Apress™
Books for Professionals by Professionals

Chapter Eight: “Strings and Regular Expressions”

A Programmer’s Introduction to PHP 4.0

by William Jason Gilmore
ISBN # 1-893115-85-2

info@apress.com

CHAPTER 8

Strings and Regular Expressions

The ability to efficiently organize, search, and disseminate information has long been a topic of great interest for computer scientists. Because most of this information is text based as alphanumeric characters, a good deal of research has been invested in developing techniques to search and organize information based on an analysis of the patterns (known as *pattern matching*) in the text itself.

Pattern matching makes it possible not only to locate specific string instances but also to replace these instances with alternative strings. Common use of pattern matching is made in the find/replace functionality in word processors such as MS Word, Emacs, and my personal favorite, vi. UNIX users are undoubtedly familiar with programs such as sed, awk, and grep, all of which use pattern-matching techniques to provide the powerful functionality in each. Summarizing, pattern matching provides four useful functions:

- Locating strings exactly matching a specified pattern
- Searching strings for substrings matching a specified pattern
- Replacing strings and substrings matching a specified pattern
- Finding strings where the specified pattern does *not* match

The advent of the Web has caused a surge in research in faster, more efficient data-mining techniques, providing users worldwide with the capability to sift through the billions of pages of information. Search engines, online financial services, and ecommerce sites would all be rendered useless without the ability to analyze the mammoth quantities of data in these sectors. Indeed, string-manipulation capabilities are a vital part of almost any sector involving itself with information technology today.

This chapter concentrates on PHP's adept string-handling functionality. I will focus on a number of the more than 60 predefined string functions, providing definitions and practical examples that will give you the knowledge you need to begin coding powerful Web applications. However, before presenting the PHP-specific content of this chapter, I would like to provide a brief introduction

to the underlying mechanics that make pattern matching possible: regular expressions.

Regular Expressions

Regular expressions, or regexps, as they are so affectionately called by programmers, provide the foundation for pattern-matching functionality. A regular expression is nothing more than a sequence or pattern of characters itself, matched against the text in which a search has been requested. This sequence may be a pattern with which you are already familiar, such as the word "dog," or it may be a pattern having specific meaning in the context of the world of pattern-matching, such as `<(?)>.*<\/.?>`.

PHP offers functions specific to two sets of regular expression functions, each corresponding to a certain type of regular expression: POSIX and Perl style. Each has its own unique style of syntax and is discussed accordingly in later sections. Keep in mind that innumerable tutorials have been written regarding this matter; you can find them both on the Web and in various books. Therefore, I will provide you with a basic introduction to both and leave it to you to search out further information should you be so inclined.

If you are not already familiar with the mechanics of general expressions, please take some time to read through the short tutorial comprising the remainder of this section. If you are already a regexp pro, feel free to skip past the tutorial to subsequent sections.

Regular Expression Syntax (POSIX)

The structure of a POSIX regular expression is not dissimilar to that of a typical arithmetic expression: various elements (operators) are combined to form more complex expressions. However, it is the meaning of the combined regexp elements that makes them so powerful. It is possible not only to locate literal expressions, such as a specific word or number, but also to locate a multitude of semantically different but syntactically similar strings, for instance, all HTML tags in a file.

The simplest regular expression is one that matches a single character, such as *g*, matching strings such as *g*, *haggle*, and *bag*. You could combine several letters together to form larger expressions, such as *gan*, which logically would match any string containing *gan*; *gang*, *organize,* or *Reagan,* for example.

It is possible to simultaneously test for several different expressions by using the pipe (|) operator. For example, you could test for *php or zend* via the regular expression php|zend.

Bracketing

Brackets ([ ]) have a special meaning when used in the context of regular expressions, used to find a *range* of characters. Contrary to the regexp *php*, which will find strings containing the explicit string *php*, the regexp [*php*] will find any string containing the character *p* or *h*. Bracketing plays a significant role in regular expressions, since many times you may be interested in finding strings containing any of a range of characters. Several commonly used character ranges follow:

- [0–9] matches any decimal digit from 0 through 9.
- [a–z] matches any character from lowercase *a* through lowercase *z*.
- [A–Z] matches any character from uppercase *A* through uppercase *Z*.
- [a–Z] matches any character from lowercase *a* through uppercase *Z*.

Of course, the ranges shown above are general; you could also use the range [0–3] to match any decimal digit ranging from 0 through 3, or the range [b–v] to match any lowercase character ranging from *b* through *v*. In short, you are free to specify whatever range you wish.

Quantifiers

The frequency or position of bracketed character sequences and single characters can be denoted by a special character, each special character having a specific connotation. The +, *, ?, {int. range}, and $ flags all follow a character sequence:

- p+ matches any string containing at least one *p*.
- p* matches any string containing zero or more *p*'s.
- p? matches any string containing zero or more *p*'s. This is just an alternative way to use p*.
- p{2} matches any string containing a sequence of two *p*'s.
- p{2,3} matches any string containing a sequence of two or three *p*'s.
- p{2, } matches any string containing a sequence of at least two *p*'s.
- p$ matches any string with *p* at the end of it.

Still other flags can precede and be inserted before and within a character sequence:

- ^p matches any string with *p* at the beginning of it.
- [^a–zA-Z] matches any string *not* containing any of the characters ranging from *a* through *z* and *A* through *Z*.
- p.p matches any string containing *p*, followed by any character, in turn followed by another *p*.

You can also combine special characters to form more complex expressions. Consider the following examples:

- ^.{2}$ matches any string containing *exactly* two characters.
- (.*) matches any string enclosed within and (presumably HTML bold tags).
- p(hp)* matches any string containing a *p* followed by zero or more instances of the sequence *hp*.

You may wish to search for these special characters in strings instead of using them in the special context just described. For you to do so, the characters must be escaped with a backslash (\). For example, if you wanted to search for a dollar amount, a plausible regular expression would be as follows: ([^\$])([0-9]+), that is, a dollar sign followed by one or more integers. Notice the backslash preceding the dollar sign. Potential matches of this regular expression include $42, $560, and $3.

Predefined Character Ranges (Character Classes)

For your programming convenience several predefined character ranges, also known as *character classes*, are available. Character classes specify an entire range of characters, for example, the alphabet or an integer set:

[[:alpha:]] matches any string containing alphabetic characters *aA* through *zZ*.

[[:digit:]] matches any string containing numerical digits 0 through 9.

[[:alnum:]] matches any string containing alphanumeric characters *aA* through *zZ* and 0 through 9.

[[:space:]] matches any string containing a space.

PHP's Regexp Functions (POSIX Extended)

PHP currently offers seven functions for searching strings using POSIX-style regular expressions:

ereg()

ereg_replace()

eregi()

eregi_replace()

split()

spliti()

sql_regcase()

These functions are discussed in the following sections.

ereg()

The ereg() function searches a string specified by *string* for a string specified by *pattern*, returning true if the pattern is found, and false otherwise. Its syntax is:

```
int ereg(string pattern, string string, [array regs])
```

The search is case sensitive in regard to alphabetical characters. Here's how you could use ereg() to search strings for .com domains:

```
$is_com = ereg("(\.)(com$)", $email);
// returns true if $email ends with ".com".
// "www.wjgilmore.com" and "someemail@apress.com" would both return true values.
```

Note that since the $ concludes the regular expression, this will match only strings that end in com. For example, while this would match www.apress.com, it would *not* match www.apress.com/catalog.

The optional input parameter *regs* contains an array of all matched expressions that were grouped by parentheses in the regular expression. Making use of this array, we could segment a URL into several pieces, as shown in Listing 8-1.

Listing 8-1: Displaying elements of $regs array

```
<?
$url = "http://www.apress.com";

// break $url down into three distinct pieces: "http://www", "apress", and "com"
$www_url = ereg("^(http://www)\.([[:alnum:]]+)\.([[:alnum:]]+)", $url, $regs);

if ($www_url) :         // if $www_url is a valid URL
     echo $regs[0];     // outputs the entire string "http://www.apress.com"
     print "<br>";
     echo $regs[1];     // outputs "http://www"
     print "<br>";
     echo $regs[2];     // outputs "apress"
     print "<br>";
     echo $regs[3];     // outputs "com"
endif;
?>
```

Executing Listing 8-1 results in:

```
http://www.apress.com
http://www
apress
com
```

ereg_replace()

The `ereg_replace()` function searches for *string* specified by *pattern* and replaces *pattern* with *replacement* if found. The syntax is:

```
string ereg_replace (string pattern, string replacement, string string)
```

The `ereg_replace()` function operates under the same premises as `ereg()`, except that the functionality is extended to finding and replacing *pattern* instead of simply locating it. After the replacement has occurred, the modified string will be returned. If no matches are found, the string will remain unchanged. Like `ereg()`, `ereg_replace()` is case sensitive. Here is a simple string replacement example that uses the function:

```
$copy_date = "Copyright 1999";
$copy_date = ereg_replace("([0-9]+)", "2000", $copy_date);
print $copy_date;     // displays "Copyright 2000"
```

A rather interesting feature of PHP's string-replacement capability is the ability to back-reference parenthesized substrings. This works much like the optional input parameter *regs* in the function `ereg()`, except that the substrings are referenced using backslashes, such as \0, \1, \2, and so on, where \0 refers to the entire string, \1 the first successful match, and so on. Up to nine back references can be used. This example shows how to replace all references to a URL with a working hyperlink:

```
$url = "Apress (http://www.apress.com)";
$url = ereg_replace("http://(([A-Za-z0-9.\-])*)", "<a href=\"\\0\">\\0</a>",$url);
print $url;
// Displays Apress (<a href="http://www.apress.com">http://www.apress.com</a>)
```

NOTE *Although* `ereg_replace()` *works just fine, another predefined function named* `str_replace()` *is actually much faster when complex regular expressions are not required.* `Str_replace()` *is discussed later in this chapter.*

eregi()

The `eregi()` function searches throughout a string specified by *pattern* for a string specified by *string*. Its syntax is:

```
int eregi(string pattern, string string, [array regs])
```

The search is *not* case sensitive. `Eregi()` can be particularly useful when checking the validity of strings, such as passwords. This concept is illustrated in the following sample:

```
$password = "abc";

if (! eregi ("[[:alnum:]]{8,10}", $password)) :
     print "Invalid password! Passwords must be from 8 through 10 characters in
length.";
endif;

// execution of the above code would produce the error message
// since "abc" is not of length ranging from 8 through 10 characters.
```

eregi_replace()

The `eregi_replace()` function operates exactly like `ereg_replace()`, except that the search for *pattern* in *string* is not case sensitive. Its syntax is:

```
string eregi_replace (string pattern, string replacement, string string)
```

split()

The `split()` function will divide a string into various elements, the boundaries of each element based on the occurrence of *pattern* in *string*. Its syntax is:

```
array split (string pattern, string string [, int limit])
```

The optional input parameter *limit* is used to signify the number of elements into which the string should be divided, starting from the left end of the string and working rightward. In cases where the pattern is an alphabetical character, `split()` is case sensitive. Here's how you would use `split()` to partition an IP address:

```
$ip = "123.456.789.000";      // some IP address
$iparr = split ("\.", $ip);    // Note that since "." is a special character, it
                                  must be escaped.
print "$iparr[0] <br>";       // outputs "123"
print "$iparr[1] <br>";       // outputs "456"
print "$iparr[2] <br>";       // outputs "789"
print "$iparr[3] <br>";       // outputs "000"
```

You could also use `split()` to limit a parameter to restrict division of `$ip`:

```
$ip = "123.456.789.000";          // some IP address
$iparr = split ("\.", $ip, 2);    // Note that since "." is a special character,
                                     it must be escaped.
print "$iparr[0] <br>";           // outputs "123"
print "$iparr[1] <br>";           // outputs "456.789.000"
```

spliti()

The `spliti()` function operates exactly in the same manner as its sibling `split()`, except that it is *not* case sensitive. Its syntax is:

```
array split (string pattern, string string [, int limit])
```

Of course, case-sensitive characters are an issue only when the pattern is alphabetical. For all other characters, `spliti()` operates exactly as `split()` does.

sql_regcase()

The `sql_regcase()` function can be thought of as a utility function, converting each character in the input parameter *string* into a bracketed expression containing two characters. Its syntax is:

```
string sql_regcase (string string)
```

If the alphabetical character has both an uppercase and a lowercase format, the bracket will contain both forms; otherwise the original character will be repeated twice. This function is particularly useful when PHP is used in conjunction with products that support solely case-sensitive regular expressions. Here's how you would use `sql_regcase()` to convert a string:

```
$version = "php 4.0";

print sql_regcase($version);
// outputs [Pp] [Hh] [Pp] [ ] [44] [..] [00]
```

Regular Expression Syntax (Perl Style)

Perl (http://www.perl.com), long considered one of the greatest parsing languages ever written, provides a comprehensive regular expression language that can be used to search and replace even the most complicated of string patterns. The developers of PHP felt that instead of reinventing the regular expression wheel, so to speak, they should make the famed Perl regular expression syntax available to PHP users, thus the Perl-style functions.

Perl-style regular expressions are similar to their POSIX counterparts. In fact, Perl's regular expression syntax is a distant derivation of the POSIX implementation, resulting in the fact that the POSIX syntax can be used almost interchangeably with the Perl-style regular expression functions.

I devote the remainder of this section to a brief introduction of Perl regexp syntax. This is a simple example of a Perl regexp:

```
/food/
```

Notice that the string 'food' is enclosed between two forward slashes. Just like with POSIX regexps, you can build a more complex string through the use of quantifiers:

```
/fo+/
```

This will match 'fo' followed by one or more characters. Some potential matches include 'food', 'fool', and 'fo4'. Here is another example of using a quantifier:

```
/fo{2,4}/
```

This matches ''f'' followed by two to four occurrences of 'o.' Some potential matches include 'fool', 'fooool', and 'foosball'.

In fact, you can use any of the quantifiers introduced in the previous POSIX section.

Metacharacters

Another cool thing you can do with Perl regexps is use various metacharacters to search for matches. A *metacharacter* is simply an alphabetical character preceded by a backslash that acts to give the combination a special meaning. For instance, you can search for large money sums using the '\d' metacharacter:

```
/([\d]+)000/
```

'\d' will search for any string of numerical character. Of course, searching for alphabetical characters is important, thus the '\w' metacharacter:

```
/<([\w]+)>/
```

This will match things like HTML tags. (By contrast, the '\W' metacharacter searches for nonalphabetical characters.)

Another useful metacharacter is '\b', which searches for word boundaries:

```
/sa\b/
```

Because the word boundary is designated to be on the right-side of the strings, this will match strings like 'pisa' and 'lisa' but not 'sand'. The opposite of the word boundary metacharacter is '\B'. This matches on anything *but* a word boundary:

```
/sa\B/
```

This will match strings like 'sand' and 'Sally' but not 'Alessia'.

Modifiers

Several modifiers are available that can make your work with regexps much easier. There are many of these; however, I will introduce just a few of the more interesting ones in Table 8-1. These modifiers are placed directly after the regexp, for example, /string/i.

Table 8-1. Three Sample Modifiers

MODIFIER	DESCRIPTION
m	Treats a string as several ('m' for multiple) lines. By default, the '^' and '$' special characters match at the very start and very end of the string in question. Using the 'm' modifier will allow for '^' and '$' to match at the beginning of *any line* in a string.
s	Accomplishes just the opposite of the 'm' modifier, treating a string as a single line, ignoring any newline characters found within.
i	Implies a case-insensitive search.

This introduction has been brief, as attempting to document regular expressions in their entirety is surely out of the scope of this book and could easily fill many chapters rather than just a few pages. For more information regarding regular expression syntax, check out these great online resources:

- http://www.php.net/manual/pcre.pattern.modifiers.php
- http://www.php.net/manual/pcre.pattern.syntax.php
- http://www.perl.com/pub/doc/manual/html/pod/perlre.html
- http://www.codebits.com/p5be/
- http://www.metronet.com/1/perlinfo/doc/FMTEYEWTK/regexps.html

PHP's Regexp Functions (Perl Compatible)

PHP offers five functions for searching strings using Perl-compatible regular expressions:

- `preg_match()`
- preg_match_all()

- preg_replace()
- preg_split()
- preg_grep()

These functions are discussed in the following sections.

preg_match()

The preg_match() function searches *string* for *pattern*, returning true if *pattern* exists, and false otherwise. Its syntax follows:

```
int preg_match (string pattern, string string [, array pattern_array])
```

If the optional input parameter *pattern_array* is provided, then *pattern_array* will contain various sections of the subpatterns contained in the search pattern, if applicable. Here's an example that uses preg_match() to perform a case-sensitive search:

```
$line = "Vi is the greatest word processor ever created!";
// perform a case-Insensitive search for the word "Vi"
if (preg_match("/\bVi\b/i", $line, $match)) :
     print "Match found!";
endif;
// The if statement will evaluate to true in this example.
```

preg_match_all()

The preg_match_all() function matches all occurrences of *pattern* in *string*. Its syntax is:

```
int preg_match_all (string pattern, string string, array pattern_array [, int
order])
```

It will place these matches in the array *pattern_array* in the order you specify using the optional input parameter *order*. There are two possible types of *order*:

- PREG_PATTERN_ORDER is the default if the optional order parameter is not included. PREG_PATTERN_ORDER specifies the order in the way that you might think most logical; $pattern_array[0] is an array of all complete pattern matches, $pattern_array[1] is an array of all strings matching the first parenthesized regexp, and so on.

- PREG_SET_ORDER will order the array a bit differently than the default setting. `$pattern_array[0]` will contain elements matched by the first parenthesized regexp, `$pattern_array[1]` will contain elements matched by the second parenthesized regexp, and so on.

Here's how you would use `preg_match_all` to find all strings enclosed in bold HTML tags:

```
$userinfo = "Name: <b>Rasmus Lerdorf</b> <br> Title: <b>PHP Guru</b>";
preg_match_all ("/<b>(.*)<\/b>/U", $userinfo, $pat_array);
print $pat_array[0][0]." <br> ".$pat_array[0][1]."\n";
```

```
Rasmus Lerdorf
PHP Guru
```

preg_replace()

The `preg_replace()` function operates just like `ereg_replace()`, except that regular expressions can be used in the *pattern* and *replacement* input parameters. Its syntax is:

```
mixed preg_replace (mixed pattern, mixed replacement, mixed string [, int limit])
```

The optional input parameter *limit* specifies how many matches should take place. Interestingly, the *pattern* and *replacement* input parameters can be arrays. `Preg_replace()` will cycle through each element of each array, making replacements as they are found.

preg_split()

The `preg_split()` function operates exactly like `split()`, except that regular expressions are accepted as input parameters for *pattern*. Its syntax is:

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

If the optional input parameter *limit* is specified, then only *limit* number of substrings are returned. This example uses `preg_split()` to parse a variable.

```
$user_info = "+WJ+++Gilmore+++++wjgilmore@hotmail.com++++++++Columbus+++OH";
$fields = preg_split("/\+{1,}/", $user_info);
while ($x < sizeof($fields)) :
   print $fields[$x]. "<br>";
   $x++;
endwhile;
```

```
WJ
Gilmore
wjgilmore@hotmail.com
Columbus
OH
```

preg_grep()

The `preg_grep()` function searches all elements of *input_array*, returning all elements matching the regexp *pattern*. Its syntax is:

```
array preg_grep (string pattern, array input_array)
```

Here's how you would use `preg_grep()` to search an array for foods beginning with *p*:

```
$foods = array("pasta", "steak", "fish", potatoes");
// find elements beginning with "p", followed by one or more letters.
$p_foods = preg_grep("/p(\w+)/", $foods);

$x = 0;

while ($x < sizeof($p_foods)) :
   print $p_foods[$x]. "<br>";
   $x++;
endwhile;
```

```
pasta
potatoes
```

Other String-Specific Functions

In addition to the regular expression–based functions discussed in the first half of this chapter, PHP provides 70+ functions geared toward manipulating practically every aspect of a string that you can think of. To list and explain each function would be out of the scope of this book and would not accomplish much more than repeat much of the information in the PHP documentation. Therefore, I have devoted the remainder of this chapter to a FAQ of sorts, the questions being those that seem to be the most widely posed in the many PHP discussion groups and related sites. Hopefully, this will be a much more efficient means for covering the generalities of the immense PHP string-handling library.

Padding and Compacting a String

For formatting reasons, it is necessary to modify the string length via either padding or stripping characters. PHP provides a number of functions for doing so.

chop()

The `chop()` function returns a string minus any ending whitespace and newlines. Its syntax is:

```
string chop (string str)
```

This example uses `chop()` to remove unnecessary newlines:

```
$header = "Table of Contents:\n\n";
$header = chop($header);
// $header = "Table of Contents"
```

str_pad()

The `str_pad()` function will pad *string* to length *pad_length* with a specified set of characters, returning the newly formatted string. Its syntax is:

```
string str_pad (string input, int pad_length [, string pad_string [, int
pad_type]])
```

If the optional parameter *pad_string* is not specified, *string* will be padded with blank spaces; otherwise it will be padded with the character pattern specified in

pad_string. By default, the *string* will be padded to the right; however, the optional *pad_type* may be assigned STR_PAD_RIGHT, STR_PAD_LEFT, or STR_PAD_BOTH, padding the string accordingly. This example shows how to pad a string using `str_pad()` defaults:

```
$food = "salad";
print str_pad ($food, 5);     // prints "salad     "
```

This sample makes use of `str_pad()`'s optional parameters:

```
$header = "Table of Contents";
print str_pad ($header, 5, "=+=+=", STR_PAD_BOTH);
// "=+=+=Table of Contents=+=+=" will be displayed to the browser.
```

trim()

The `trim()` function will remove all whitespace from both the left and right sides of *string*, returning the resulting string. Its syntax is:

```
string trim (string string)
```

It will also remove the special characters "\n", "\r", "\t", "\v" and "\0".

ltrim()

The `ltrim()` function will remove the whitespace and special characters from the left side of *string*, returning the remaining string. Its syntax follows:

```
string ltrim (string str)
```

The special characters that will be removed are the same as those removed by `trim()`.

Finding Out the Length of a String

You can determine the length of a string through use of the `strlen()` function. This function returns the length of a string, each character in the string being equivalent to one unit. Its syntax is:

```
int strlen (string str)
```

This example uses `strlen()` to determine the length of a string:

```
$string = "hello";
$length = strlen($string);
// $length = 5
```

Comparing Two Strings

String comparison is arguably one of the most important features of the string-handling capabilities of any language. Although there are many ways in which two strings can be compared for equality, PHP provides four functions for performing this task:

- `strcmp()`
- strcasecmp()
- `strspn()`
- `strcspn()`

These functions are discussed in the following sections.

strcmp()

The `strcmp()` function performs a case-sensitive comparison of two strings. Its syntax follows:

```
int strcmp (string string1, string string2)
```

On completion of the comparison, `strcmp()` will return one of three possible values:

- 0 if *string1* and *string2* are equal
- < 0 if *string1* is less than *string2*
- > 0 if *string2* is less than *string1*

This listing compares two equivalent string values:

```
$string1 = "butter";
$string2 = "butter";

if ((strcmp($string1, $string2)) == 0) :
     print "Strings are equivalent!";
endif;
// If statement will evaluate to true
```

strcasecmp()

The `strcasecmp()` function operates exactly like `strcmp()`, except that its comparison is case *insensitive*. Its syntax is:

```
int strcasecmp (string string1, string string2)
```

The following example compares two equivalent string values:

```
$string1 = "butter";
$string2 = "Butter";

if ((strcasecmp($string1, $string2)) == 0) :
     print "Strings are equivalent!";
endif;
// If statement will evaluate to true
```

strspn()

The `strspn()` function returns the length of the first segment in *string1* containing characters also in *string2*. Its syntax is:

```
int strspn (string string1, string string2)
```

Here's how you would use `strspn()` to validate a password:

```
$password = "12345";
if (strspn($password, "1234567890") != strlen($password)) :
     print "Password cannot consist solely of numbers!";
endif;
```

strcspn()

The strcspn() function returns the length of the first segment in *string1* containing characters *not* in *string2*. Its syntax is:

```
int strcspn (string str1, string str2)
```

Here's an example of password validation using strcspn():

```
$password = "12345";
if (strcspn($password, "1234567890") == 0) :
     print "Password cannot consist solely of numbers!";
endif;
```

Alternatives for Regular Expression Functions

When processing large amounts of information, the regular expression functions can slow matters dramatically. You should use these functions only when you are interested in parsing relatively complicated strings that require the use of regular expressions. If you are instead interested in parsing for simple expressions, there are a variety of predefined functions that will speed up the process considerably. Each of these functions is described below.

strtok()

The strtok() function will tokenize *string*, using the characters specified in *tokens*. Its syntax is:

```
string strtok (string string, string tokens)
```

One oddity about strtok() is that it must be continually called in order to completely tokenize a string; Each call to strtok() only tokenizes the next piece of the string. However, the *string* parameter only needs to be specified once, as the function will keep track of its position in *string* until it either completely tokenizes *string* or a new *string* parameter is specified. This example tokenizes a string with several delimiters:

```
$info = "WJ Gilmore:wjgilmore@hotmail.com|Columbus, Ohio";

// delimiters include colon (:), vertical bar (|), and comma (,)
$tokens = ":|,";

$tokenized = strtok($info, $tokens);
// print out each element in the $tokenized array
while ($tokenized) :
     echo "Element = $tokenized<br>";
     // Note how strtok does not take the first argument on subsequent executions
     $tokenized = strtok ($tokens);
endwhile;
```

```
Element =WJ Gilmore
Element = wjgilmore@hotmail.com
Element = Columbus
Element = Ohio
```

parse_str()

The `parse_str()` function parses *string* into various variables, setting the variables in the current scope. The syntax is:

```
void parse_str (string string)
```

This function is particularly useful when handling URLs that contain HTML form or otherwise extended information. The following example parses information passed via a URL. This string is the common form for a grouping of data that is passed from one page to another, compiled either directly in a hyperlink or in an HTML form:

```
$url = "fname=wj&lname=gilmore&zip=43210";
parse_str($url);
// after execution of parse_str(), the following variables are available:
// $fname = "wj"
// $lname = "gilmore"
// $zip = "43210"
```

Because this function was created to work with URLs, it ignores the ampersand (&) symbol.

NOTE *The subject of PHP and HTML forms is introduced in Chapter 10, "Forms."*

explode()

The explode() function will divide *string* into various elements, returning these elements in an array. The syntax is:

```
array explode (string separator, string string [, int limit])
```

The division takes place at each occurrence of *separator*, and the number of divisions can be regulated with the optional inclusion of the input parameter *limit*. This example divides a string using the explode() function:

```
$info = "wilson|baseball|indians";
$user = explode("|", $info);
// $user[0] = "wilson";
// $user[1] = "baseball";
// $user[2] = "indians";
```

NOTE *The* explode() *function is virtually identical to the POSIX regular expression function* split()*, described earlier in this chapter. The main difference is that split() should only be used when you need to employ regular expressions in the input parameters.*

implode()

Just as you can use the explode() function to project a string into various elements of an array, you can implode an array to form a string. This is accomplished with the implode() function. Its syntax is:

```
string implode (string delimiter, array pieces)
```

This example forms a string out of the elements of an array:

```
$ohio_cities = array("Columbus", "Youngstown", "Cleveland", "Cincinnati");
$city_string = implode("|", $ohio_cities);
// $city_string = "Columbus|Youngstown|Cleveland|Cincinnati";
```

NOTE `Join()` *is an alias for* `implode`.

strpos()

The `strpos()` function finds the position of the first *occurrence* in *string*. Its syntax is:

```
int strpos (string string, string occurrence [, int offset])
```

The optional input parameter *offset* specifies the position at which to begin the search. If *occurrence* is not in *string*, `strpos()` will return false (0).

The following example determines the location of the first date entry in an abbreviated log:

```
$log = "
206.169.23.11:/www/:2000-08-10
206.169.23.11:/www/logs/:2000-02-04
206.169.23.11:/www/img/:1999-01-31";
// what is first occurrence of year 1999 in log?
$pos = strpos($log, "1999");
// $pos = 95, because first occurrence of "1999" is
// at position 95 of the string contained in $log,
```

strrpos()

The `strrpos()` function locates the last occurrence of *character* in *string*. Its syntax is:

```
int strrpos (string string, char character)
```

This function is less powerful than its counterpart, `strpos()`, because the search can only be performed on one character rather than a string. If a string is passed as the second input parameter into `strrpos()`, only the first character of that string will be used in the search.

str_replace()

The `str_replace()` function searches for *occurrence* in *string*, replacing all instances with *replacement*. Its syntax is:

```
string str_replace (string occurrence, string replacement, string string)
```

If *occurrence* is not in *string*, the string is not modified.

> **TIP** `substr_replace()`, *described later in this section, allows you to replace just a portion of a string. This example shows how* `str_replace()` *can replace several instances of an element in a string:*
>
> ```
> $favorite_food = "My favorite foods are ice cream and chicken wings";
> $favorite_food = str_replace("chicken wings", "pizza", $favorite_food);
> // $favorite_food = "My favorite foods are ice cream and pizza"
> ```

strstr()

The `strstr()` function returns the remainder of *string* beginning at the first *occurrence*. Its syntax is:

```
string strstr (string string, string occurrence)
```

This example uses `strstr()` to return the domain name of a URL:

```
$url = "http://www.apress.com";
$domain = strstr($url, ".");
// $domain = ".apress.com"
```

substr()

The `substr()` function returns the part of the string between the *start* and *start*+*length* parameters. Its syntax is:

```
string substr (string string, int start [, int length])
```

If the optional *length* parameter is not specified, the substring is considered to be the string starting at *start* and ending at the end of *string*. There are four points to keep in mind when using this function:

- If *start* is positive, the returned substring will begin at the *start*'th position of the string.
- If *start* is negative, the returned substring will begin at the string (*length* – *start*'th position of the string.

- If *length* is provided and is positive, the returned substring will consist of the characters between *start* and (*start* + *length*). If this distance is greater than the distance between *start* and the end of *string*, then only the substring between *start* and the string's end will be returned.

- If *length* is provided and is negative, the returned substring will end *length* characters from the end of *string*.

TIP *Keep in mind that start is the offset from the first character of the string; therefore the returned string will actually start at character position (start + 1).*

This sample returns a portion of a string using `substr()`:

```
$car = "1944 Ford";
$model = substr($car, 6);
// $model = "Ford"
```

The following code is an example of a positive `substr()` *length* parameter:

```
$car = "1944 Ford";
$yr = substr($car, 0, 4);
// $yr = "1944"
```

Here is an example of a negative `substr()` *length* parameter:

```
$car = "1944 Ford";
$yr = substr($car, 2, -5);
// $yr = "44"
```

substr_count()

The `substr_count()` function returns the number of times *substring* occurs in *string*. Its syntax is:

```
int substr_count (string string, string substring)
```

This example counts the frequency of occurrence of a substring in a string:

```
$tng_twist = "The rain falls mainly on the plains of Spain";
$count = substr_count($tng_twist, "ain");
// $count = 4
```

substr_replace()

The `substr_replace()` function will replace a portion of *string* with *replacement*, beginning the replacement at *start* position of the string, and ending at *start* + *length* (assuming that the optional input parameter *length* is included). Its syntax is:

```
string substr_replace (string string, string replacement, int start [, int
length])
```

Alternatively, the replacement will stop on the complete placement of *replacement* in *string*. There are several subtleties regarding the values of *start* and *length*:

- If *start* is positive, *replacement* will begin at character *start*.
- If *start* is negative, *replacement* will begin at (string length – *start*).
- If *length* is provided and is positive, *replacement* will be *length* characters long.
- If *length* is provided and is negative, *replacement* will end at (string length – *length*) characters.

This example shows a simple replacement of the remainder of a string using `substr_replace()`:

```
$favs = "'s favorite links";
$name = "Alessia";
// The "0, 0" means that the replacement should begin at
// string's first position, and end at the original first position.
$favs = substr_replace($favs, $name, 0, 0);
print $favs;
```

Resulting in:

```
Alessia's favorite links
```

Converting Strings and Files to HTML and Vice Versa

Converting a string or an entire file into one suitable for viewing on the Web (and vice versa) is easier than you would think. Several functions are suited for this task.

Plain Text to HTML

It is often useful to be able to quickly convert plain text into a format that is readable in a Web browser. Several functions can aid you in doing so. These functions are the subject of this section.

nl2br()

The `nl2br()` function will convert all newline (\n) characters in a string to their HTML equivalent, that is,
. Its syntax is:

```
string nl2br (string string)
```

The newline characters could be invisible, created via hard returns, or visible, explicitly written in the string. The following example translates a text string (having newline characters '\n' to break lines) to HTML format:

```
// text string as it may be seen in a word processor.
$text_recipe = "
Party Sauce recipe:
1 can stewed tomatoes
3 tablespoons fresh lemon juice
Stir together, Serve cold.";
// convert the newlines to <br>'s.
$html_recipe = nl2br ($text_recipe);
```

Subsequently printing `$html_recipe` to the browser would result in the following HTML content being output:

```
Party Sauce recipe:<br>
1 can stewed tomatoes<br>
3 tablespoons fresh lemon juice<br>
Stir together, Serve cold.<br>
```

htmlentities()

The `htmlentities()` function will convert all characters into their equivalent HTML entities. The syntax is:

```
string htmlentities (string string)
```

The following example converts necessary characters for Web display:

```
$user_input = "The cookbook, entitled 'Cafè Française' costs < $42.25.";
$converted_input =  htmlentities($user_input);
// $converted_input = "The cookbook, entitled 'Caf&egrave;
// Fran&ccedil;aise' costs &lt; 42.24.";
```

> **NOTE** *The* `htmlentities()` *function currently only works in conjunction with the ISO-8859-1 (ISO-Latin-1) character set. Also,* `htmlentities()` *does not convert spaces to   as you may expect.*

htmlspecialchars()

The `htmlspecialchars()` function converts a select few characters having special meaning in the context of HTML into their equivalent HTML entities. Its syntax is:

```
string htmlspecialchars (string string)
```

The `htmlspecialchars()` function currently only converts the following characters:

- & becomes &amp
- “” becomes &quot
- < becomes <
- > becomes &gt

This function is particularly useful in preventing users from entering HTML markup into an interactive Web application, such as a message board. Improperly coded HTML markup can cause an entire page to be formed incorrectly. However, perhaps a more efficient way to do this is to use `strip_tags()`, which deletes the tags from the string altogether.

The following example converts potentially harmful characters using `htmlspecialchars()`:

```
$user_input = "I just can't get <<enough>> of PHP & those fabulous cooking
recipes!";
$conv_input = htmlspecialchars($user_input);
// $conv_input = "I just can't get &lt;&lt;enough&gt;&gt; of PHP &amp those
fabulous cooking recipes!"
```

> **TIP** *If you are using* `gethtmlspecialchars()` *in conjunction with* `nl2br()`*, you should execute* `nl2br()` *after* `gethtmlspecialchars()`*; otherwise the
's generated with* `nl2br()` *will be converted to visible characters.*

get_html_translation_table()

Using `get_html_translation_table()` is a convenient way to translate text to its HTML equivalent. Its syntax is:

```
string get_html_translation_table (int table)
```

Basically, `get_html_translation_table()` returns one of the two translation tables (specified by the input parameter *table*) used for the predefined `htmlspecialchars()` and `htmlentities()` functions. This returned value can then be used in conjunction with another predefined function, `strtr()` (defined later in this chapter), to essentially translate the text into HTML code.

The two tables that can be specified as input parameters to this function are:

- HTML_ENTITIES
- HTML_SPECIALCHARS

The following sample uses `get_html_translation_table()` to convert text to HTML:

```
$string = "La pasta é il piatto piú amato in Italia";
$translate = get_html_translation_table(HTML _ENTITIES);
print strtr($string, $translate);
// the special characters are converted to HTML entities and properly
// displayed in the browser.
```

Interestingly, `array_flip()` is capable of reversing the text-to-HTML translation and vice versa. Assume that instead of printing the result of `strtr()` in the preceding code sample, we assigned it to the variable $translated_string.

The next example uses `array_flip()` to return a string back to its original value:

```
$translate = array_flip($translate);
$translated_string = "La pasta &eacute; il piatto pi&uacute; amato in Italia"
$original_string = array_flip($translated_string, $translate);
// $original_string = "La pasta é il piatto piú amato in Italia";
```

strtr()

The `strtr()` function will convert all characters contained in *destination* to their corresponding character matches in *source*. Its syntax:

```
string strtr (string string, string source, string destination)
```

If the *source* and *destination* strings are of different length, any characters in the longer of the two will be truncated.

Essentially, you can think of `strtr()` as inversely comparing the values of two sets of arrays and making the replacements from *source* to *destination* as necessary. This example converts HTML characters to XML-like format:

```
$source = array("<title>" => "<h1>", "</title>" => "</h1>");
$string = "<h1>Today In PHP-Powered News</h1>";
print strtr($string, $source);
// prints "<title>Today In PHP-Powered News</title>"
```

HTML to Plain Text

You may sometimes need to convert an HTML file to plain text. The following functions can help you to do so.

strip_tags()

The `strip_tags()` function will remove all HTML and PHP tags from *string*, leaving only the text entities. Its syntax is:

```
string strip_tags (string string [, string allowable_tags])
```

The optional *allowable_tags* parameter allows you to specify which tags you would like to be skipped during this process.

This example uses `strip_tags()` to delete all HTML tags from a string:

```
$user_input = "i just <b>love</b> PHP and <i>gourmet</i> recipes!";
$stripped_input = strip_tags($user_input);
// $stripped_input = "I just love PHP and gourmet recipes!";
```

The following sample strips all except a few tags:

```
$input = "I <b>love</b> to <a href = \"http://www.eating.com\">eat<a>!";
$strip_input = strip_tags($user_input, "<a>");
// $strip_input = "I love to <a href = \"http://www.eating.com\">eat</a>!";
```

> **NOTE** *Another function that performs similarly to* `strip_tags()` *is* `fgetss()`. *This function is described in Chapter 7, "File I/O and the File System."*

get_meta_tags()

Although perhaps not relating directly to the question of conversion, `get_meta_tags()` can be such a useful function that I did not want to leave it out. Its syntax is:

```
array get_meta_tags (string filename/URL [, int use_include_path])
```

The `get_meta_tags()` function will search an HTML file for what are known as META tags.

META tags are special descriptive tags that provide information, primarily to search engines, about a particular page. These tags are contained in the <head>…</head> HTML tags of a page. An example set of some available META tags might look like the following (we'll call this example.html, as it will be used in Listing 8-1):

```
<html>
<head>
<title>PHP Recipes</title>
<META NAME="keywords" CONTENT="gourmet, PHP, food, code, recipes, chef,
programming, Web">
```

```
<META NAME="description" CONTENT="PHP Recipes provides savvy readers with the
latest in PHP programming and gourmet cuisine!">
<META NAME="author" CONTENT="WJ Gilmore">
</head>
```

`Get_meta_tags()` will search for tags beginning with the word META in the head of a document and will place all tag names and their content in an associative array. Considering the previous META tag example, take a look at Listing 8-1.

Listing 8-1: Using get_meta_tags() to parse META tags in an HTML file

```
$meta_tags = get_meta_tags("example.html");
// $meta_tags will return an array containing the following information:
// $meta_tags["keywords"] = "gourmet, PHP, food, code, recipes, chef, programming,
Web";
// $meta_tags["description"] = "PHP-Powered Recipes provides savvy readers with
the latest in PHP
// programming and gourmet cuisine!";
// $meta_tags["author"] = "WJ Gilmore";
```

Interestingly, it is possible to extract META tags not only from a file residing on the server from which the script resides but also from other URLs.

> **TIP** *For a great tutorial describing what META tags are and how to use them, I suggest checking out Joe Burn's tutorial, "So, You Want a Meta Command, Huh?" on the HTML Goodies site at http://htmlgoodies.earthweb.com/tutors/meta.html.*

Converting a String into Uppercase and Lowercase Letters

Four functions are available to aid you in accomplishing this task:

- `strtolower()`
- strtoupper()
- `ucfirst()`
- `ucwords()`

These functions are discussed in the following sections.

strtolower()

The `strtolower()` function does exactly what you would expect it to: it converts a string to all lowercase letters. Its syntax is:

```
string strtolower (string string)
```

Nonalphabetical characters are not affected. The following example uses `strtolower()` to convert a string to all lowercase letters:

```
$sentence = "COOKING and PROGRAMMING PHP are my TWO favorite pastimes!";
$sentence = strtolower($sentence);
// $sentence is now
// "cooking and programming php are my two favorite pastimes!"
```

strtoupper()

Just as you can convert a string to lowercase, so can you convert one to uppercase. This is accomplished with the function `strtoupper()`, and its syntax is:

```
string strtoupper (string string)
```

Nonalphabetical characters are not affected. This example uses `strtoupper()` to convert a string to all uppercase letters:

```
$sentence = "cooking and programming PHP are my two favorite pastimes!";
$sentence = strtoupper($sentence);
// $sentence is now
// "COOKING AND PROGRAMMING PHP ARE MY TWO FAVORITE PASTIMES!"
```

ucfirst()

The `ucfirst()` function capitalizes the first letter of a string, provided that it is alphabetical. Its syntax is:

```
string ucfirst (string string)
```

Nonalphabetical characters will not be affected. The following example uses `ucfirst()` to capitalize the first letter of a string:

```
$sentence = "cooking and programming PHP are my two favorite pastimes!";
$sentence = ucfirst($sentence);
// $sentence is now
// "Cooking and programming PHP are my two favorite pastimes!"
```

ucwords()

The `ucwords()` function capitalizes the first letter of each word in a string. Its syntax is:

```
string ucwords (string string)
```

Nonalphabetical characters are not affected. A *word* is defined as a string of characters separated from other entities in the string by a blank space on each side. This example uses `ucwords()` to capitalize each word in a string:

```
$sentence = "cooking and programming PHP are my two favorite pastimes!";
$sentence = ucwords($sentence);
// $sentence is now
// "Cooking And Programming PHP Are My Two Favorite Pastimes!"
```

Project: Browser Detection

Anyone who attempts to develop a user-friendly Web site must take into account the differences of page formatting when the site is viewed using the various Web browsers and operating systems. Even though the W3 (http://www.w3.org) organization continues to offer standards that Internet application developers should adhere to, the various browser developers just love to add their own little "improvements" to these standards, essentially causing havoc and confusion for content developers worldwide. Developers have largely resolved this problem by actually creating different pages for each type of browser and operating system, a process that at times can be painful but results in sites that conform perfectly for any user, building the reputability and confidence that user has in returning to that site.

For users to view the page format that is intended for their browser and operating system, the incoming page request is "sniffed" for browser and platform information. Once the necessary information has been determined, users are then redirected to the correct page.

The purpose of this project is to show you how PHP's regular expression functionality can be used to build a "browser sniffer," sniffer.php. This sniffer will determine the operating system and browser type and version, displaying the information to the browser window. However, before delving into the code, I would

like to take a moment to review one of the primary pieces of the sniffer, the predefined PHP variable $HTTP_USER_AGENT. This variable basically stores various information in string format about the requesting user's browser and operating system, which is exactly what we are looking for. We could easily display this information to the screen with just one line of code:

```
<?
echo $HTTP_USER_AGENT;
?>
```

If you are using Internet Explorer 5.0 on a Windows 98 machine, you will see the following output:

```
Mozilla/4.0 (compatible; MSIE 5.0; Windows 98; DigExt)
```

In contrast, Netscape Navigator 4.75 would display the following:

```
Mozilla/4.75 [en] (Win98; U)
```

Finally, the Opera browser displays:

```
Mozilla/4.73 (Windows 98; U) Opera 4.02  [en]
```

Sniffer.php will make use of the information returned by $HTTP_USER_AGENT, parsing out the relevant pieces using various regular expression and string-handling functions. Before reviewing the code, take a moment to read through the following pseudocode:

- Two functions are used to determine the browser and operating system information: `browser_info()` and `opsys_info()`. Let's start with the `browser_info()` pseudocode.

- Determine the browser type using the `ereg()` function. Although it is slower than using another non-Perl-style function such as `strstr()`, it comes in handy because a regular expression can be used to determine the browser version.

- Use a compound *if* statement to test for the following browsers and their versions: Internet Explorer, Opera, Netscape, and unknown. Simple enough.

- The resulting browser and version is returned in an array.
- The `opsys_info()` function determines the operating system type. This time, the `strstr()` function is used because there is no need to use a regular expression to determine the OS.
- A compound *if* statement is used to test for the following operating systems: Windows, Linux, UNIX, Macintosh, and unknown.

The resulting operating system is returned.

Listing 8-3: Determining client operating system and browser

```
<?
/*
File: sniffer.php
Purpose: Determines browser type / version and platform information
Date: August 24, 2000
*/

// Function: browser_info
// Purpose: Returns browser type and version

function browser_info ($agent) {
     // Determine browser type
     //  Search for Internet Explorer signature.
     if (ereg( 'MSIE ([0-9].[0-9]{1,2})', $agent, $version)) :
          $browse_type = "IE";
          $browse_version = $version[1];

     //  Search for Opera signature.
     elseif (ereg( 'Opera ([0-9].[0-9]{1,2})', $agent, $version)) :
          $browse_type = "Opera";
          $browse_version = $version[1];

     //  Search for Netscape signature. The search for the Netscape browser
     //  *must* take place after the search for the Internet Explorer and Opera
     //  browsers, because each likes to call itself
     //  Mozilla as well as by its actual name.
     elseif (ereg( 'Mozilla/([0-9].[0-9]{1,2})', $agent, $version)) :
          $browse_type = "Netscape";
          $browse_version = $version[1];
```

```
        // If not Internet Explorer, Opera, or Netscape, then call it unknown.
        else :
            $browse_type = "Unknown";
            $browse_version = "Unknown";
        endif;

        // return the browser type and version as array
        return array($browse_type, $browse_version);

    } // end browser_info

    // Function: opsys_info
    // Purpose: Returns the user operating system

    function opsys_info($agent) {
        // Determine operating system
        // Search for Windows platform
        if ( strstr ($agent, 'Win') ) :
            $opsys = "Windows";

        // Search for Linux platform
        elseif ( strstr($agent, 'Linux') ) :
            $opsys = "Linux";

        // Search for UNIX platform
        elseif ( strstr ($agent, 'Unix') ) :
            $opsys = "Unix";

        // Search for Macintosh platform
        elseif ( strstr ($agent,'Mac') ) :
            $opsys = "Macintosh";

        // Platform is unknown
        else :
            $opsys = "Unknown";
        endif;

        // return the operating system
        return $opsys;

    } // end opsys_info

    // receive returned array as a list
```

```
list ($browse_type, $browse_version) = browser_info ($HTTP_USER_AGENT);
$operating_sys = opsys_info ($HTTP_USER_AGENT);

print "Browser Type: $browse_type <br>";
print "Browser Version: $browse_version <br>";
print "Operating System: $operating_sys <br>";

?>
```

Easy as that! For example, if the user is using Netscape 4.75 on a Windows machine, the following will be displayed:

```
Browser Type: Netscape
Browser Version: 4.75
Operating System: Windows
```

Next chapter, you'll learn how to perform page redirects and even create style sheets based on the operating system and browser.

What's Next?

This chapter covered quite a bit of ground. After all, what good would a programming language be to you if you couldn't work with text? In particular, the following subjects were covered:

- A general introduction to regular expressions, both POSIX and Perl style
- PHP's predefined regular expression functionality
- Manipulating string length
- Determining string length
- Faster alternatives to PHP's regular expression functionality
- Converting plain text to HTML and vice versa
- Manipulating character case of strings

Next chapter begins Part II of this book, which also happens to be my favorite. Here we begin a survey of PHP's Web capabilities, covering dynamic content creation, file inclusion, and basic template generation. Subsequent chapters in Part II delve into HTML forms usage, databasing, session tracking, and advanced templates. So hold on to your hat; things are about to get really interesting!