, it includes all the properties that start with that letter. In this case, in the absence of negation, the curly brackets in the escape sequence are optional; these two examples have the same effect:

```
\p{L} \pL
```

Supported property codes

C	Other
Cc	Control
Cf	Format
Cn	Unassigned
Co	Private use

Cs	Surrogate
L	Letter
Ll	Lower case letter
Lm	Modifier letter
Lo	Other letter
Lt	Title case letter
Lu	Upper case letter
M	Mark
Mc	Spacing mark
Me	Enclosing mark
Mn	Non-spacing mark
N	Number
Nd	Decimal number
Nl	Letter number
No	Other number
P	Punctuation
Pc	Connector punctuation
Pd	Dash punctuation
Pe	Close punctuation
Pf	Final punctuation
Pi	Initial punctuation
Po	Other punctuation
Ps	Open punctuation
S	Symbol
Sc	Currency symbol
Sk	Modifier symbol

Sm	Mathematical symbol
So	Other symbol
Z	Separator
Zl	Line separator
Zp	Paragraph separator
Zs	Space separator

Extended properties such as "Greek" or "InMusicalSymbols" are not supported by PCRE.

Specifying caseless matching does not affect these escape sequences. For example, *\p{Lu}* always matches only upper case letters.

The *\X* escape matches any number of Unicode characters that form an extended Unicode sequence. *\X* is equivalent to *(?>\PM\pM*)*.

That is, it matches a character without the "mark" property, followed by zero or more characters with the "mark" property, and treats the sequence as an atomic group (see below). Characters with the "mark" property are typically accents that affect the preceding character.

Matching characters by Unicode property is not fast, because PCRE has to search a structure that contains data for over fifteen thousand characters. That is why the traditional escape sequences such as *\d* and *\w* do not use Unicode properties in PCRE.

Circumflex and dollar

Outside a character class, in the default matching mode, the circumflex character is an assertion which is true only if the current matching point is at the start of the subject string. Inside a character class, circumflex has an entirely different meaning (see below).

Circumflex need not be the first character of the pattern if a number of alternatives are involved, but it should be the first thing in each alternative in which it appears if the pattern is ever to match that branch. If all possible alternatives start with a circumflex, that is, if the pattern is constrained to match only at the start of the subject, it is said to be an "anchored" pattern. (There are also other constructs that can cause a pattern to be anchored.)

A dollar character is an assertion which is **TRUE** only if the current matching point is at the end of the subject string, or immediately before a newline character that is the last character in the string (by default). Dollar need not be the last character of the pattern if a number of alternatives are involved, but it should be the last item in any branch in which it appears. Dollar has no special meaning in a character class.

The meaning of dollar can be changed so that it matches only at the very end of the string,

by setting the PCRE_DOLLAR_ENDONLY option at compile or matching time. This does not affect the \Z assertion.

The meanings of the circumflex and dollar characters are changed if the PCRE_MULTILINE option is set. When this is the case, they match immediately after and immediately before an internal "\n" character, respectively, in addition to matching at the start and end of the subject string. For example, the pattern /^abc$/ matches the subject string "def\nabc" in multiline mode, but not otherwise. Consequently, patterns that are anchored in single line mode because all branches start with "^" are not anchored in multiline mode. The PCRE_DOLLAR_ENDONLY option is ignored if PCRE_MULTILINE is set.

Note that the sequences \A, \Z, and \z can be used to match the start and end of the subject in both modes, and if all branches of a pattern start with \A is it always anchored, whether PCRE_MULTILINE is set or not.

Full stop

Outside a character class, a dot in the pattern matches any one character in the subject, including a non-printing character, but not (by default) newline. If the PCRE_DOTALL option is set, then dots match newlines as well. The handling of dot is entirely independent of the handling of circumflex and dollar, the only relationship being that they both involve newline characters. Dot has no special meaning in a character class.

\C can be used to match single byte. It makes sense in *UTF-8 mode* where full stop matches the whole character which can consist of multiple bytes.

Square brackets

An opening square bracket introduces a character class, terminated by a closing square bracket. A closing square bracket on its own is not special. If a closing square bracket is required as a member of the class, it should be the first data character in the class (after an initial circumflex, if present) or escaped with a backslash.

A character class matches a single character in the subject; the character must be in the set of characters defined by the class, unless the first character in the class is a circumflex, in which case the subject character must not be in the set defined by the class. If a circumflex is actually required as a member of the class, ensure it is not the first character, or escape it with a backslash.

For example, the character class [aeiou] matches any lower case vowel, while [^aeiou] matches any character that is not a lower case vowel. Note that a circumflex is just a convenient notation for specifying the characters which are in the class by enumerating those that are not. It is not an assertion: it still consumes a character from the subject string, and fails if the current pointer is at the end of the string.

When caseless matching is set, any letters in a class represent both their upper case and lower case versions, so for example, a caseless [aeiou] matches "A" as well as "a", and a caseless [^aeiou] does not match "A", whereas a caseful version would.

The newline character is never treated in any special way in character classes, whatever the setting of the PCRE_DOTALL or PCRE_MULTILINE options is. A class such as [^a] will always match a newline.

The minus (hyphen) character can be used to specify a range of characters in a character class. For example, [d-m] matches any letter between d and m, inclusive. If a minus character is required in a class, it must be escaped with a backslash or appear in a position where it cannot be interpreted as indicating a range, typically as the first or last character in the class.

It is not possible to have the literal character "]" as the end character of a range. A pattern such as [W-]46] is interpreted as a class of two characters ("W" and "-") followed by a literal string "46]", so it would match "W46]" or "-46]". However, if the "]" is escaped with a backslash it is interpreted as the end of range, so [W-\]46] is interpreted as a single class containing a range followed by two separate characters. The octal or hexadecimal representation of "]" can also be used to end a range.

Ranges operate in ASCII collating sequence. They can also be used for characters specified numerically, for example [\000-\037]. If a range that includes letters is used when caseless matching is set, it matches the letters in either case. For example, [W-c] is equivalent to [][\^_`wxyzabc], matched caselessly, and if character tables for the "fr" locale are in use, [\xc8-\xcb] matches accented E characters in both cases.

The character types \d, \D, \s, \S, \w, and \W may also appear in a character class, and add the characters that they match to the class. For example, [\dABCDEF] matches any hexadecimal digit. A circumflex can conveniently be used with the upper case character types to specify a more restricted set of characters than the matching lower case type. For example, the class [^\W_] matches any letter or digit, but not underscore.

All non-alphanumeric characters other than \, -, ^ (at the start) and the terminating] are non-special in character classes, but it does no harm if they are escaped.

Vertical bar

Vertical bar characters are used to separate alternative patterns. For example, the pattern *gilbert*|*sullivan* matches either "gilbert" or "sullivan". Any number of alternatives may appear, and an empty alternative is permitted (matching the empty string). The matching process tries each alternative in turn, from left to right, and the first one that succeeds is used. If the alternatives are within a subpattern (defined below), "succeeds" means matching the rest of the main pattern as well as the alternative in the subpattern.

Internal option setting

The settings of PCRE_CASELESS, PCRE_MULTILINE, PCRE_DOTALL, PCRE_UNGREEDY, PCRE_EXTRA, PCRE_EXTENDED and PCRE_DUPNAMES can be changed from within the pattern by a sequence of Perl option letters enclosed between "(?" and ")". The option letters are:

Internal option letters

i	for PCRE_CASELESS
m	for PCRE_MULTILINE
s	for PCRE_DOTALL
x	for PCRE_EXTENDED
U	for PCRE_UNGREEDY
X	for PCRE_EXTRA
J	for PCRE_INFO_JCHANGED

For example, (?im) sets caseless, multiline matching. It is also possible to unset these options by preceding the letter with a hyphen, and a combined setting and unsetting such as (?im-sx), which sets PCRE_CASELESS and PCRE_MULTILINE while unsetting PCRE_DOTALL and PCRE_EXTENDED, is also permitted. If a letter appears both before and after the hyphen, the option is unset.

When an option change occurs at top level (that is, not inside subpattern parentheses), the change applies to the remainder of the pattern that follows. So */ab(?i)c/* matches only "abc" and "abC". This behaviour has been changed in PCRE 4.0, which is bundled since PHP 4.3.3. Before those versions, */ab(?i)c/* would perform as */abc/i* (e.g. matching "ABC" and "aBc").

If an option change occurs inside a subpattern, the effect is different. This is a change of behaviour in Perl 5.005. An option change inside a subpattern affects only that part of the subpattern that follows it, so *(a(?i)b)c* matches abc and aBc and no other strings (assuming PCRE_CASELESS is not used). By this means, options can be made to have different settings in different parts of the pattern. Any changes made in one alternative do carry on into subsequent branches within the same subpattern. For example, *(a(?i)b|c)* matches "ab", "aB", "c", and "C", even though when matching "C" the first branch is abandoned before the option setting. This is because the effects of option settings happen at compile time. There would be some very weird behaviour otherwise.

The PCRE-specific options PCRE_UNGREEDY and PCRE_EXTRA can be changed in the same way as the Perl-compatible options by using the characters U and X respectively. The (?X) flag setting is special in that it must always occur earlier in the pattern than any of the additional features it turns on, even when it is at top level. It is best put at the start.

Subpatterns

Subpatterns are delimited by parentheses (round brackets), which can be nested. Marking part of a pattern as a subpattern does two things:

1. It localizes a set of alternatives. For example, the pattern *cat(aract|erpillar|)* matches

one of the words "cat", "cataract", or "caterpillar". Without the parentheses, it would match "cataract", "erpillar" or the empty string.

2. It sets up the subpattern as a capturing subpattern (as defined above). When the whole pattern matches, that portion of the subject string that matched the subpattern is passed back to the caller via the *ovector* argument of **pcre_exec()**. Opening parentheses are counted from left to right (starting from 1) to obtain the numbers of the capturing subpatterns.

For example, if the string "the red king" is matched against the pattern *the ((red|white) (king|queen))* the captured substrings are "red king", "red", and "king", and are numbered 1, 2, and 3.

The fact that plain parentheses fulfil two functions is not always helpful. There are often times when a grouping subpattern is required without a capturing requirement. If an opening parenthesis is followed by "?:", the subpattern does not do any capturing, and is not counted when computing the number of any subsequent capturing subpatterns. For example, if the string "the white queen" is matched against the pattern *the ((?:red|white) (king|queen))* the captured substrings are "white queen" and "queen", and are numbered 1 and 2. The maximum number of captured substrings is 99, and the maximum number of all subpatterns, both capturing and non-capturing, is 200.

As a convenient shorthand, if any option settings are required at the start of a non-capturing subpattern, the option letters may appear between the "?" and the ":". Thus the two patterns
`(?i:saturday|sunday) (?:(?i)saturday|sunday)`
match exactly the same set of strings. Because alternative branches are tried from left to right, and options are not reset until the end of the subpattern is reached, an option setting in one branch does affect subsequent branches, so the above patterns match "SUNDAY" as well as "Saturday".

It is possible to name the subpattern with *(?P<name>pattern)* since PHP 4.3.3. Array with matches will contain the match indexed by the string alongside the match indexed by a number, then.

Repetition

Repetition is specified by quantifiers, which can follow any of the following items:

- a single character, possibly escaped
- the . metacharacter
- a character class
- a back reference (see next section)
- a parenthesized subpattern (unless it is an assertion - see below)

The general repetition quantifier specifies a minimum and maximum number of permitted matches, by giving the two numbers in curly brackets (braces), separated by a comma.

The numbers must be less than 65536, and the first must be less than or equal to the second. For example: *z{2,4}* matches "zz", "zzz", or "zzzz". A closing brace on its own is not a special character. If the second number is omitted, but the comma is present, there is no upper limit; if the second number and the comma are both omitted, the quantifier specifies an exact number of required matches. Thus *[aeiou]{3,}* matches at least 3 successive vowels, but may match many more, while *\d{8}* matches exactly 8 digits. An opening curly bracket that appears in a position where a quantifier is not allowed, or one that does not match the syntax of a quantifier, is taken as a literal character. For example, {,6} is not a quantifier, but a literal string of four characters.

The quantifier {0} is permitted, causing the expression to behave as if the previous item and the quantifier were not present.

For convenience (and historical compatibility) the three most common quantifiers have single-character abbreviations:

Single-character quantifiers

*	equivalent to *{0,}*
+	equivalent to *{1,}*
?	equivalent to *{0,1}*

It is possible to construct infinite loops by following a subpattern that can match no characters with a quantifier that has no upper limit, for example: *(a?)**

Earlier versions of Perl and PCRE used to give an error at compile time for such patterns. However, because there are cases where this can be useful, such patterns are now accepted, but if any repetition of the subpattern does in fact match no characters, the loop is forcibly broken.

By default, the quantifiers are "greedy", that is, they match as much as possible (up to the maximum number of permitted times), without causing the rest of the pattern to fail. The classic example of where this gives problems is in trying to match comments in C programs. These appear between the sequences /* and */ and within the sequence, individual * and / characters may appear. An attempt to match C comments by applying the pattern */*.**/* to the string */* *first comment* **/* *not comment* */** *second comment* **/* fails, because it matches the entire string due to the greediness of the .* item.

However, if a quantifier is followed by a question mark, then it ceases to be greedy, and instead matches the minimum number of times possible, so the pattern */*.*?*/* does the right thing with the C comments. The meaning of the various quantifiers is not otherwise changed, just the preferred number of matches. Do not confuse this use of question mark with its use as a quantifier in its own right. Because it has two uses, it can sometimes appear doubled, as in *\d??\d* which matches one digit by preference, but can match two if that is the only way the rest of the pattern matches.

If the PCRE_UNGREEDY option is set (an option which is not available in Perl) then the quantifiers are not greedy by default, but individual ones can be made greedy by following

them with a question mark. In other words, it inverts the default behaviour.

Quantifiers followed by + are "possessive". They eat as many characters as possible and don't return to match the rest of the pattern. Thus *.*abc* matches "aabc" but *.*+abc* doesn't because *.*+ eats the whole string. Possessive quantifiers can be used to speed up processing since PHP 4.3.3.

When a parenthesized subpattern is quantified with a minimum repeat count that is greater than 1 or with a limited maximum, more store is required for the compiled pattern, in proportion to the size of the minimum or maximum.

If a pattern starts with .* or .{0,} and the PCRE_DOTALL option (equivalent to Perl's /s) is set, thus allowing the . to match newlines, then the pattern is implicitly anchored, because whatever follows will be tried against every character position in the subject string, so there is no point in retrying the overall match at any position after the first. PCRE treats such a pattern as though it were preceded by \A. In cases where it is known that the subject string contains no newlines, it is worth setting PCRE_DOTALL when the pattern begins with .* in order to obtain this optimization, or alternatively using ^ to indicate anchoring explicitly.

When a capturing subpattern is repeated, the value captured is the substring that matched the final iteration. For example, after *(tweedle[dume]{3}\s*)+* has matched "tweedledum tweedledee" the value of the captured substring is "tweedledee". However, if there are nested capturing subpatterns, the corresponding captured values may have been set in previous iterations. For example, after */(a|(b))+/* matches "aba" the value of the second captured substring is "b".

Back references

Outside a character class, a backslash followed by a digit greater than 0 (and possibly further digits) is a back reference to a capturing subpattern earlier (i.e. to its left) in the pattern, provided there have been that many previous capturing left parentheses.

However, if the decimal number following the backslash is less than 10, it is always taken as a back reference, and causes an error only if there are not that many capturing left parentheses in the entire pattern. In other words, the parentheses that are referenced need not be to the left of the reference for numbers less than 10. See the section entitled "Backslash" above for further details of the handling of digits following a backslash.

A back reference matches whatever actually matched the capturing subpattern in the current subject string, rather than anything matching the subpattern itself. So the pattern *(sens|respons)e and \1ibility* matches "sense and sensibility" and "response and responsibility", but not "sense and responsibility". If caseful matching is in force at the time of the back reference, then the case of letters is relevant. For example, *((?i)rah)\s+\1* matches "rah rah" and "RAH RAH", but not "RAH rah", even though the original capturing subpattern is matched caselessly.

There may be more than one back reference to the same subpattern. If a subpattern has not actually been used in a particular match, then any back references to it always fail. For example, the pattern *(a|(bc))\2* always fails if it starts to match "a" rather than "bc". Because there may be up to 99 back references, all digits following the backslash are

taken as part of a potential back reference number. If the pattern continues with a digit character, then some delimiter must be used to terminate the back reference. If the PCRE_EXTENDED option is set, this can be whitespace. Otherwise an empty comment can be used.

A back reference that occurs inside the parentheses to which it refers fails when the subpattern is first used, so, for example, (a\1) never matches. However, such references can be useful inside repeated subpatterns. For example, the pattern *(a|b\1)*+ matches any number of "a"s and also "aba", "ababaa" etc. At each iteration of the subpattern, the back reference matches the character string corresponding to the previous iteration. In order for this to work, the pattern must be such that the first iteration does not need to match the back reference. This can be done using alternation, as in the example above, or by a quantifier with a minimum of zero.

Back references to the named subpatterns can be achieved by *(?P=name)* or, since PHP 5.2.4, also by *k<name>*, *k'name'*, *k{name}* or *g{name}*.

Assertions

An assertion is a test on the characters following or preceding the current matching point that does not actually consume any characters. The simple assertions coded as \b, \B, \A, \Z, \z, ^ and $ are described above. More complicated assertions are coded as subpatterns. There are two kinds: those that look ahead of the current position in the subject string, and those that look behind it.

An assertion subpattern is matched in the normal way, except that it does not cause the current matching position to be changed. Lookahead assertions start with (?= for positive assertions and (?! for negative assertions. For example, *\w+(?=;)* matches a word followed by a semicolon, but does not include the semicolon in the match, and *foo(?!bar)* matches any occurrence of "foo" that is not followed by "bar". Note that the apparently similar pattern *(?!foo)bar* does not find an occurrence of "bar" that is preceded by something other than "foo"; it finds any occurrence of "bar" whatsoever, because the assertion (?!foo) is always **TRUE** when the next three characters are "bar". A lookbehind assertion is needed to achieve this effect.

Lookbehind assertions start with (?<= for positive assertions and (?<! for negative assertions. For example, *(?<!foo)bar* does find an occurrence of "bar" that is not preceded by "foo". The contents of a lookbehind assertion are restricted such that all the strings it matches must have a fixed length. However, if there are several alternatives, they do not all have to have the same fixed length. Thus *(?<=bullock|donkey)* is permitted, but *(?<!dogs?|cats?)* causes an error at compile time. Branches that match different length strings are permitted only at the top level of a lookbehind assertion. This is an extension compared with Perl 5.005, which requires all branches to match the same length of string. An assertion such as *(?<=ab(c|de))* is not permitted, because its single top-level branch can match two different lengths, but it is acceptable if rewritten to use two top-level branches: *(?<=abc|abde)* The implementation of lookbehind assertions is, for each alternative, to temporarily move the current position back by the fixed width and then try to match. If there are insufficient characters before the current position, the match is deemed to fail. Lookbehinds in conjunction with once-only subpatterns can be particularly useful for matching at the ends of strings; an example is given at the end of the section on once-only

subpatterns.

Several assertions (of any sort) may occur in succession. For example, *(?<=\d{3})(?<!999)foo* matches "foo" preceded by three digits that are not "999". Notice that each of the assertions is applied independently at the same point in the subject string. First there is a check that the previous three characters are all digits, then there is a check that the same three characters are not "999". This pattern does not match "foo" preceded by six characters, the first of which are digits and the last three of which are not "999". For example, it doesn't match "123abcfoo". A pattern to do that is *(?<=\d{3}...)(?<!999)foo*

This time the first assertion looks at the preceding six characters, checking that the first three are digits, and then the second assertion checks that the preceding three characters are not "999".

Assertions can be nested in any combination. For example, *(?<=(?<!foo)bar)baz* matches an occurrence of "baz" that is preceded by "bar" which in turn is not preceded by "foo", while *(?<=\d{3}...(?<!999))foo* is another pattern which matches "foo" preceded by three digits and any three characters that are not "999".

Assertion subpatterns are not capturing subpatterns, and may not be repeated, because it makes no sense to assert the same thing several times. If any kind of assertion contains capturing subpatterns within it, these are counted for the purposes of numbering the capturing subpatterns in the whole pattern. However, substring capturing is carried out only for positive assertions, because it does not make sense for negative assertions.

Assertions count towards the maximum of 200 parenthesized subpatterns.

Once-only subpatterns

With both maximizing and minimizing repetition, failure of what follows normally causes the repeated item to be re-evaluated to see if a different number of repeats allows the rest of the pattern to match. Sometimes it is useful to prevent this, either to change the nature of the match, or to cause it fail earlier than it otherwise might, when the author of the pattern knows there is no point in carrying on.

Consider, for example, the pattern \d+foo when applied to the subject line *123456bar*

After matching all 6 digits and then failing to match "foo", the normal action of the matcher is to try again with only 5 digits matching the \d+ item, and then with 4, and so on, before ultimately failing. Once-only subpatterns provide the means for specifying that once a portion of the pattern has matched, it is not to be re-evaluated in this way, so the matcher would give up immediately on failing to match "foo" the first time. The notation is another kind of special parenthesis, starting with (?> as in this example: *(?>\d+)bar*

This kind of parenthesis "locks up" the part of the pattern it contains once it has matched, and a failure further into the pattern is prevented from backtracking into it. Backtracking past it to previous items, however, works as normal.

An alternative description is that a subpattern of this type matches the string of characters that an identical standalone pattern would match, if anchored at the current point in the

subject string.

Once-only subpatterns are not capturing subpatterns. Simple cases such as the above example can be thought of as a maximizing repeat that must swallow everything it can. So, while both \d+ and \d+? are prepared to adjust the number of digits they match in order to make the rest of the pattern match, (?>\d+) can only match an entire sequence of digits.

This construction can of course contain arbitrarily complicated subpatterns, and it can be nested.

Once-only subpatterns can be used in conjunction with look-behind assertions to specify efficient matching at the end of the subject string. Consider a simple pattern such as *abcd$* when applied to a long string which does not match. Because matching proceeds from left to right, PCRE will look for each "a" in the subject and then see if what follows matches the rest of the pattern. If the pattern is specified as ^*.*abcd$*then the initial .* matches the entire string at first, but when this fails (because there is no following "a"), it backtracks to match all but the last character, then all but the last two characters, and so on. Once again the search for "a" covers the entire string, from right to left, so we are no better off. However, if the pattern is written as *^(?>.*)(?<=abcd)* then there can be no backtracking for the .* item; it can match only the entire string. The subsequent lookbehind assertion does a single test on the last four characters. If it fails, the match fails immediately. For long strings, this approach makes a significant difference to the processing time.

When a pattern contains an unlimited repeat inside a subpattern that can itself be repeated an unlimited number of times, the use of a once-only subpattern is the only way to avoid some failing matches taking a very long time indeed. The pattern *(\D+|<\d+>)*[!?]* matches an unlimited number of substrings that either consist of non-digits, or digits enclosed in <>, followed by either ! or ?. When it matches, it runs quickly. However, if it is applied to *aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa* it takes a long time before reporting failure. This is because the string can be divided between the two repeats in a large number of ways, and all have to be tried. (The example used [!?] rather than a single character at the end, because both PCRE and Perl have an optimization that allows for fast failure when a single character is used. They remember the last single character that is required for a match, and fail early if it is not present in the string.) If the pattern is changed to *((?>\D+)|<\d+>)*[!?]* sequences of non-digits cannot be broken, and failure happens quickly.

Conditional subpatterns

It is possible to cause the matching process to obey a subpattern conditionally or to choose between two alternative subpatterns, depending on the result of an assertion, or whether a previous capturing subpattern matched or not. The two possible forms of conditional subpattern are

```
(?(condition)yes-pattern) (?(condition)yes-pattern|no-pattern)
```

If the condition is satisfied, the yes-pattern is used; otherwise the no-pattern (if present) is used. If there are more than two alternatives in the subpattern, a compile-time error occurs.

There are two kinds of condition. If the text between the parentheses consists of a sequence of digits, then the condition is satisfied if the capturing subpattern of that number

has previously matched. Consider the following pattern, which contains non-significant white space to make it more readable (assume the PCRE_EXTENDED option) and to divide it into three parts for ease of discussion: *(\( )? [^()]+ (?(1) \))*

The first part matches an optional opening parenthesis, and if that character is present, sets it as the first captured substring. The second part matches one or more characters that are not parentheses. The third part is a conditional subpattern that tests whether the first set of parentheses matched or not. If they did, that is, if subject started with an opening parenthesis, the condition is **TRUE**, and so the yes-pattern is executed and a closing parenthesis is required. Otherwise, since no-pattern is not present, the subpattern matches nothing. In other words, this pattern matches a sequence of non-parentheses, optionally enclosed in parentheses.

If the condition is the string *(R)*, it is satisfied if a recursive call to the pattern or subpattern has been made. At "top level", the condition is false.

If the condition is not a sequence of digits or (R), it must be an assertion. This may be a positive or negative lookahead or lookbehind assertion. Consider this pattern, again containing non-significant white space, and with the two alternatives on the second line:
`(?(?=[^a-z]*[a-z]) \d{2}-[a-z]{3}-\d{2} | \d{2}-\d{2}-\d{2} )`
The condition is a positive lookahead assertion that matches an optional sequence of non-letters followed by a letter. In other words, it tests for the presence of at least one letter in the subject. If a letter is found, the subject is matched against the first alternative; otherwise it is matched against the second. This pattern matches strings in one of the two forms dd-aaa-dd or dd-dd-dd, where aaa are letters and dd are digits.

Comments

The sequence (?# marks the start of a comment which continues up to the next closing parenthesis. Nested parentheses are not permitted. The characters that make up a comment play no part in the pattern matching at all.

If the PCRE_EXTENDED option is set, an unescaped # character outside a character class introduces a comment that continues up to the next newline character in the pattern.

Recursive patterns

Consider the problem of matching a string in parentheses, allowing for unlimited nested parentheses. Without the use of recursion, the best that can be done is to use a pattern that matches up to some fixed depth of nesting. It is not possible to handle an arbitrary nesting depth. Perl 5.6 has provided an experimental facility that allows regular expressions to recurse (among other things). The special item (?R) is provided for the specific case of recursion. This PCRE pattern solves the parentheses problem (assume the PCRE_EXTENDED option is set so that white space is ignored): *\(((?>[^()]+) | (?R))* \)*

First it matches an opening parenthesis. Then it matches any number of substrings which can either be a sequence of non-parentheses, or a recursive match of the pattern itself (i.e. a correctly parenthesized substring). Finally there is a closing parenthesis.

This particular example pattern contains nested unlimited repeats, and so the use of a once-only subpattern for matching strings of non-parentheses is important when applying the pattern to strings that do not match. For example, when it is applied to *(aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa()* it yields "no match" quickly. However, if a once-only subpattern is not used, the match runs for a very long time indeed because there are so many different ways the + and * repeats can carve up the subject, and all have to be tested before failure can be reported.

The values set for any capturing subpatterns are those from the outermost level of the recursion at which the subpattern value is set. If the pattern above is matched against *(ab(cd)ef)* the value for the capturing parentheses is "ef", which is the last value taken on at the top level. If additional parentheses are added, giving *\((((?>[^()]+) | (?R))*) \)* then the string they capture is "ab(cd)ef", the contents of the top level parentheses. If there are more than 15 capturing parentheses in a pattern, PCRE has to obtain extra memory to store data during a recursion, which it does by using pcre_malloc, freeing it via pcre_free afterwards. If no memory can be obtained, it saves data for the first 15 capturing parentheses only, as there is no way to give an out-of-memory error from within a recursion.

Since PHP 4.3.3, *(?1)*, *(?2)* and so on can be used for recursive subpatterns too. It is also possible to use named subpatterns: *(?P>name)* or *(?P&name)*.

If the syntax for a recursive subpattern reference (either by number or by name) is used outside the parentheses to which it refers, it operates like a subroutine in a programming language. An earlier example pointed out that the pattern *(sens|respons)e and \1ibility* matches "sense and sensibility" and "response and responsibility", but not "sense and responsibility". If instead the pattern *(sens|respons)e and (?1)ibility* is used, it does match "sense and responsibility" as well as the other two strings. Such references must, however, follow the subpattern to which they refer.

The maximum length of a subject string is the largest positive number that an integer variable can hold. However, PCRE uses recursion to handle subpatterns and indefinite repetition. This means that the available stack space may limit the size of a subject string that can be processed by certain patterns.

Performances

Certain items that may appear in patterns are more efficient than others. It is more efficient to use a character class like [aeiou] than a set of alternatives such as (a|e|i|o|u). In general, the simplest construction that provides the required behaviour is usually the most efficient. Jeffrey Friedl's book contains a lot of discussion about optimizing regular expressions for efficient performance.

When a pattern begins with .* and the PCRE_DOTALL option is set, the pattern is implicitly anchored by PCRE, since it can match only at the start of a subject string. However, if PCRE_DOTALL is not set, PCRE cannot make this optimization, because the . metacharacter does not then match a newline, and if the subject string contains newlines, the pattern may match from the character immediately following one of them instead of from the very start. For example, the pattern *(.*) second* matches the subject "first\nand second" (where \n stands for a newline character) with the first captured substring being

"and". In order to do this, PCRE has to retry the match starting after every newline in the subject.

If you are using such a pattern with subject strings that do not contain newlines, the best performance is obtained by setting PCRE_DOTALL, or starting the pattern with ^.* to indicate explicit anchoring. That saves PCRE from having to scan along the subject looking for a newline to restart at.

Beware of patterns that contain nested indefinite repeats. These can take a long time to run when applied to a string that does not match. Consider the pattern fragment *(a+)**

This can match "aaaa" in 33 different ways, and this number increases very rapidly as the string gets longer. (The * repeat can match 0, 1, 2, 3, or 4 times, and for each of those cases other than 0, the + repeats can match different numbers of times.) When the remainder of the pattern is such that the entire match is going to fail, PCRE has in principle to try every possible variation, and this can take an extremely long time.

An optimization catches some of the more simple cases such as *(a+)*b* where a literal character follows. Before embarking on the standard matching procedure, PCRE checks that there is a "b" later in the subject string, and if there is not, it fails the match immediately. However, when there is no following literal this optimization cannot be used. You can see the difference by comparing the behaviour of *(a+)*\d* with the pattern above. The former gives a failure almost instantly when applied to a whole line of "a" characters, whereas the latter takes an appreciable time with strings longer than about 20 characters.

PCRE Functions

preg_grep

preg_grep -- Return array entries that match the pattern

Description

array **preg_grep** (string `$pattern`, array `$input` [, int `$flags`])

Returns the array consisting of the elements of the *`input`* array that match the given *`pattern`*.

Parameters

`pattern`
: The pattern to search for, as a string.

`input`
: The input array.

`flags`
: If set to **PREG_GREP_INVERT**, this function returns the elements of the input array that do *not* match the given *`pattern`*.

Return Values

Returns an array indexed using the keys from the *`input`* array.

ChangeLog

Version	Description
4.2.0	The *`flags`* parameter was added.
4.0.4	Prior to this version, the returned array was indexed regardless of the keys of the *`input`* array. If you want to reproduce this old behavior, use array_values() on the returned array to reindex the values.

Examples

Example #3 - preg_grep() example

```
<?php
// return all array elements
// containing floating point numbers
$fl_array = preg_grep("/^(\d+)?\.\d+$/", $array);
?>
```

preg_last_error

preg_last_error -- Returns the error code of the last PCRE regex execution

Description

int **preg_last_error** (void)

Returns the error code of the last PCRE regex execution.

Example #4 - preg_last_error() example

```
<?php

preg_match('/(?:\D+|<\d+>)*[!?]/', 'foobar foobar foobar');

if (preg_last_error() == PREG_BACKTRACK_LIMIT_ERROR) {
   print 'Backtrack limit was exhausted!';
}

?>
```

The above example will output:

```
Backtrack limit was exhausted!
```

Return Values

Returns one of the following constants (explained on their own page):

- **PREG_NO_ERROR**
- **PREG_INTERNAL_ERROR**
- **PREG_BACKTRACK_LIMIT_ERROR** (see also pcre.backtrack_limit)
- **PREG_RECURSION_LIMIT_ERROR** (see also pcre.recursion_limit)
- **PREG_BAD_UTF8_ERROR**
- **PREG_BAD_UTF8_OFFSET_ERROR** (since PHP 5.3.0)

preg_match_all

preg_match_all -- Perform a global regular expression match

Description

int **preg_match_all** (string `$pattern`, string `$subject`, array `&$matches` [, int `$flags` [, int `$offset`]])

Searches *`subject`* for all matches to the regular expression given in *`pattern`* and puts them in *`matches`* in the order specified by *`flags`*.

After the first match is found, the subsequent searches are continued on from end of the last match.

Parameters

`pattern`
: The pattern to search for, as a string.

`subject`
: The input string.

`matches`
: Array of all matches in multi-dimensional array ordered according to *`flags`*.

`flags`
: Can be a combination of the following flags (note that it doesn't make sense to use **PREG_PATTERN_ORDER** together with **PREG_SET_ORDER**):
PREG_PATTERN_ORDER
: Orders results so that $matches[0] is an array of full pattern matches, $matches[1] is an array of strings matched by the first parenthesized subpattern, and so on.

```
<?php
preg_match_all("|<[^>]+>(.*)</[^>]+>|U",
   "<b>example: </b><div align=left>this is a test</div>",
   $out, PREG_PATTERN_ORDER);
echo $out[0][0] . ", " . $out[0][1] . "\n";
echo $out[1][0] . ", " . $out[1][1] . "\n";
?>
```

The above example will output:

```
<b>example: </b>, <div align=left>this is a test</div>
example: , this is a test
```

So, $out[0] contains array of strings that matched full pattern, and $out[1] contains array of strings enclosed by tags.

PREG_SET_ORDER

Orders results so that $matches[0] is an array of first set of matches, $matches[1] is an array of second set of matches, and so on.

```
<?php
preg_match_all("|<[^>]+>(.*)</[^>]+>|U",
   "<b>example: </b><div align=\"left\">this is a test</div>",
   $out, PREG_SET_ORDER);
echo $out[0][0] . ", " . $out[0][1] . "\n";
echo $out[1][0] . ", " . $out[1][1] . "\n";
?>
```

The above example will output:

```
<b>example: </b>, example:
<div align="left">this is a test</div>, this is a test
```

PREG_OFFSET_CAPTURE

If this flag is passed, for every occurring match the appendant string offset will also be returned. Note that this changes the value of *matches* in an array where every element is an array consisting of the matched string at offset *0* and its string offset into *subject* at offset *1*.

If no order flag is given, **PREG_PATTERN_ORDER** is assumed.

offset

Normally, the search starts from the beginning of the subject string. The optional parameter *offset* can be used to specify the alternate place from which to start the search (in bytes).

Note
Using *offset* is not equivalent to passing *substr($subject, $offset)* to preg_match_all() in place of the subject string, because *pattern* can contain assertions such as ^, *$*or *(?<=x)*. See preg_match() for examples.

Return Values

Returns the number of full pattern matches (which might be zero), or **FALSE** if an error occurred.

ChangeLog

Version	Description
4.3.3	The *offset* parameter was added

4.3.0	The **PREG_OFFSET_CAPTURE** flag was added

Examples

Example #5 - Getting all phone numbers out of some text.

```
<?php
preg_match_all("/\(?  (\d{3})?  \)?  (?(1)  [\-\s] ) \d{3}-\d{4}/x",
               "Call 555-1212 or 1-800-555-1212", $phones);
?>
```

Example #6 - Find matching HTML tags (greedy)

```
<?php
// The \\2 is an example of backreferencing. This tells pcre that
// it must match the second set of parentheses in the regular expression
// itself, which would be the ([\w]+) in this case. The extra backslash is
// required because the string is in double quotes.
$html = "<b>bold text</b><a href=howdy.html>click me</a>";

preg_match_all("/(<([\w]+)[^>]*>)(.*)(<\/\\2>)/", $html, $matches,
PREG_SET_ORDER);

foreach ($matches as $val) {
   echo "matched: " . $val[0] . "\n";
   echo "part 1: " . $val[1] . "\n";
   echo "part 2: " . $val[3] . "\n";
   echo "part 3: " . $val[4] . "\n\n";
}
?>
```

The above example will output:

```
matched: <b>bold text</b>
part 1: <b>
part 2: bold text
part 3: </b>

matched: <a href=howdy.html>click me</a>
part 1: <a href=howdy.html>
part 2: click me
part 3: </a>
```

Example #7 - Using named subpattern

```
<?php

$str = <<<FOO
a: 1
b: 2
c: 3
FOO;

preg_match_all('/(?<name>\w+): (?<digit>\d+)/', $str, $matches);

print_r($matches);

?>
```

The above example will output:

```
Array
(
   [0] => Array
       (
           [0] => a: 1
           [1] => b: 2
           [2] => c: 3
       )

   [name] => Array
       (
           [0] => a
           [1] => b
           [2] => c
       )

   [1] => Array
       (
           [0] => a
           [1] => b
           [2] => c
       )

   [digit] => Array
       (
           [0] => 1
           [1] => 2
           [2] => 3
       )

   [2] => Array
       (
           [0] => 1
           [1] => 2
           [2] => 3
       )

)
```

See Also

- preg_match()
- preg_replace()
- preg_split()

preg_match

preg_match -- Perform a regular expression match

Description

int **preg_match** (string `$pattern`, string `$subject` [, array `&$matches` [, int `$flags` [, int `$offset`]]])

Searches *`subject`* for a match to the regular expression given in *`pattern`*.

Parameters

`pattern`
: The pattern to search for, as a string.

`subject`
: The input string.

`matches`
: If *`matches`* is provided, then it is filled with the results of search. *$matches[0]* will contain the text that matched the full pattern, *$matches[1]* will have the text that matched the first captured parenthesized subpattern, and so on.

`flags`
: *`flags`* can be the following flag:

  PREG_OFFSET_CAPTURE
  : If this flag is passed, for every occurring match the appendant string offset will also be returned. Note that this changes the return value in an array where every element is an array consisting of the matched string at index *0* and its string offset into *`subject`* at index *1*.

`offset`
: Normally, the search starts from the beginning of the subject string. The optional parameter *`offset`* can be used to specify the alternate place from which to start the search (in bytes).

Note
Using *`offset`* is not equivalent to passing *substr($subject, $offset)* to preg_match() in place of the subject string, because *`pattern`* can contain assertions such as ^, *$* or *(?<=x)*. Compare:

```
<?php
$subject = "abcdef";
$pattern = '/^def/';
preg_match($pattern, $subject, $matches, PREG_OFFSET_CAPTURE, 3);
print_r($matches);
```

```
?>
```

The above example will output:

```
Array
(
)
```

while this example

```
<?php
$subject = "abcdef";
$pattern = '/^def/';
preg_match($pattern, substr($subject,3), $matches, PREG_OFFSET_CAPTURE);
print_r($matches);
?>
```

will produce

```
Array
(
    [0] => Array
        (
            [0] => def
            [1] => 0
        )

)
```

Return Values

preg_match() returns the number of times *pattern* matches. That will be either 0 times (no match) or 1 time because preg_match() will stop searching after the first match. preg_match_all() on the contrary will continue until it reaches the end of *subject*. preg_match() returns **FALSE** if an error occurred.

ChangeLog

Version	Description
4.3.3	The *offset* parameter was added
4.3.0	The **PREG_OFFSET_CAPTURE** flag was added
4.3.0	The *flags* parameter was added

Examples

Example #8 - Find the string of text "php"

```
<?php
// The "i" after the pattern delimiter indicates a case-insensitive search
if (preg_match("/php/i", "PHP is the web scripting language of choice.")) {
   echo "A match was found.";
} else {
   echo "A match was not found.";
}
?>
```

Example #9 - Find the word "web"

```
<?php
/* The \b in the pattern indicates a word boundary, so only the distinct
* word "web" is matched, and not a word partial like "webbing" or "cobweb"
*/
if (preg_match("/\bweb\b/i", "PHP is the web scripting language of
choice.")) {
   echo "A match was found.";
} else {
   echo "A match was not found.";
}

if (preg_match("/\bweb\b/i", "PHP is the website scripting language of
choice.")) {
   echo "A match was found.";
} else {
   echo "A match was not found.";
}
?>
```

Example #10 - Getting the domain name out of a URL

```
<?php
// get host name from URL
preg_match('@^(?:http://)?([^/]+)@i',
   "http://www.php.net/index.html", $matches);
$host = $matches[1];

// get last two segments of host name
preg_match('/[^.]+\.[^.]+$/', $host, $matches);
echo "domain name is: {$matches[0]}\n";
?>
```

The above example will output:

```
domain name is: php.net
```

Example #11 - Using named subpattern

```
<?php

$str = 'foobar: 2008';

preg_match('/(?<name>\w+): (?<digit>\d+)/', $str, $matches);

print_r($matches);

?>
```

The above example will output:

```
Array
(
   [0] => foobar: 2008
   [name] => foobar
   [1] => foobar
   [digit] => 2008
   [2] => 2008
)
```

Notes

Tip

Do not use preg_match() if you only want to check if one string is contained in another string. Use strpos() or strstr() instead as they will be faster.

See Also

- preg_match_all()
- preg_replace()
- preg_split()

preg_quote

preg_quote -- Quote regular expression characters

Description

string **preg_quote** (string `$str` [, string `$delimiter`])

preg_quote() takes *`str`* and puts a backslash in front of every character that is part of the regular expression syntax. This is useful if you have a run-time string that you need to match in some text and the string may contain special regex characters.

The special regular expression characters are:. \ + * ? [^] $ ( ) { } = ! < > | :

Parameters

`str`
: The input string.

`delimiter`
: If the optional *`delimiter`* is specified, it will also be escaped. This is useful for escaping the delimiter that is required by the PCRE functions. The / is the most commonly used delimiter.

Return Values

Returns the quoted string.

Examples

Example #12 - preg_quote() example

```
<?php
$keywords = '$40 for a g3/400';
$keywords = preg_quote($keywords, '/');
echo $keywords; // returns \$40 for a g3\/400
?>
```

Example #13 - Italicizing a word within some text

```
<?php
// In this example, preg_quote($word) is used to keep the
```

```
// asterisks from having special meaning to the regular
// expression.

$textbody = "This book is *very* difficult to find.";
$word = "*very*";
$textbody = preg_replace ("/" . preg_quote($word) . "/",
                          "<i>" . $word . "</i>",
                          $textbody);
?>
```

Notes

Note
This function is binary-safe.

preg_replace_callback

preg_replace_callback -- Perform a regular expression search and replace using a callback

Description

mixed **preg_replace_callback** (mixed `$pattern`, callback `$callback`, mixed `$subject` [, int `$limit` [, int `&$count`]])

The behavior of this function is almost identical to preg_replace(), except for the fact that instead of *`replacement`* parameter, one should specify a *`callback`*.

Parameters

`pattern`
The pattern to search for. It can be either a string or an array with strings.

`callback`
A callback that will be called and passed an array of matched elements in the *`subject`* string. The callback should return the replacement string. You'll often need the *`callback`* function for a preg_replace_callback() in just one place. In this case you can use create_function() to declare an anonymous function as callback within the call to preg_replace_callback(). By doing it this way you have all information for the call in one place and do not clutter the function namespace with a callback function's name not used anywhere else.

Example #14 - preg_replace_callback() and create_function()

```
<?php
/* a unix-style command line filter to convert uppercase
* letters at the beginning of paragraphs to lowercase */
$fp = fopen("php://stdin", "r") or die("can't read stdin");
while (!feof($fp)) {
   $line = fgets($fp);
   $line = preg_replace_callback(
       '|<p>\s*\w|',
       create_function(
           // single quotes are essential here,
           // or alternative escape all $ as \$
           '$matches',
           'return strtolower($matches[0]);'
       ),
       $line
   );
   echo $line;
}
fclose($fp);
?>
```

subject
: The string or an array with strings to search and replace.

limit
: The maximum possible replacements for each pattern in each *subject* string. Defaults to *-1* (no limit).

count
: If specified, this variable will be filled with the number of replacements done.

Return Values

preg_replace_callback() returns an array if the *subject* parameter is an array, or a string otherwise.

If matches are found, the new subject will be returned, otherwise *subject* will be returned unchanged.

ChangeLog

Version	Description
5.1.0	The *count* parameter was added

Examples

Example #15 - preg_replace_callback() example

```
<?php
// this text was used in 2002
// we want to get this up to date for 2003
$text = "April fools day is 04/01/2002\n";
$text.= "Last christmas was 12/24/2001\n";
// the callback function
function next_year($matches)
{
 // as usual: $matches[0] is the complete match
 // $matches[1] the match for the first subpattern
 // enclosed in '(...)' and so on
 return $matches[1].($matches[2]+1);
}
echo preg_replace_callback(
           "|(\d{2}/\d{2}/)(\d{4})|",
           "next_year",
           $text);

?>
```

The above example will output:

```
April fools day is 04/01/2003
Last christmas was 12/24/2002
```

Example #16 - preg_replace_callback() using recursive structure to handle encapsulated BB code

```
<?php
$input = "plain [indent] deep [indent] deeper [/indent] deep [/indent]
plain";

function parseTagsRecursive($input)
{

   $regex = '#\[indent]((?:[^[]|\[(?!/?indent])|(?R))+)\[/indent]#';

   if (is_array($input)) {
       $input = '<div style="margin-left: 10px">'.$input[1].'</div>';
   }

   return preg_replace_callback($regex, 'parseTagsRecursive', $input);
}

$output = parseTagsRecursive($input);

echo $output;
?>
```

See Also

- preg_replace()
- create_function()
- information about the callback type

preg_replace

preg_replace -- Perform a regular expression search and replace

Description

mixed **preg_replace** (mixed `$pattern`, mixed `$replacement`, mixed `$subject` [, int `$limit` [, int `&$count`]])

Searches *`subject`* for matches to *`pattern`* and replaces them with *`replacement`*.

Parameters

`pattern`
: The pattern to search for. It can be either a string or an array with strings. The *e* modifier makes preg_replace() treat the *`replacement`* parameter as PHP code after the appropriate references substitution is done. Tip: make sure that *`replacement`* constitutes a valid PHP code string, otherwise PHP will complain about a parse error at the line containing preg_replace().

`replacement`
: The string or an array with strings to replace. If this parameter is a string and the *`pattern`* parameter is an array, all patterns will be replaced by that string. If both *`pattern`* and *`replacement`* parameters are arrays, each *`pattern`* will be replaced by the *`replacement`* counterpart. If there are fewer elements in the *`replacement`* array than in the *`pattern`* array, any extra *`pattern`* s will be replaced by an empty string. *`replacement`* may contain references of the form \\ *n* or (since PHP 4.0.4) *$n*, with the latter form being the preferred one. Every such reference will be replaced by the text captured by the *n* 'th parenthesized pattern. *n* can be from 0 to 99, and *0* or *$0* refers to the text matched by the whole pattern. Opening parentheses are counted from left to right (starting from 1) to obtain the number of the capturing subpattern. When working with a replacement pattern where a backreference is immediately followed by another number (i.e.: placing a literal number immediately after a matched pattern), you cannot use the familiar *1* notation for your backreference. *11*, for example, would confuse preg_replace() since it does not know whether you want the *1* backreference followed by a literal *1*, or the *11* backreference followed by nothing. In this case the solution is to use *\${1}1*. This creates an isolated *$1* backreference, leaving the *1* as a literal. When using the *e* modifier, this function escapes some characters (namely *'*, *"*, \ and NULL) in the strings that replace the backreferences. This is done to ensure that no syntax errors arise from backreference usage with either single or double quotes (e.g. *'strlen(\'$1\')+strlen("$2")'*). Make sure you are aware of PHP's string syntax to know exactly how the interpreted string will look like.

`subject`
: The string or an array with strings to search and replace. If *`subject`* is an array, then the search and replace is performed on every entry of *`subject`*, and the return value is an array as well.

`limit`

The maximum possible replacements for each pattern in each *subject* string. Defaults to *-1* (no limit).

count
If specified, this variable will be filled with the number of replacements done.

Return Values

preg_replace() returns an array if the *subject* parameter is an array, or a string otherwise.

If matches are found, the new *subject* will be returned, otherwise *subject* will be returned unchanged or **NULL** if an error occurred.

ChangeLog

Version	Description
5.1.0	Added the *count* parameter
4.0.4	Added the '$n' form for the *replacement* parameter
4.0.2	Added the *limit* parameter

Examples

Example #17 - Using backreferences followed by numeric literals

```
<?php
$string = 'April 15, 2003';
$pattern = '/(\w+) (\d+), (\d+)/i';
$replacement = '${1}1,$3';
echo preg_replace($pattern, $replacement, $string);
?>
```

The above example will output:

```
April1,2003
```

Example #18 - Using indexed arrays with preg_replace()

```
<?php
$string = 'The quick brown fox jumped over the lazy dog.';
$patterns[0] = '/quick/';
$patterns[1] = '/brown/';
$patterns[2] = '/fox/';
$replacements[2] = 'bear';
$replacements[1] = 'black';
$replacements[0] = 'slow';
echo preg_replace($patterns, $replacements, $string);
?>
```

The above example will output:

```
The bear black slow jumped over the lazy dog.
```

By ksorting patterns and replacements, we should get what we wanted.

```
<?php
ksort($patterns);
ksort($replacements);
echo preg_replace($patterns, $replacements, $string);
?>
```

The above example will output:

```
The slow black bear jumped over the lazy dog.
```

Example #19 - Replacing several values

```
<?php
$patterns = array ('/(19|20)(\d{2})-(\d{1,2})-(\d{1,2})/',
                   '/^\s*{(\w+)}\s*=/');
$replace = array ('\3/\4/\1\2', '$\1 =');
echo preg_replace($patterns, $replace, '{startDate} = 1999-5-27');
?>
```

The above example will output:

```
$startDate = 5/27/1999
```

Example #20 - Using the 'e' modifier

```
<?php
preg_replace("/(<\/?)(\w+)([^>]*>)/e",
            "'\\1'.strtoupper('\\2').'\\3'",
            $html_body);
```

```
?>
```

This would capitalize all HTML tags in the input text.

Example #21 - Strip whitespace

This example strips excess whitespace from a string.

```
<?php
$str = 'foo   o';
$str = preg_replace('/\s\s+/', ' ', $str);
// This will be 'foo o' now
echo $str;
?>
```

Example #22 - Using the *count* parameter

```
<?php
$count = 0;

echo preg_replace(array('/\d/', '/\s/'), '*', 'xp 4 to', -1 , $count);
echo $count; //3
?>
```

The above example will output:

```
xp***to
3
```

Notes

Note

When using arrays with *pattern* and *replacement*, the keys are processed in the order they appear in the array. This is *not necessarily* the same as the numerical index order. If you use indexes to identify which *pattern* should be replaced by which *replacement*, you should perform a ksort() on each array prior to calling preg_replace() .

See Also

- preg_match()
- preg_replace_callback()
- preg_split()

preg_split

preg_split -- Split string by a regular expression

Description

array **preg_split** (string `$pattern`, string `$subject` [, int `$limit` [, int `$flags`]])

Split the given string by a regular expression.

Parameters

`pattern`
: The pattern to search for, as a string.

`subject`
: The input string.

`limit`
: If specified, then only substrings up to *`limit`* are returned, and if *`limit`* is -1, it actually means "no limit", which is useful for specifying the *`flags`*.

`flags`
: *`flags`* can be any combination of the following flags (combined with bitwise | operator):

PREG_SPLIT_NO_EMPTY
: If this flag is set, only non-empty pieces will be returned by preg_split().

PREG_SPLIT_DELIM_CAPTURE
: If this flag is set, parenthesized expression in the delimiter pattern will be captured and returned as well.

PREG_SPLIT_OFFSET_CAPTURE
: If this flag is set, for every occurring match the appendant string offset will also be returned. Note that this changes the return value in an array where every element is an array consisting of the matched string at offset *0* and its string offset into *`subject`* at offset *1*.

Return Values

Returns an array containing substrings of *`subject`* split along boundaries matched by *`pattern`*.

ChangeLog

Version	Description
4.3.0	The **PREG_SPLIT_OFFSET_CAPTURE** was added
4.0.5	The **PREG_SPLIT_DELIM_CAPTURE** was added
4.0.0	The *flags* parameter was added

Examples

Example #23 - preg_split() example : Get the parts of a search string

```
<?php
// split the phrase by any number of commas or space characters,
// which include " ", \r, \t, \n and \f
$keywords = preg_split("/[\s,]+/", "hypertext language, programming");
?>
```

Example #24 - Splitting a string into component characters

```
<?php
$str = 'string';
$chars = preg_split('//', $str, -1, PREG_SPLIT_NO_EMPTY);
print_r($chars);
?>
```

Example #25 - Splitting a string into matches and their offsets

```
<?php
$str = 'hypertext language programming';
$chars = preg_split('/ /', $str, -1, PREG_SPLIT_OFFSET_CAPTURE);
print_r($chars);
?>
```

The above example will output:

```
Array
(
   [0] => Array
       (
           [0] => hypertext
```

```
            [1] => 0
        )

    [1] => Array
        (
            [0] => language
            [1] => 10
        )

    [2] => Array
        (
            [0] => programming
            [1] => 19
        )

)
```

Notes

Tip
If you don't need the power of regular expressions, you can choose faster (albeit simpler) alternatives like explode() or str_split().

See Also

- spliti()
- split()
- implode()
- preg_match()
- preg_match_all()
- preg_replace()