Regular Expressions

Disclaimer: вы смотрите просто запись лекции, это **НЕ** специально подготовленный видеокурс!

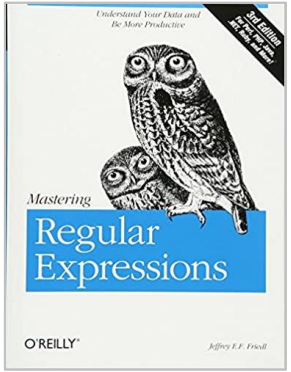
Intro

Regular expression (regex, regexp) is a mechanism of text processing (i.e., search and/or replace) based on flexible patterns.

Disclaimer

Like many other topics in this course, regular expressions are much more complex than they may seem at the first glance.

Please refer to this perfect book for more information:
“Mastering Regular Expressions” by Jeffrey Friedl.



Intro

PHP supports so-called PCRE (Perl Compatible Regular Expressions). PCRE support is very common among programming languages and tools. So, you may easily apply the knowledge from this part of our course in many other cases.

Please refer to these sources for details:
https://www.php.net/manual/en/reference.pcre.pattern.syntax.php

The basis for regular expressions

The fundamental idea behind regular expressions is that there are some special characters (“meta-characters”) and characters sequences (“escape sequences”) that do not stand for themselves but instead are interpreted in a special way.

In “normal life” this is just a dot…

```
.
```

… but inside a regular expression this is a meta-character that means “any character except newline (by default)”.

In “normal life” these are just a back-slash and “d” letter…

```
\d
```

… but inside a regular expression this is an escape sequences that means “any digit” (i.e. “0123456789”).

The basic syntax and trivial sample

From the “under-the-hood” point of view, in PHP regexes are just strings (unlike in some other programming languages, where regexes are objects).

When using regexes, it is required that the pattern is enclosed by delimiters. A delimiter can be any non-alphanumeric, non-backslash, non-whitespace character. Leading whitespace before a valid delimiter is silently ignored.

Often used delimiters are forward slashes (/), hash signs (#) and tildes (~).

The basic syntax and trivial sample

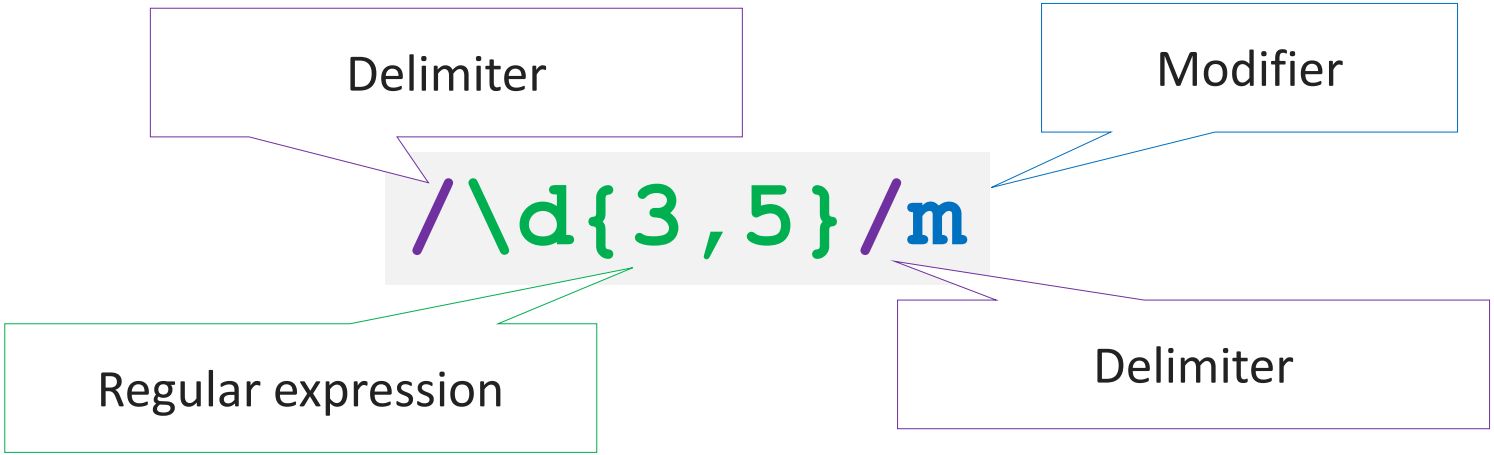
So, let's look at the most trivial sample of regex usage.

```
<?php

$text = 'Some date is 12.10.2005, and some number is 890.';
preg_match_all("/\d/", $text, $result);
print_r($result);

/* Array
  ([0] => Array ([0] => 1, [1] => 2, [2] => 1, [3] => 0, [4] => 2,
                 [5] => 0, [6] => 0, [7] => 5, [8] => 8, [9] => 9,  [10] => 0))
*/

/*
Other valid delimiters samples:
/foo bar/
#^[^0-9]$#
+php+
%[a-zA-Z0-9_-]%
*/
```

Digit

The basic syntax and trivial sample

Once again, here how it looks like:



Meta-characters and escape sequences

A **meta-character** is “a character describing other characters”. The most important meta-character in regexes is “\” (a back-slash).

When put before “a normal character”, a back-slash produces an escape sequence: “d” is just a letter, “\d” is “any digit”.

When put before another meta-character, a back-slash produces a “normal character”: “.” is “any symbol”, “\.” is just “a dot”.

First we have to list all meta-characters and escape sequences, then we’ll use them step-by-step in simple examples.

Meta-characters

Here's the list of all meta-characters used in PHP regexes.
See details here: https://www.php.net/manual/en/regexp.reference.meta.php

Meta-character	Description
\	general escape character with several uses
^	assert start of subject (or line, in multiline mode)
$	assert end of subject or before a terminating newline (or end of 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 makes greedy quantifiers lazy
*	0 or more quantifier
+	1 or more quantifier
{	start min/max quantifier
}	end min/max quantifier

Meta-character	Description
\	general escape character
^	negate the class, but only if the first character
-	indicates character range

Inside square brackets

Outside square brackets

Escape sequences

Here's the list of all escape sequences used in PHP regexes.
See details here: https://www.php.net/manual/en/regexp.reference.escape.php

Escape sequence	Description
\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)
\p{xx}	a character with the xx property, see "unicode properties" for more info
\P{xx}	a character without the xx property, see "unicode properties" for more info
\r	carriage return (hex 0D)
\R	line break: matches \n, \r and \r\n
\t	tab (hex 09)
\xhh	character with hex code hh
\ddd	character with octal code ddd, or backreference

Escape sequence	Description
\d	any decimal digit
\D	any character that is not a decimal digit
\h	any horizontal whitespace character
\H	any character that is not a horizontal whitespace character
\s	any whitespace character
\S	any character that is not a whitespace character
\v	any vertical whitespace character
\V	any character that is not a vertical whitespace character
\w	any "word" character
\W	any "non-word" character
\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

Simplified cheat-sheet

Meta-character or escape sequence	Description
\	general escape character (turns “normal” characters in meta-characters and vice versa)
^ and $	data (or line) start and end
.	any symbol (except \n by default)
[and]	start and end of a character class definition
|	start of alternative branch
(and)	subpattern start and end
{ and }	quantifier start and end
?	0 or 1 quantifier (in most cases)
*	0 or more quantifier
+	1 or more quantifier
\d	any decimal digit
\D	any character that is not a decimal digit
\s	any whitespace character
\S	any character that is not a whitespace character
\w	any “word” character
\W	any “non-word” character
\b	word boundary
\B	not a word boundary
\t, \n, \r	just a usual “tab”, “newline”, “carriage return” symbols

Samples…

OK, let’s use this information and solve some trivial tasks.

```php
<?php

$text = 'Some date is 12.10.2005.';

// 1) Find all digits:
preg_match_all("/\d/", $text, $result);
print_r($result);
// Array ([0] => 1, [1] => 2, [2] => 1, [3] => 0, [4] => 2, [5] => 0, [6] => 0, [7] => 5)

// 2) Find all "two-digits sequences":
preg_match_all("/\d\d/", $text, $result);
print_r($result);
// Array ([0] => 12, [1] => 10, [2] => 20, [3] => 05)

// 3) Find all "three-digits sequences":
preg_match_all("/\d\d\d/", $text, $result);
print_r($result);
// Array ([0] => 200)
```

Digit

Two digits

Three digits

Pay attention! By default, regexes are NOT recursive!

And some more samples…

```php
<?php

$text = 'Some date is 12.10.2005.';

// 4) Find all "two-digits sequences" using quantifiers:
preg_match_all("/\d{2}/", $text, $result);
print_r($result);
// Array ([0] => 12, [1] => 10, [2] => 20, [3] => 05)

// 5) Find all "three-digits sequences" using quantifiers:
preg_match_all("/\d{3}/", $text, $result);
print_r($result);
// Array ([0] => 200)

// 6) Find all "2-3-4-digits sequences" using quantifiers:
preg_match_all("/\d{2,4}/", $text, $result);
print_r($result);
// Array ([0] => 12, [1] => 10, [2] => 2005)
```

Two digits

Three digits

Two, three, or four digits

Character classes

A character class matches a single character in the subject.
See details here:
https://www.php.net/manual/en/regexp.reference.character-classes.php

Some samples:

- [1234567890abcdefABCDEF] is a “hexadecimal digit”
- [\da-fA-F] is also a “hexadecimal digit”
- [^\da-fA-F] is NOT a “hexadecimal digit” (the “^” at the beginning is a “not”)

Samples…

Let’s continue with samples.

```
<?php

$text = 'There are item codes: ABC-123 and DEF-789 in the order.';

// 7) Find all item codes:
preg_match_all("/[A-Z]{3}-\d{3}/", $text, $result);
print_r($result);
// Array ([0] => ABC-123, [1] => DEF-789)

// 8) Find all "words":
preg_match_all("/\w+/", $text, $result);
print_r($result);
// Array ([0] => There, [1] => are, [2] => item, [3] => ABC, [4] => 123, [5] => and,
//        [6] => DEF, [7] => 789, [8] => in, [9] => the, [10] => order)

// 9) Find all "word-like-sequences":
preg_match_all("/[\w-]+/", $text, $result);
print_r($result);
// Array ([0] => There, [1] => are, [2] => item, [3] => ABC-123, [4] => and,
//        [5] => DEF-789, [6] => in, [7] => the, [8] => order)
```

Three uppercase letters

“-”

Three digits

1+ letters, digits, _

1+ letters, digits, _, -

No one knows what a word is.
You don’t’ believe, do you? OK.
“2 + 3 = 5” – how many words are here?
“2010-2015” – how many words are here?
“Apollo-11 and/or Apollo 12” – how many words are here? And so on… ☹

Samples…

Let’s look at alternatives (“|”).

```
<?php

$text = 'Date 1 is 28.02.1995. Date 2 is 2/28/95.';

// 10) Find all dates:
preg_match_all("/\d{2}\.\d{2}\.\d{4}|\d{1,2}\/\d{1,2}\/\d{2}/", $text, $result);
print_r($result);
// Array ([0] => 28.02.1995, [1] => 2/28/95)
```

Two digits | Dot | Two digits | Dot | Four digits | 1-2 digits | Slash | Slash

1-2 digits | Two digits

OR

The task of searching for a date needs more complex approach. Here’s just a simplifies sample.

Samples…

But what if we need date's parts also? We may use subpatterns.

```
<?php

$text = 'Date 1 is 28.02.1995. Date 2 is 2/28/95.';

// 10) Find all dates with parts:
preg_match_all("/((\d{2})\.(\d{2})\.(\d{4}))|((\d{1,2})\/(\d{1,2})\/(\d{2}))/", $text, $result);
print_r($result);
/*
[0] => Array ([0] => 28.02.1995, [1] => 2/28/95)
[1] => Array ([0] => 28.02.1995, [1] => )
[2] => Array ([0] => 28, [1] => )
[3] => Array ([0] => 02, [1] => )
[4] => Array ([0] => 1995, [1] => )
[5] => Array ([0] => [1] => 2/28/95)
[6] => Array ([0] => [1] => 2 )
[7] => Array ([0] => [1] => 28 )
[8] => Array ([0] => [1] => 95 )
*/
```

See previous sample.
Here each pair of ( )
produces a subpattern.

Functions to use with regexes

The most common PHP functions to use with regexes are:

- preg_match() – detects a match;
- preg_match_all() – returns matches;
- preg_replace() – replaces matches;
- preg_replace_callback() – replaces matches with a callback function result.

See the full list here: https://www.php.net/manual/en/ref.pcre.php

We'll see these functions in action in a couple of seconds…

And the last thing – modifiers

We'll see some of these modifiers in action in a couple of seconds…

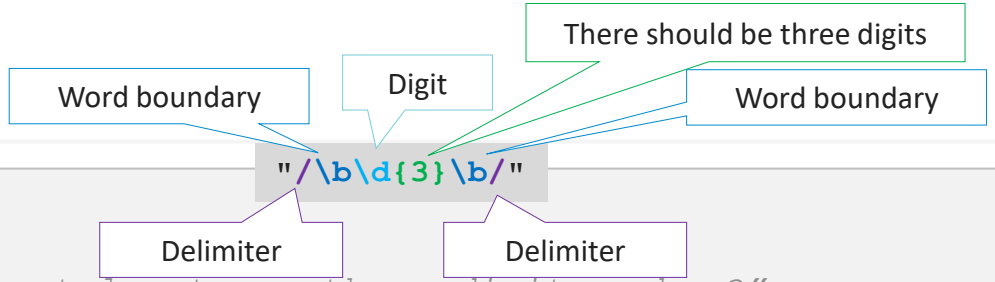
Modifiers may apply to the whole expression or to a group (see "internal options"), they are:

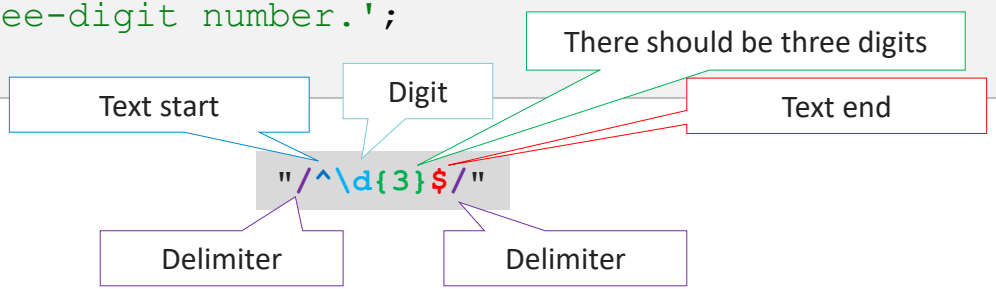
- i – enables case insensitivity;
- m – enables multiline-mode (PCRE starts recognizing "newlines");
- s – adds "\n" to the "any character" set ("." meta-character);
- x – enables comments inside a regexp;
- U – turns of the "greediness";
- u – switches PCRS functions to UTF-8 mode.

See details:
https://www.php.net/manual/en/reference.pcre.pattern.modifiers.php

And more samples…

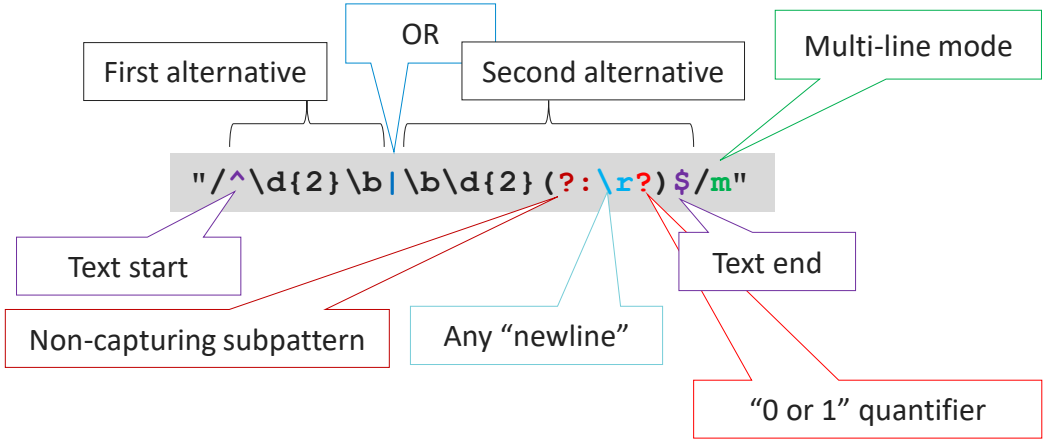
```php
<?php

// 1) "Does the text contain at least one three-digit number?"
$text = "This is some text with two numbers: 1234 and 567.";
if (preg_match("/\b\d{3}\b/", $text)) {
    echo 'There is a three-digit number in the text.';
}

// 2) "Is the string a three-digit number?"
$text = "890";
if (preg_match("/^\d{3}$/", $text)) {
    echo 'The string is a three-digit number.';
}
```





And more samples…

```
// 3) "Show all two-digits numbers that are at the beginning
// or at the end of a line".
$text = "12 34 56
78 90 21";
preg_match_all("/^\d{2}\b|\b\d{2}(?:\r?)$/m", $text, $result);
print_r($result);
// Array ([0] => 12, [1] => 56, [2] => 78, [3] => 21)
```



And more samples…

```
// 4) "Extract parts of dates in [D]D.[M]M.[YY]YY format."
$text = "1.7.99, 10.05.2001";
preg_match_all("/\b(\d{1,2})\.(\d{1,2})\.(\d{4}|\d{2})\b/", $text, $result, PREG_SET_ORDER);
print_r($result);
preg_match_all("/\b([0123]?\d)\.([01]?\d)\.((?:19|20)\d{2}|\d{2})\b/", $text, $result, PREG_SET_ORDER);
print_r($result);
// [0] => Array ([0] => 1.7.99, [1] => 1, [2] => 7, [3] => 99)
// [1] => Array ([0] => 10.05.2001, [1] => 10, [2] => 05, [3] => 2001)
```

The longest alternative should go BEFORE the shortest one

```
"/\b(\d{1,2})\.(\d{1,2})\.(\d{4}|\d{2})\b/"
```

Day

Month

Year

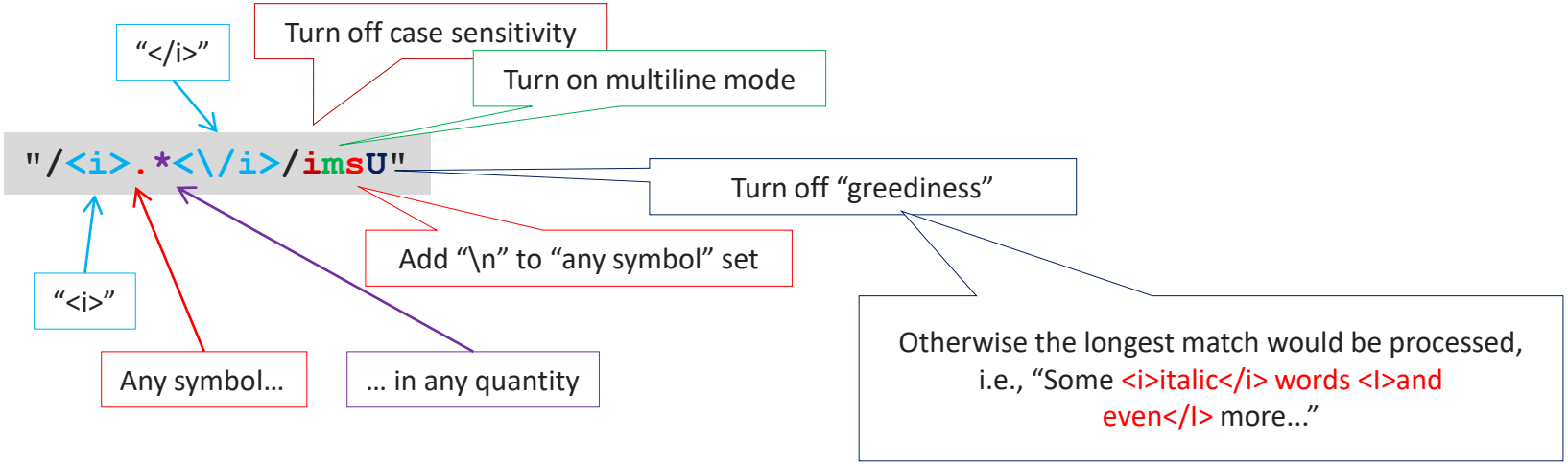
```
"/\b([0123]?\d)\.([01]?\d)\.((?:19|20)\d{2}|\d{2})\b/"
```

Just dots

Non-capturing subpattern. Otherwise we get “19” or “20” as a separate result.

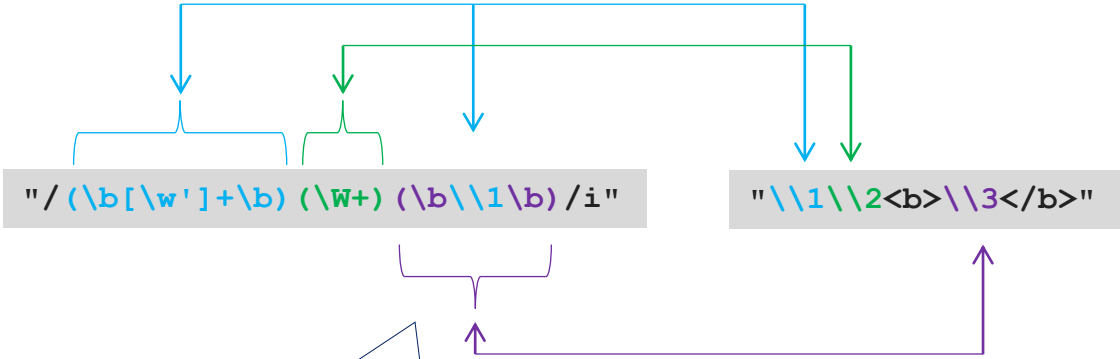
And more samples…

```
// 5) "Delete all italic text."
$text = "Some <i>italic</i> words <I>and
even</I> more...";
echo $text = preg_replace("/<i>.*<\/i>/imsU","", $text);
// Some  words  more...
```



And more samples…

```
// 6) "Mark duplicate words with <b> tag."
$text = "Some words words and even even more...";
echo $text = preg_replace("/(\b[\w']+\b)(\W+)(\b\\1\b)/i","\\1\\2<b>\\3</b>", $text);
// Some words <b>words</b> and even <b>even</b> more...
```



These \\1, \\2, \\3 parts are “back-references” (i.e., references to a data detected by a corresponding subpattern, where \\0 is the whole match, and \\1..99 are sub-matches.

And more samples…

```
// 7) "Is the string a number with ',' or ' ' as a groups separator?"
$test[] = '123456';
$test[] = '123 456';
$test[] = '12 34.56';
$test[] = '12 3 4.56';
$test[] = '1234.56';
$test[] = '1,234.56';
$test[] = '1,234';
foreach ($test as $number) {
    echo $number;
    if (preg_match("/^\d{1,3}([ ,]\d{3})*([.,]\d+)?$/", $number)) {
        echo " is OK\n";
    } else {
        echo " is NOT OK\n";
    }
}
// 123456 is NOT OK
// 123 456 is OK
// 12 34.56 is NOT OK
// 12 3 4.56 is NOT OK
// 1234.56 is NOT OK
// 1,234.56 is OK
// 1,234 is OK
```

Text start

1-3 digits

" " or "," followed by three digits

Text end

```
"/^\d{1,3}([ ,]\d{3})*([.,]\d+)?$/"
```

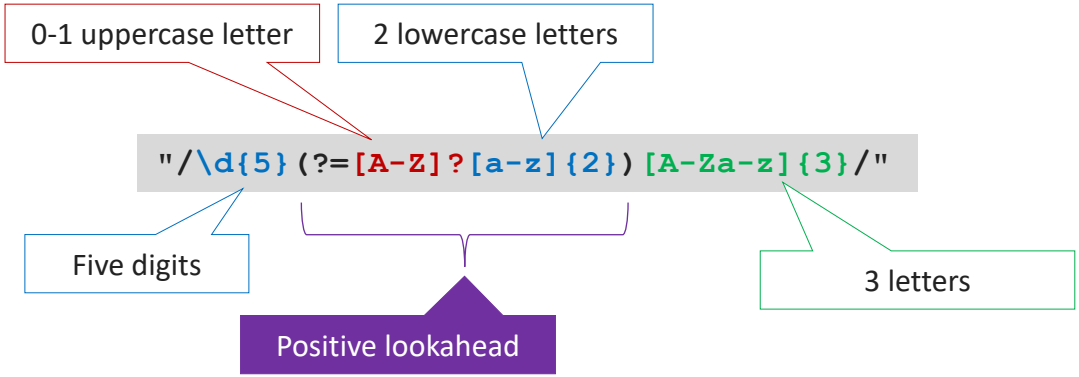
0..infinity quantifier

0..1 quantifier

The decimal part

And more samples…

```
// 8) "Show all product codes (5 digits, 3 letters,
// at least two sequential letters are in lowercase)."
$text = "12345aBc 12312deF 87654STU 12312Mnk 12312xYZ 45678gtk";
preg_match_all("/\d{5}(?=[A-Z]?[a-z]{2})[A-Za-z]{3}/", $text, $result);
print_r($result);
// Array ([0] => 12312deF, [1] => 12312Mnk, [2] => 45678gtk)
```



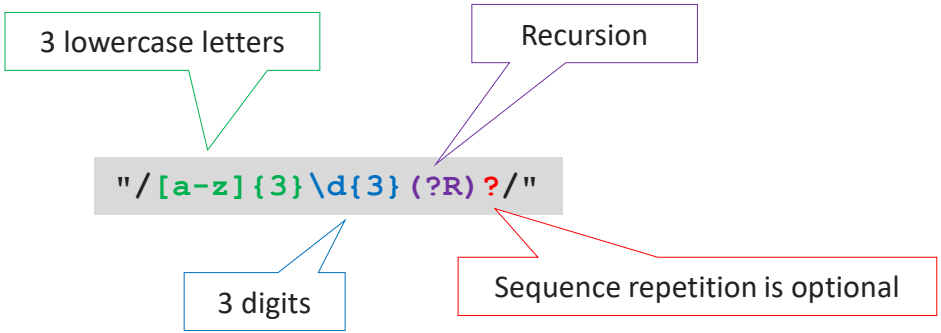
And more samples…

```
// 9) "Make all integers <b>bold</b> and all doubles <i>italic</i>
// (and round to one decimal digit)."
$text = "Just 12.233 some 45 text with 12.12.12 some numbers 99.";

function numbers(array $candidate): string
{
    $test = trim($candidate[0], '.');
    if (preg_match("/^\d+$/", $test)) {
        return '<b>' . $test . '</b>';
    } elseif (substr_count($test, '.') == 1) {
        return '<i>' . round($test, 1) . '</i>';
    } else {
        return $candidate[0];
    }
}

echo preg_replace_callback("/\b[\d.-]+\b/", 'numbers', $text);
// Just <i>12.2</i> some <b>45</b> text with 12.12.12 some numbers <b>99</b>.
```

Sometimes we may use trivial patterns and split the task into subtasks. E.g., here in the main regex we are just detecting “all that looks like a number”, and then (in the callback function) we perform final validation and processing.

And more samples…

```
// 10) "Find all sequences of 'three lowercase letters, three digits'
// with any quantity of repetitions."
$text = "Some aaa111 text bbb222ccc333 and dd444 and eee55 and fff555";
preg_match_all("/[a-z]{3}\d{3}(?R)?/", $text, $result);
print_r($result);
// Array ([0] => aaa111, [1] => bbb222ccc333, [2] => fff555)
```



P.S. More info

Refer to these sources:

- “Mastering Regular Expressions” (by Jeffrey Friedl)
- https://regexper.com
- https://regexr.com
- https://regexone.com
- https://www.php.net/manual/en/reference.pcre.pattern.syntax.php

Regular Expressions

Disclaimer: вы смотрите просто запись лекции, это **НЕ** специально подготовленный видеокурс!