

CNT 4714: Enterprise Computing

Fall 2011

Introduction to PHP – Part 5 – Pattern Matching

Instructor : Dr. Mark Llewellyn
 markl@cs.ucf.edu
 HEC 236, 407-823-2790
 <http://www.cs.ucf.edu/courses/cnt4714/fall2011>

Department of Electrical Engineering and Computer Science
Computer Science Division
University of Central Florida



Pattern Matching In PHP

- Many programming problems require matching or manipulating patterns in string variables. One reason to match patterns is to verify data received from an XHTML input form.
- For example, if you are expecting an XHTML form field to provide a U.S. telephone number as input, your script needs a way to verify that the input comprises a string of seven or ten digits.
- Another reason to match patterns arises when your script uses an input data file with fields that are delimited by characters such as colons or tabs.
- Pattern matching in PHP is handled via regular expressions.



Pattern Matching In PHP

- **Regular expressions (regex)** are one of the black arts of practical modern programming. Those who master regular expressions will find that they can solve many problems quite easily while those who don't will waste many hours pursuing complicated work-arounds.
- Regular expressions, although complicated, are not really difficult to understand. Fundamentally, they are a way to describe *patterns* of text using a single set of strings.
- Unlike a simple search-and-replace operations, such as changing all instances of "Marty" to "Mark", regex allow for much more flexibility – for example, finding all occurrences of the letters "Mar" followed by either "ty" or "k", and so on.



Pattern Matching In PHP

- Regular expressions were initially described in the 1950s by a mathematician named S.C. Kleene, who formalized models that were first designed by Warren McCulloch and Walter Pitts to describe the human nervous system.
- Regex were not actually applied to computer science until Ken Thompson (one of the original designers of the Unix OS) used them as a means to search and replace text in his *qed* editor.
- Regex eventually made their way into the Unix operating system (and later into the POSIX standard) and into Perl as well, where they are considered one of the language's strongest features.



Pattern Matching In PHP

- PHP actually supports both the POSIX standard and the Perl standard of regular expressions.
- The Perl version is known as PCRE (Perl-Compatible Regular Expressions).
- PCRE are much more powerful than their POSIX counterparts – and consequently more complex and difficult to use. You'll want to master POSIX regex before you attempt to work with PCRE.
- We'll look at the simpler POSIX form first and then look in more details at the PCRE format.



Pattern Matching In PHP

- Regex is, essentially, a whole new language, with its own rules, own structures, and its own quirks. What you know about other programming languages has little or no bearing on regex, for the simple reason that regular expressions are highly specialized and follow their own rules.

Regular Expression Axioms as defined by S. C. Kleene

- A single character is a regular expression denoting itself.
- A sequence of regular expressions is a regular expression.
- Any regular expression followed by a * character (known as the “Kleene Star”) is a regular expression composed of zero or more instances of that regular expression.
- Any pair of regular expressions separated by a pipe character (|) is a regular expression composed of either the left or the right regular expression.
- Parentheses can be used to group regular expressions.



Pattern Matching In PHP

- While Kleene's definition of what makes a regular expression might, at first, seem confusing, the basics are actually pretty easy to understand.
- First, the simplest regular expression is a single character. For example, the regex `a` would match the character "a" in the word "Mark".
- Next, single character regex can be grouped by placing them next to each other. Thus the regex `Mark` would match the word "Mark" in "Your instructor is Mark for CIS 4004."
- So far, regex are not very different from normal search operations. However, this is where their similarities end.



Pattern Matching In PHP

- The Kleene Star can be used to create regex that can be repeated any number of times (including none).
- Consider the following string:

```
seeking the treasures of the sea
```

- The regex `se*` will be interpreted as “the letter `s` followed by zero or more instances of the letter `e`” and will match the following:
 - The letters “see” of the word “seeking”, where the regex `e` is repeated twice.
 - Both instances of the letter `s` in “treasures”, where `s` is followed by zero instances of `e`.
 - The letters “se” of the word “sea”, where the `e` is present once.



Pattern Matching In PHP

- It's important to understand in the regex se^* that only the expression e is considered with dealing with the star.
- Although its possible to use parentheses to group regular expressions, you should not be tempted to think that using $(se)^*$ is a good idea, because the regex compiler will interpret it as meaning “zero or more occurrences of se ”.
- If you apply this regex to the same string, you will encounter a total of 32 matches, because every character in the string would match the expression. (Remember? 0 or more occurrences!)



Pattern Matching In PHP

- You'll find parentheses are often used in conjunction with the pipe operator to specify alternative regex specifications.
- For example, the regex `gr(u|a)b` with the string: “grab the grub and pull” would match both “grub” and “grab”.
- Although regular expressions are quite powerful because of the original rules, inherent limitations make their use impractical.
- For example, there is no regular expression that can be used to specify the concept of “any character”.
- As a result of the inherent limitations, the practical implementations of regex have grown to include a number of other rules, the most common of which are shown beginning on the next page.



Additional Syntax For Regex

- The special character “^” is used to identify the beginning of the string.
- The special character “\$” is used to identify the end of the string.
- The special character “.” is used to identify any character.
- Any nonnumeric character following the character “\” is interpreted literally (instead of being interpreted according to its regex meaning). Note that this escaping sequence is relative to the regex compiler and not to PHP. This means that you must ensure that an actual backslash character reaches the regex functions by escaping it as needed (i.e., if you’re using double quotes, you will need to input `\\`).



Additional Syntax For Regex

- Any regular expression followed by a “+” character is a regular expression composed of one or more instances of that regular expression.
- Any regular expression followed by a “?” character is a regular expression composed of either zero or one instance of that regular expression.
- Any regular expression followed by an expression of the type {min [, |, max]} is a regular expression composed of a variable number of instances of that regular expression. The min parameter indicates the minimum acceptable number of instances, whereas the max parameter, if present, indicates the maximum acceptable number of instances. If only the comma is present, no upper limit exists. If only min is defined, it indicates the only acceptable number of instances.
- Square brackets can be used to identify groups of characters acceptable for a given character position.



Some Basic Regex Usage

- It's sometimes useful to be able to recognize whether a portion of a regular expression should appear at the beginning or the end of a string.
- For example, suppose you're trying to determine whether a string represents a valid HTTP URL. The regex `http://` would match both `http://www.cs.ucf.edu`, which is valid and `nhttp://www.cs.ucf.edu` which is not valid, and could easily represent a typo on the user's part.
- Using the special character “`^`”, you can indicate that the following regular expression should only be matched at the beginning of the string. Thus, `^http://` will match only the first of our two strings.



Some Basic Regex Usage

- The same concept – although in reverse – applies to the end-of-string marker “\$”, which indicates that the regular expression preceding it must end exactly at the end of the string.
- Thus, `com$` will match “amazon.com” but not “communication”.
- Having a “wildcard” that can be used to match any character is extremely useful in a wide range of scenarios, particularly considering that the “.” character is considered a regular expression in its own right, so that it can be combined with the Kleene Star and any of the other modifiers.



Some Basic Regex Usage

- Consider the regex: `.+@.+\. .+`
- This regex can be used to indicate:
 - At least one instance of any character, followed by
 - The @ character, followed by
 - At least one instance of any character, followed by
 - The “.” character, followed by
 - At least one instance of any character
- Can you guess what sort of string this regex might validate?

Does this look familiar? `markl@cs.ucf.edu`

It's a very rough form of an email address. Notice how the backslash character was used to force the regex compiler to interpret the next to last “.” as a literal character, rather than as another instance of the “any character” regular expression.



Some Basic Regex Usage

- The regex on the previous page is a fairly crude way of checking the validity of an email address. After all, only letters of the alphabet, the underscore character, the minus character, and digits are allowed in the name, domain, and extension of an email.
- This is where the **range denominators** come into play. As mentioned previously (last paragraph of page 12), anything within non-escaped square brackets represents a set of alternatives for a particular character position. For example, the regex `[abc]` indicated either an “a”, a “b”, or a “c” character. However, representing something like “any character” by including every possible symbol in the square brackets would give rise to some ridiculously long regular expressions.



Some Basic Regex Usage

- Fortunately, range denominators make it possible to specify a “range” of characters by separating them with a dash.
- For example `[a-z]` means “any lowercase character.
- You can also specify more than one range and combine them with individual characters by placing them side-by-side.
- For example, our email validation regex could be satisfied by the expression `[A-Za-z0-9_]`.
- Using this new tool our full email validation expression becomes:

```
[A-Za-z0-9_]+@ [A-Za-z0-9_]+\ . [A-Za-z0-9_]+
```



Some Basic Regex Usage

- The range specifications that we have seen so far are all *inclusive* – that is, they tell the regex compiler which characters *can* be in the string. Sometimes, its more convenient to use *exclusive* specification, dictating that any character *except* the characters you specify are valid.
- This is done by prepending a caret character (^) to the character specifications inside the square bracket.
- For example, [^A-Z] means any character except any uppercase letter of the alphabet.



Some Basic Regex Usage

- Going back to our email example, its still not as good as it could be because we know for sure that a domain extension must have a minimum of two characters and a maximum of four.
- We can further modify our regex by using the minimum-maximum length specifier introduced on page 12.

```
[A-Za-z0-9_]+@[A-Za-z0-9_]+\.[A-Za-z0-9_]{2,4}
```

- Naturally, you might want to allow only email addresses that have a three-letter domain. This can be accomplished by omitting the comma and the `max` parameter from the length specifier, as in:

```
[A-Za-z0-9_]+@[A-Za-z0-9_]+\.[A-Za-z0-9_]{3}
```



Some Basic Regex Usage

- On the other hand, you might want to leave the maximum number of characters open in anticipation of the fact that longer domain extensions might be introduced in the future, so you could use the regex:

```
[A-Za-z0-9_]+@[A-Za-z0-9_]+\.[A-Za-z0-9_]{3,}
```

- Which indicates that the last regex in the expression should be repeated at least a minimum of three times, with no fixed upper limit.



POSIX Regular Expressions

- POSIX (**P**ortable **O**perating **S**ystem **I**nterface for **uniX**) is a collection of standards that define some of the functionality that a Unix operating system should support.
- One of these standards defines two flavors of regular expressions.
 - BRE (Basic Regular Expressions) standardizes a flavor similar to the one used by the traditional Unix `grep` command. This is probably the oldest regular expression flavor still in use today.
 - ERE (Extended Regular Expressions) standardizes a flavor similar to the one used by the Unix `egrep` command. Most modern regex flavors are extensions of the ERE flavor.
- The POSIX standard is the simplest form of regex available in PHP (as opposed to the PCRE), and as such is the best way to learn regular expressions.



POSIX Regular Expressions

- In addition to the standard rules of regex that we've already discussed, the POSIX regex standard defines the concept of **character classes** as a way to make it even easier to specify character ranges.
- Character classes are always enclosed in a set of colon characters (:) and must be enclosed in square brackets.
- There are 12 character classes defined in the POSIX standard. These are listed in the table on the following page.



Character class	Description
alpha	Represents a letter of the alphabet (either lower or upper case). Equivalent to [A-Za-z]
digit	Represents a digit between 0 and 9. Equivalent to [0-9]
alnum	Represents an alphanumeric character. Equivalent to [0-9A-Za-z]
blank	Represents “blank” characters, normally space and tab
cntrl	Represents “control” characters, such as DEL, INS, and so on
graph	Represents all printable characters except the space
lower	Represents lowercase letters of the alphabet only
upper	Represents uppercase letters of the alphabet only
print	Represents all printable characters
punct	Represent punctuation characters such as “.”, or “,”
space	Represents the whitespace
xdigit	Represents hexadecimal digits

The POSIX character classes



POSIX Regular Expressions

- Rewriting our previous email regex using the POSIX standard notation the following:

```
[A-Za-z0-9_]+@[A-Za-z0-9_]+\.[A-Za-z0-9_]{2,4}
```

becomes:

```
[[:alnum:]]+@[[:alnum:]]+\.[[:alnum:]]{2,4}
```

- This notation is a bit simpler, and it unfortunately also makes mistakes a little less obvious.



POSIX Regular Expressions

- Another important concept introduced by the POSIX standard is the [reference](#).
- Recall that we discussed the use of parentheses to group regular expressions (see page 6 – one of Kleene’s original regex axioms).
- When you use parentheses in a POSIX regex, when the expression is executed the interpreter assigns a numeric identifier to each grouped expression that is matched.
- This identifier can be used in various operations – such as finding and replacing.
- Consider the example on the following page:



POSIX Regular Expressions

- Suppose we have the string: `markl@cs.ucf.edu` and the regex:

```
([[:alnum:]]_+ )@([[:alnum:]]_+ )\.[[:alnum:]]_+ {2,4}
```

- The regex should match the email address string. However, because we have grouped the username, domain name, and the domain extensions, they will each become a reference, as shown in the table below:

Reference Number	Value
0	<code>markl@cs.ucf.edu</code> (string matches the entire regex)
1	<code>markl</code>
2	<code>cs.ucf</code>
3	<code>edu</code>



POSIX Regular Expressions

- PHP provides support for POSIX through functions of the `ereg*` class.
- Unfortunately, as of PHP 5.3.0 the `ereg*` class has been deprecated and is no longer being supported by PHP. This means that you don't want to develop new code using this class. However, for the time being at least, you can get a brief introduction to regex using the class if you don't mind seeing a warning message in your output. We'll go ahead and use this class of functions for the time being before we look at the PCRE class of functions which have replaced the `ereg*` class.



POSIX Regular Expressions

- The simplest form of regex matching is performed through the `ereg()` function which has the following form:

```
ereg(pattern, string[, matches]);
```

- The `ereg()` function works by compiling the regular expression stored in `pattern` and then comparing it against `string`. If the regex is matched against `string`, the result value of the function is `true` – otherwise, it is `false`. If the `matches` parameter is specified, it is filled with an array containing all the references specified by `pattern` that were found in `string`. Position 0 in this array represents the entire matched string.
- An example is shown on the next page.



```
C:\Program Files\Apache Software Foundation\Apache2.2\htdocs\CNT4714\PHP\ereg() example 1.php - Notepad++
File Edit Search View Encoding Language Settings Macro Run Plugins Window ?
ereg() example 1.php  ereg() example 2.php  pcre example 1.php  pcre reference - version 2.php
4 </head>
5 <body style = "font-family: arial, sans-serif;
6     background-color: #856363" background=image1.jpg>
7 <?php
8     function var_dump_pre($mixed = null) {
9         echo '<pre>';
10        var_dump($mixed);
11        echo '</pre>';
12        return null;
13    }
14
15    $str = 'markl@cs.ucf.edu';
16    if (ereg('([[:alpha:]]+)([[:alpha:]]+)\.([[:alpha:]]{2,4})\.([[:alpha:]]{2,4})'
17        , $str, $matches)){
18        print("Regular expression successful. Dumping matches: <br />");
19        var_dump_pre($matches);
20    }
21    else {
22        print("Regular expression unsuccessful...no match.\n");
23    }
24    ?>
25 </body>
26 </html>
```

PHP Hypertext Preprocessor | length : 623 lines : 28 | Ln : 20 Col : 1 Sel : 0 | UNIX | ANSI | INS



Regular expression successful. Dumping matches:

```
array(5) {  
  [0]=>  
  string(16) "markl@cs.ucf.edu"  
  [1]=>  
  string(5) "markl"  
  [2]=>  
  string(2) "cs"  
  [3]=>  
  string(3) "ucf"  
  [4]=>  
  string(3) "edu"  
}
```



A Practice Exercise

- See if you can create a POSIX based regex that will validate a string representing a date in the format `mm/dd/yyyy`. In other words, `04/05/2011` would be matched but `4/5/11` would not.
- Step 1: form a basic regex. A regex such as `.+` (one or more characters) is a bit too vague even as a starting point. So how about something like this?

```
[[:digit:]]{2}/[[:digit:]]{2}/[[:digit:]]{4}
```

- This will work and validate `04/05/2011`. However, it will also validate `99/99/2011` which is not a valid date, so we still need some refinement.



A Practice Exercise (continued)

- For the month component of our regex, the first digit must always be either a 0 or a 1, but the second digit can be any of 0 through 9.
- Similarly, for the day component of the regex, the first digit can only be 0, 1, 2, or 3.
- Our final regex now becomes:

```
[0-1][[:digit:]]/[0-3][[:digit:]]/[[:digit:]]{4}
```

- This will work and validate 04/05/2011.



Perl-Compatible Regular Expressions (PCRE)

- Perl-Compatible Regular Expressions (PCRE) are much more powerful than their POSIX counterparts. This of course makes them more complex and difficult to use as well, but well worth the effort for PHP programmers/scripters.
- PCRE adds its own character classes to the extended regular expressions available in the POSIX standard.
- There are nine of these character classes in PCRE and are shown in the table on the next page.



Perl-Compatible Regular Expressions (PCRE)

Character class	Description
<code>\w</code>	Represents a “word” character and is equivalent to the expression <code>[A-Za-z0-9]</code>
<code>\W</code>	Represents the opposite of <code>\w</code> and is equivalent to the expression <code>[^A-Za-z0-9]</code>
<code>\s</code>	Represents a whitespace character
<code>\S</code>	Represents a non-whitespace character
<code>\d</code>	Represents a digit and is equivalent to the expression <code>[0-9]</code>
<code>\D</code>	Represents a non-digit (the opposite of <code>\w</code>) and is equivalent to the expression <code>[^0-9]</code>
<code>\n</code>	Represents a new line character
<code>\r</code>	Represents a return character
<code>\t</code>	Represents a tab character

PCRE character classes



Perl-Compatible Regular Expressions (PCRE)

- Using PCRE formatted regex allows for significantly more concise regex than is possible for the POSIX formatted regex.
- Consider, for example, the email address validation expression we developed in POSIX earlier:

```
[[:alnum:]]_]+@[[:alnum:]]_]+\.[[:alnum:]]_{2,4}
```

- Using the new character classes of PCRE this expression becomes:

```
/\w+@\w+\.\w{2,4}/
```

Note in the example script on the next page I added another `\w{2,4}` term so that I could easily pickup the sub-domain used in my email address.

Notice that the regex string now begins and ends with forward slashes. PCRE requires that the actual regular expression be delimited by two characters. By convention, two forward slashes are used, although any character other than the backslash that is not alphanumeric would work just as well.



```
C:\Program Files\Apache Software Foundation\Apache2.2\htdocs\CNT4714\PHP\preg example 1.php - Notepad++
File Edit Search View Encoding Language Settings Macro Run Plugins Window
ereg() example 1.php ereg() example 2.php pcre example 1.php pcre referer
1 <html>
2 <head>
3     <title> Using PCRE in PHP - example 1 </title>
4 </head>
5 <body style = "font-family: arial, sans-serif; background-color: #856363" background-color: #856363">
6
7 <?php
8     function var_dump_pre($mixed = null) {
9         echo '<pre>';
10        var_dump($mixed);
11        echo '</pre>';
12        return null;
13    }
14
15    $str = 'markl@cs.ucf.edu';
16    if (preg_match('/\w+@\w+\.\w{2,4}\.\w{2,4}/', $str, $matches)){
17        print("Regular expression successful. Dumping matches: \n");
18        var_dump_pre($matches);
19    }
20    else {
21        print("Regular expression unsuccessful...no match.\n");
22    }
23 ?>
24 </body>
</html>
```

Using PCRE in PHP - example 1 - Opera

Opera Using PCRE in PHP - example 1 - Opera localhost:8081/CNT4

Regular expression successful. Dumping matches:

```
array(1) {
  [0]=>
    string(16) "markl@cs.ucf.edu"
}
```



Perl-Compatible Regular Expressions (PCRE)

- Regardless of which character you use to delimit your PCRE regex (I suggest you stick with the convention however), you will need to escape the delimiter whenever you use it as part of the regex itself.
- For example, `/face\/off/` would be the PCRE equivalent to the regex `face/off`.
- PCRE also extends the concept of references by making them useful not only as a byproduct of the regex operation, but as part of the operation itself.
- In PCRE, it is possible to use a reference that was defined previously in a regular expression as part of the expression itself. Consider the following example:



Perl-Compatible Regular Expressions (PCRE)

- Suppose you have a situation where you need to verify that in strings such as:

Mark is a cyclist. Mark's specialty is road racing.

Karen is a cyclist. Karen's specialty is road racing.

the name of the person to whom the sentence refers is the same in both positions.

- Using a normal search-and-replace operation would take a significant effort, and so would using a POSIX regex, simply because you do not know the name of the person in advance.
- With PCRE this is a trivial task because you simply use a reference within the regex as shown on the next page.



Perl-Compatible Regular Expressions (PCRE)

- You start by matching the first portion of the string. The name is the first word:

```
/^(\w+) is a cyclist.
```

Next, you need to specify the name again, however, since we enclosed the first instance of the name in parentheses we created a reference. In the subsequent part of the expression you simply recall that reference inside the regex itself and use it as needed.

```
/^(\w+) is a cyclist. \1's specialty is road racing.
```

- The next page shows a PHP script that uses this example.



```
10     var_dump($mixed);
11     echo '</pre>';
12     return null;
13 }
14
15 $str = 'Mark is a cyclist. Mark\'s specialty is road racing.';
16 if (preg_match('/^(\\w+) is a cyclist. \\1\'s specialty is road racing./',
17     $str, $matches)){
18     print("Regular expression successful. Dumping matches: <br />");
19     var_dump_pre($matches);
20 }
21 else {
22     print("Regular expression unsuccessful...no match.<br />");
23 }
24 $str = 'Mark is a cyclist. Karen\'s specialty is road racing.';
25 if (preg_match('/^(\\w+) is a cyclist. \\1\'s specialty is road racing./', $str, $r
26     print("Regular expression successful. Dumping matches: <br />");
27     var_dump_pre($matches);
28 }
29 else {
30     print("Regular expression unsuccessful...no match.\\n");
31 }
32 ?>
```



Using PCRE With References in PHP - Opera

Opera Using PCRE With Refere... x

Web localhost:8081/CNT4714/PHP/pcrereference-%20version%202.

Regular expression successful. Dumping matches:

```
array(2) {
  [0]=>
  string(51) "Mark is a cyclist. Mark's specialty is road racing."
  [1]=>
  string(4) "Mark"
}
```

Regular expression unsuccessful...no match.

First case works fine since Mark is on the second name position and matches with reference 1.

Second case doesn't match since Karen is in the second name position and no match with reference 1.



Perl-Compatible Regular Expressions (PCRE)

- To illustrate the power of the PCRE version of regex in PHP, the next page provides a POSIX version of the previous example of matching the two subjects.
- The conciseness of the PCRE version of regex should be apparent after looking at this script.



```

7  <?php
8  # This is the PCRE version
9  print("This is the PCRE version. \n");
10 $str = 'Mark is a cyclist. Mark\'s specialty is road racing.';
11 if (preg_match('/^(\\w+) is a cyclist. \\1\'s specialty is road racing./', $str, $matches)){
12     print("Regular expression successful. Dumping matches: \n");
13     var_dump($matches);
14 }
15 else {
16     print("Regular expression unsuccessful...no match.\n");
17 }
18 # This is the POSIX version
19 print("This is the POSIX version.\n");
20 $str = 'Mark is a cyclist. Mark\'s specialty is road racing.';
21 if (ereg('^([[:alpha:]]+) is a cyclist.', $str, $matches1)){
22     if (ereg('([[:alpha:]]+)\\'s specialty is road racing.', $str, $matches2)) {
23         if ($matches1[1] === $matches2[1]) {
24             print("Regular expression successful. Two patterns match.\n");
25         }
26         else {
27             print ("Regular expression fails.  Two patterns do not match.\n");
28         }
29     }
30     else {
31         print ("Regular expression fails.  Two patterns do not match.\n");

```



Using PCRE With References in PHP - Opera

Using PCRE With Refere...

localhost:8081/CNT4714/PHP/posix%20version%20of%20pcre%20versic

Search with Google

This is the PCRE version. Regular expression successful. Dumping matches:

```
array(2) {  
  [0]=>  
  string(51) "Mark is a cyclist. Mark's specialty is road racing."  
  [1]=>  
  string(4) "Mark"  
}
```

This is the POSIX version. Regular expression successful. Two patterns match.

The POSIX version works but is clearly more complex and requires more code.



Perl-Compatible Regular Expressions (PCRE)

- As the previous example illustrates, PHP provides support for PCRE-formatted regular expression through the `preg*` class of function.
- The main PCRE function in PHP is `preg_match()` which has the following basic syntax:

```
preg_match( pattern, string [, matches [, flags] ] );
```

- As with the `ereg()` function for the POSIX standard, the `preg_match()` function causes the regex stored in `pattern` to be matched against `string`, and any references matched are stored in `matches`.



Perl-Compatible Regular Expressions (PCRE)

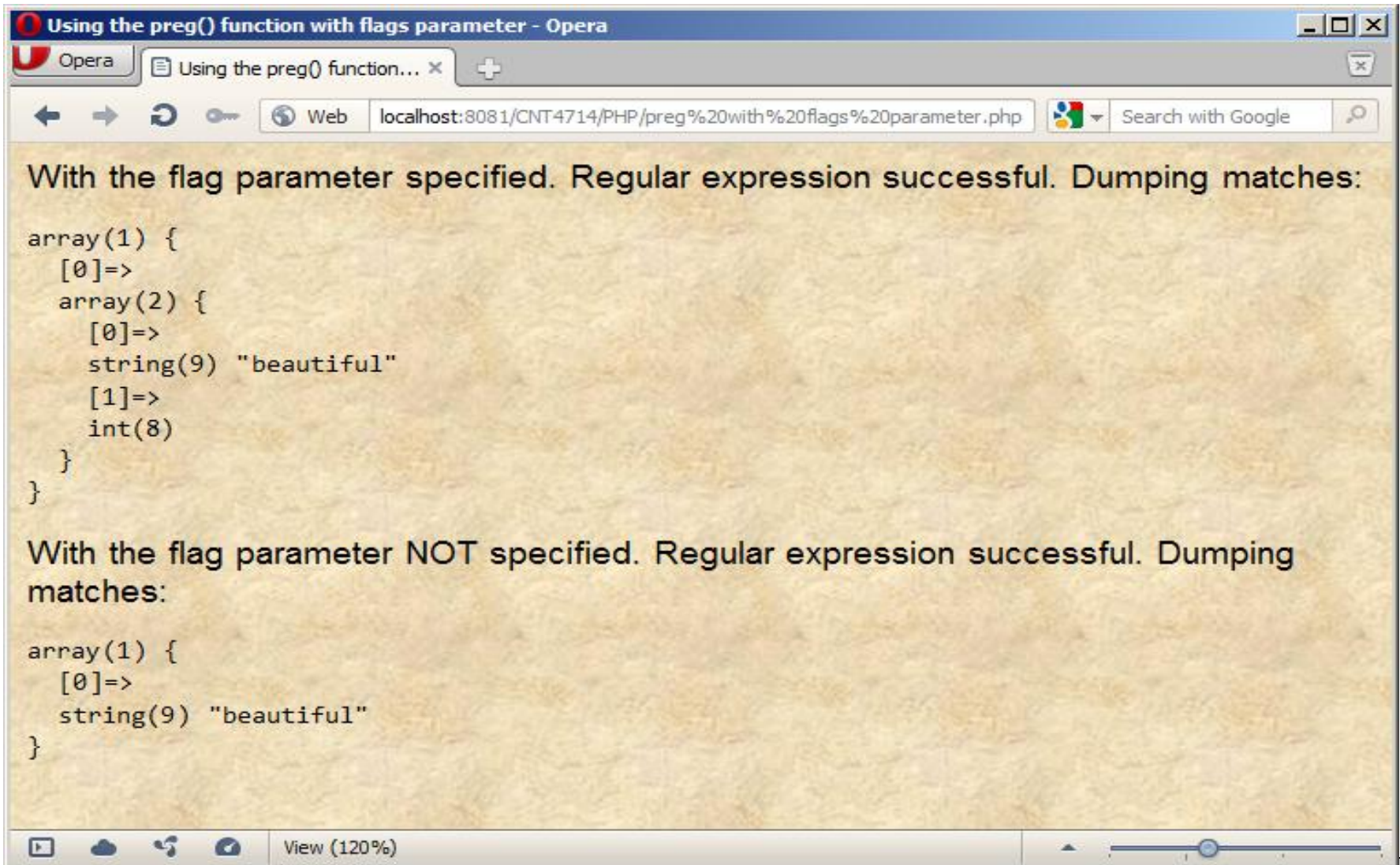
- The optional `flags` parameter can, for the time being, only contain the value `PREG_OFFSET_CAPTURE`.
- If this parameter is specified, it will cause `preg_match()` to change the format of matches so that it will contain both the text and the position of each reference inside the string.
- When this parameter is specified, the `matches` array will contain another array for each reference. The latter array, in turn, contains both the string matched and its position within the original string.
- The example on the following page illustrates both cases.



```

7  <?php
8  function var_dump_pre($mixed = null) {
9      echo '<pre>';
10     var_dump($mixed);
11     echo '</pre>';
12     return null;
13 }
14 $str = 'Another beautiful day!';
15 # With the flag parameter specified
16 print("With the flag parameter specified. \n\n\n");
17 if (preg_match('/beautiful/', $str, $matches, PREG_OFFSET_CAPTURE)){
18     print("Regular expression successful. Dumping matches: \n");
19     var_dump_pre($matches);
20 }
21 else {
22     print("Regular expression unsuccessful...no match.\n");
23 }
24 # With the flag parameter NOT specified
25 print("With the flag parameter NOT specified. \n");
26 if (preg_match('/beautiful/', $str, $matches)){
27     print("Regular expression successful. Dumping matches: \n");
28     var_dump_pre($matches);
29 }
30 else {
31     print("Regular expression unsuccessful...no match.\n");
32 }
    
```

Perl-Compatible Regular Expressions (PCRE)



The screenshot shows a browser window titled "Using the preg() function with flags parameter - Opera". The address bar shows the URL "localhost:8081/CNT4714/PHP/preg%20with%20flags%20parameter.php". The page content displays two examples of PHP's `preg()` function output:

With the flag parameter specified. Regular expression successful. Dumping matches:

```
array(1) {
  [0]=>
  array(2) {
    [0]=>
    string(9) "beautiful"
    [1]=>
    int(8)
  }
}
```

With the flag parameter NOT specified. Regular expression successful. Dumping matches:

```
array(1) {
  [0]=>
  string(9) "beautiful"
}
```



Perl-Compatible Regular Expressions (PCRE)

- Another very useful function in the `preg*` family is `preg_match_all()`, which has the same syntax as the `preg_match()` function, but searches a string for all the occurrences of the regular expression, rather than for a specific instance.
- The example on the following page illustrates the `preg_match_all()` function.



```
1 <html>
2 <head>
3     <title> Using the preg_match_all() function </title>
4 </head>
5 <body style = "font-family: arial, sans-serif;
6     background-color: #856363" background=image1.jpg>
7 <?php
8     function var_dump_pre($mixed = null) {
9         echo '<pre>';
10        var_dump($mixed);
11        echo '</pre>';
12        return null;
13    }
14    $str = 'Another beautiful day spent with a beautiful lady on a beauty of a lake!';
15    if (preg_match_all('/beaut[^\ ]+/', $str, $matches, PREG_OFFSET_CAPTURE)){
16        print("Regular expression successful. Dumping matches: <br />");
17        var_dump_pre($matches);
18    }
19    else {
20        print("Regular expression unsuccessful...no match.<br />");
21    }
22    ?>
23 </body>
24 </html>
25
26
```



Perl-Compatible Regular Expressions (PCRE)



```
Using the preg_match_all() function - Opera
Opera Using the preg_match_al... x
Web localhost:8081/CNT4714/PHP/preg_match_all%20example.php Search with Google

Regular expression successful. Dumping matches:

array(1) {
  [0]=>
  array(3) {
    [0]=>
    array(2) {
      [0]=>
      string(9) "beautiful"
      [1]=>
      int(8)
    }
    [1]=>
    array(2) {
      [0]=>
      string(9) "beautiful"
      [1]=>
      int(35)
    }
    [2]=>
    array(2) {
      [0]=>
      string(6) "beauty"
      [1]=>
      int(55)
    }
  }
}
```



Perl-Compatible Regular Expressions (PCRE)

- Search and replace operations using PCRE regex are handled by the `preg_replace()` function. This function has the following syntax:

```
preg_replace(pattern, replacement, string [, limit]);
```

- This function applies the regex `pattern` to `string` and then substitutes the placeholders in `replacement` with the references defined in it. The `limit` parameter can be used to limit the number of replacements to a maximum number.
- The example on the following page illustrates three different applications of this function. The first two simply replaces a single word in the string, while the third replaces the entire string.



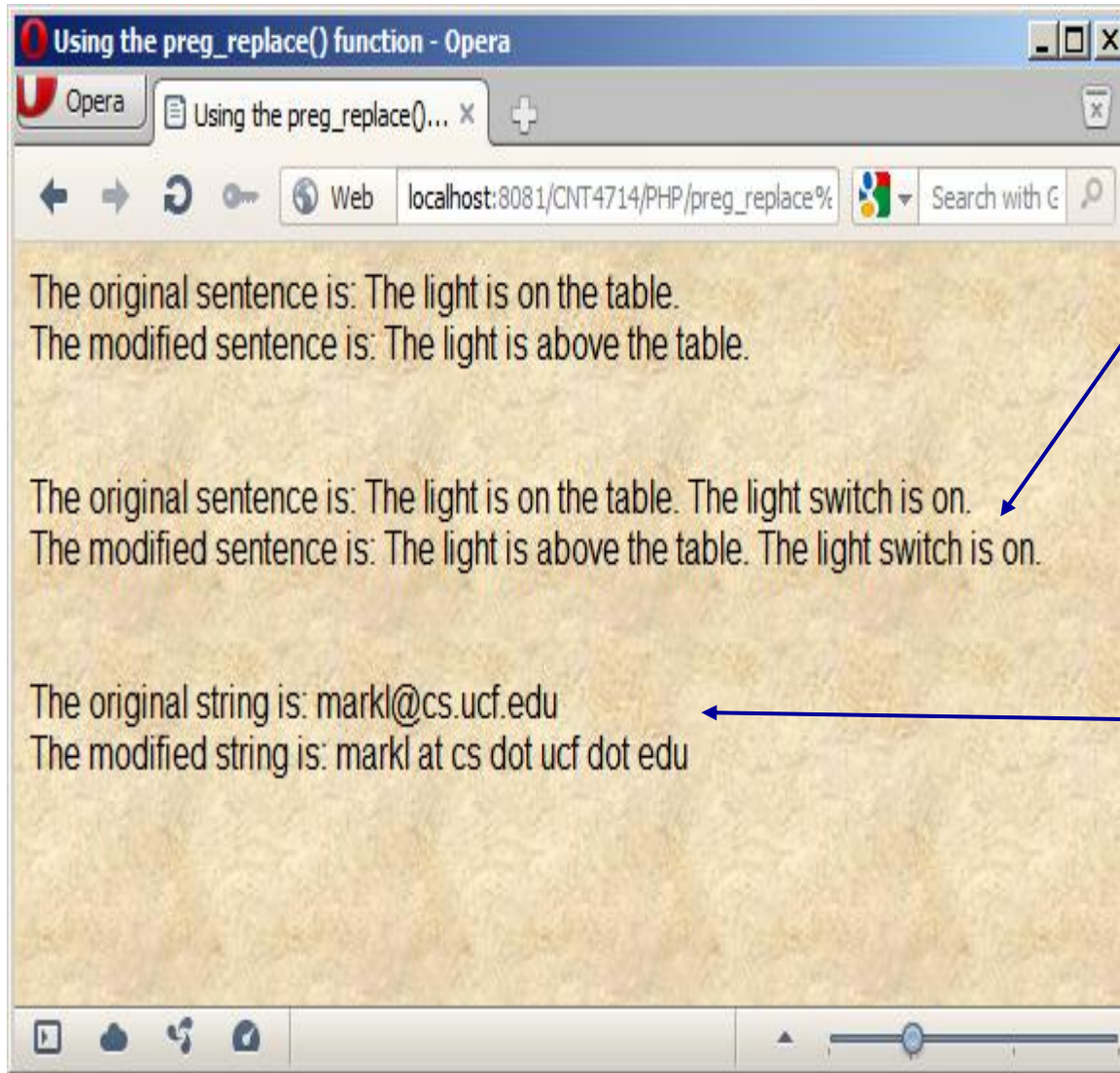


```

1 <html>
2 <head>
3     <title> Using the preg_replace() function </title>
4 </head>
5 <body style = "font-family: arial, sans-serif;
6     background-color: #856363" background=image1.jpg>
7 <?php
8     $str = 'The light is on the table.';
9     print("The original sentence is: $str <br />");
10    $modstr = preg_replace('/on/', 'above', $str);
11    print("The modified sentence is: $modstr <br /><br /><br />");
12    $str = 'The light is on the table. The light switch is on.';
13    print("The original sentence is: $str <br />");
14    $modstr = preg_replace('/on/', 'above', $str, 1);
15    print("The modified sentence is: $modstr <br /><br /><br />");
16    $str = "markl@cs.ucf.edu";
17    print("The original string is: $str <br />");
18    $modstr = preg_replace('/^(\\w+)@(\\w+)\\. (\\w{2,4})\\. (\\w{2,4})/', '\\1 at \\2 dot \\3 dot \\4', $s
19    print("The modified string is: $modstr");
20 ?>
21 </body>
22 </html>
  
```



Perl-Compatible Regular Expressions (PCRE)



The screenshot shows a web browser window titled "Using the preg_replace() function - Opera". The address bar shows "localhost:8081/CNT4714/PHP/preg_replace%". The page content is as follows:

The original sentence is: The light is on the table.
The modified sentence is: The light is above the table.

The original sentence is: The light is on the table. The light switch is on.
The modified sentence is: The light is above the table. The light switch is on.

The original string is: markl@cs.ucf.edu
The modified string is: markl at cs dot ucf dot edu

With the limit parameter set to 1 only the first occurrence of "on" is replaced.

Entire strings are replaced in this case.



Perl-Compatible Regular Expressions (PCRE)

- The last `preg*` function that we'll look at is `preg_split()`, which has the following syntax:

```
preg_split(pattern, string [, limit [, flags] ] );
```

- The `preg_split()` function works by breaking `string` in substrings delimited by sequences of characters delimited by `pattern`. The optional `limit` parameter can be used to specify a maximum number of splitting operations (by default a value of -1, 0, or null means no limit). The `flags` parameter, on the other hand is used to modify the behavior of the function as described in the table on the next page.
- An example using the `preg_split()` function appears on page 57.



Perl-Compatible Regular Expressions (PCRE)

Flag Value	Description of Function Behavior
<code>PREG_SPLIT_NO_EMPTY</code>	Causes empty substrings to be discarded.
<code>PREG_SPLIT_DELIM_CAPTURE</code>	Causes any reference inside <code>pattern</code> to be captured and returned as part of the function's output.
<code>PREG_SPLIT_OFFSET_CAPTURE</code>	Causes the position of each substring to be returned as part of the function's output (similar to <code>PREG_OFFSET_CAPTURE</code> in <code>preg_match()</code>).

Flag values for the `preg_split()` function





```
5 <body style = "font-family: arial, sans-serif;
6 background-color: #856363" background=image1.jpg>
7 <?php
8 function var_dump_pre($mixed = null) {
9     echo '<pre>';
10    var_dump($mixed);
11    echo '</pre>';
12    return null;
13 }
14 $str = 'Ten times she called, and ten times nobody answered.';
15 # This causes the string to be split whenever a space or a comma is matched.
16 var_dump_pre (preg_split('/[ \s,]/', $str));
17 # This version casues teh string to be split only on a comma regardless of whitespace before
18 var_dump_pre (preg_split("/[\s]*[,][\s]*/", $str));
19
20 $str = 'eXtensible HyperText Markup Language';
21 # This causes the string to be split into its component words and retain their starting pos:
22 var_dump_pre (preg_split('/\s/', $str, -1, PREG_SPLIT_OFFSET_CAPTURE));
23
24 ?>
25 </body>
26 </html>
```



```

array(12) {
  [0]=>
  string(3) "Ten"
  [1]=>
  string(5) "times"
  [2]=>
  string(3) "she"
  [3]=>
  string(6) "called"
  [4]=>
  string(0) ""
  [5]=>
  string(0) ""
  [6]=>
  string(0) ""
  [7]=>
  string(3) "and"
  [8]=>
  string(3) "ten"
  [9]=>
  string(5) "times"
  [10]=>
  string(6) "nobody"
  [11]=>
  string(9) "answered."
}

```

In version 1, any whitespace character or a comma forces a split.

```

array(2) {
  [0]=>
  string(20) "Ten times she called"
  [1]=>
  string(30) "and ten times nobody answered."
}

```

In version 2, only a comma forces a split regardless of preceding or trailing whitespace.

The third case splits the string only on a whitespace character and retains the starting position of the substring in the original string.

```

array(12) {
  [0]=>
  string(3) "Ten"
  [1]=>
  string(5) "times"
  [2]=>
  string(3) "she"
  [3]=>
  string(6) "called"
  [4]=>
  string(0) ""
  [5]=>
  string(0) ""
  [6]=>
  string(0) ""
  [7]=>
  string(3) "and"
  [8]=>
  string(3) "ten"
  [9]=>
  string(5) "times"
  [10]=>
  string(6) "nobody"
  [11]=>
  string(9) "answered."
}

```



Using the preg_split() function - Opera

Opera Using the preg_split() fu... x

Web localhost:8081/CNT4714/PHP/preg_split%20€ Search with G

```
array(4) {
  [0]=>
  array(2) {
    [0]=>
    string(10) "eXtensible"
    [1]=>
    int(0)
  }
  [1]=>
  array(2) {
    [0]=>
    string(9) "HyperText"
    [1]=>
    int(11)
  }
  [2]=>
  array(2) {
    [0]=>
    string(6) "Markup"
    [1]=>
    int(21)
  }
  [3]=>
  array(2) {
    [0]=>
    string(8) "Language"
    [1]=>
    int(28)
  }
}
```

The third case splits the string only on a whitespace character and retains the starting position of the substring in the original string.



Perl-Compatible Regular Expressions (PCRE)

- The last aspect of PCRE regex that we'll examine is that of assertions.
- In a regular expression, an assertion is a fact about the pattern that must be true. For example, we've already shown how you can use the `^` and `$` metacharacters to make an assertion about the position of the pattern in the string. Using the `^` requires that the pattern appear at the beginning of the string, while the `$` requires the pattern to appear at the end of the string.
- Another type of assertion in PCRE is that of a look-ahead assertion. A **look-ahead assertion** places a condition on the characters that follow the assertion. This allows you to specify an additional pattern for a regex.



Perl-Compatible Regular Expressions (PCRE)

- Look-ahead assertions are position dependent. This means that the pattern in the assertion must be matched starting at the current location in the string.
- To create a look-ahead assertion, you use an opening parenthesis followed by a question mark, an equal sign, the pattern for the assertion to test, and a closing parenthesis as illustrated below:

```
(?=assertion)
```

- The example on the following page illustrates a couple of look-ahead assertions.



```
7 <?php
8     $str = 'My office is HEC 236';
9     # The look-ahead assertion makes sure that 3 digits follow the building code (HEC).
10    if (preg_match_all('/(?:=\w*\sHEC\s\d{3})/', $str, $matches)){
11        print("Regular expression successful. Match on: $str <br />");
12    }
13    else {
14        print("Regular expression unsuccessful...no match.<br />n");
15    }
16    print("<br /><br />");
17    $str = 'My office is in HEC 2';
18    # The look-ahead assertion makes sure that 3 digits follow the building code (HEC).
19    if (preg_match('/(?:=\w*\sHEC\s\d{3})/', $str, $matches)){
20        print("Regular expression successful. Match on: $str <br />");
21    }
22    else {
23        print("Regular expression unsuccessful...no match.<br />");
24    }
25    print("<br /><br />");
26    $str = 'HEC 236 is my office';
27    # The look-ahead assertion makes sure that 3 digits follow the building code (HEC).
28    if (preg_match('/(?:=\w*\sHEC\s\d{3})/', $str, $matches)){
29        print("Regular expression successful. Match on: $str <br />");
30    }
31    else {
32        print("Regular expression unsuccessful...no match.<br />");
33    }
```



Perl-Compatible Regular Expressions (PCRE)

The screenshot shows a web browser window titled "Using look-ahead assertions - Opera". The address bar shows "localhost:8081/CNT4714/PHP/look-ahead%2f". The page content displays three lines of text, each with a callout box pointing to it:

- Line 1: "Regular expression successful. Match on: My office is HEC 236". Callout: "In version 1 look-ahead assertion succeeds in matching the 3 digits after HEC."
- Line 2: "Regular expression unsuccessful...no match.". Callout: "In version 2 look-ahead assertion fails to match 3 digits after HEC."
- Line 3: "Regular expression unsuccessful...no match.". Callout: "Version 3 illustrates position dependence of the look-ahead assertion. It fails because it is improperly positioned in the string."



Perl-Compatible Regular Expressions (PCRE)

- A negative look-ahead assertion is similar to a look-ahead assertion except that it checks to see that its pattern is *not* matched.
- The syntax for a negative look-ahead assertion is similar to that of a look-ahead assertion except that the equal sign is replaced by an exclamation mark:

```
(?!assertion)
```

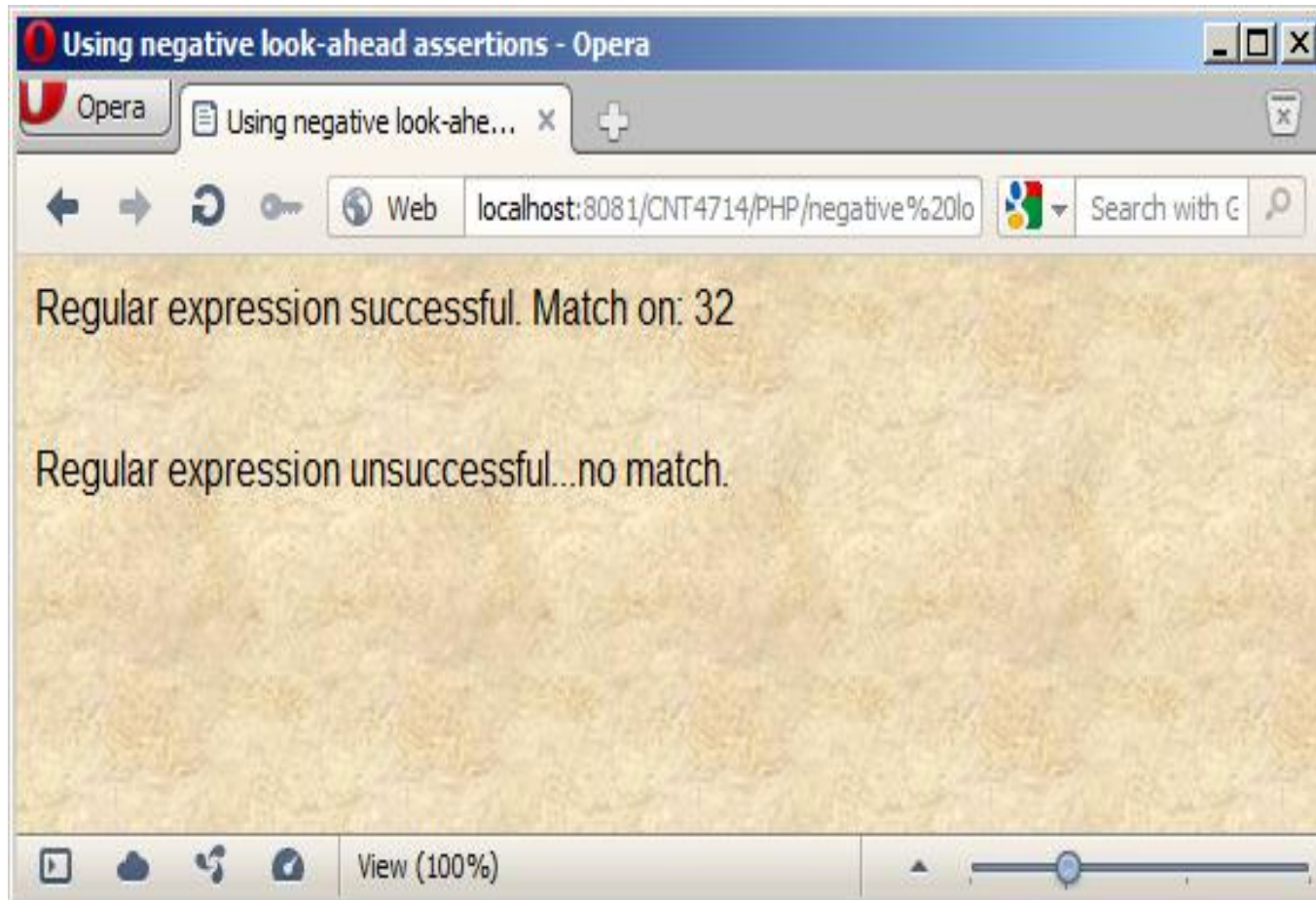
- The example on the following page illustrates the use of a negative look-ahead assertion.





```
1 <html>
2 <head>
3     <title> Using negative look-ahead assertions </title>
4 </head>
5 <body style = "font-family: arial, sans-serif;
6     background-color: #856363" background=image1.jpg>
7 <?php
8     $str = '32';
9     # The negative look-ahead assertion excludes the numbers 34 to 39 from matching.
10    if (preg_match('/^(?!3[4-9])[0-3]\d/', $str, $matches)){
11        print("Regular expression successful. Match on: $str <br />");
12    }
13    else {
14        print("Regular expression unsuccessful...no match.<br />n");
15    }
16    print("<br /><br />");
17    $str = '38';
18    # The negative look-ahead assertion excludes the numbers 34 to 39 from matching.
19    if (preg_match('/^(?!3[4-9])[0-3]\d/', $str, $matches)){
20        print("Regular expression successful. Match on: $str <br />");
21    }
22    else {
23        print("Regular expression unsuccessful...no match.<br />");
24    }
25    ?>
26 </body>
27 </html>
```





Practice Problems

- Here are a few regular expression practice problems. You might try to construct regular expressions for these in both POSIX and PCRE formats. I'll post the solutions in a day or so.
1. Credit card numbers in the format 9999-9999-9999-9999
 2. Zip codes in either 5 digit or 9 digit formats, e.g., 99999 or 99999-9999
 3. Phone numbers in the format (area code) prefix – number.
 4. Social security numbers.

