CHAPTER 9

Strings and Regular Expressions

As programmers, we build applications that are based on established rules regarding the classification, parsing, storage, and display of information, whether that information consists of gourmet recipes, store sales receipts, poetry, or some other collection of data. In this chapter, we examine many of the PHP functions that you'll undoubtedly use on a regular basis when performing such tasks.

This chapter covers the following topics:

- **PHP 5's new string offset syntax:** In an effort to remove ambiguity and pave the way for potential optimization of run-time string processing, a change to the string offset syntax was made in PHP 5.
- **Regular expressions:** A brief introduction to regular expressions touches upon the features and syntax of PHP's two supported regular expression implementations: POSIX and Perl. Following that is a complete introduction to PHP's respective function libraries.
- **String manipulation:** It's conceivable that throughout your programming career, you'll somehow be required to modify every conceivable aspect of a string. Many of the powerful PHP functions that can help you to do so are introduced in this chapter.
- **The PEAR `Validate_US` package:** In this and subsequent chapters, various PEAR packages are introduced that are relevant to the respective chapter's subject matter. This chapter introduces `Validate_US`, a PEAR package that is useful for validating the syntax for items of information commonly used in applications of all types, including phone numbers, social security numbers, ZIP codes, and state abbreviations. (If you're not familiar with PEAR, it's introduced in Chapter 11.)

Complex (Curly) Offset Syntax

Because PHP is a loosely typed language, it makes sense that a string could also easily be treated as an array. Therefore, any string, `php` for example, could be treated as both a contiguous entity and as a collection of three characters, meaning that you could output such a string in two fashions:

```
<?php
   $thing = "php";
   echo $thing;
   echo "<br />";
   echo $thing[0];
   echo $thing[1];
   echo $thing[2];
?>
```

This returns the following:

```
php
php
```

Although this behavior is quite convenient, it isn't without problems. For starters, it invites ambiguity. Looking at the code, was it the developer's intention to treat this data as a string or as an array? Also, this loose syntax prevents you from creating any sort of run-time code optimization intended solely for strings, because the scripting engine can't differentiate between strings and arrays. To resolve this problem, the square bracket offset syntax has been deprecated in preference to curly bracket syntax when working with strings. Here's another look at the previous example, this time using the preferred syntax:

```
<?php
   $thing = "php";
   echo $thing;
   echo "<br />";
   echo $thing{0};
   echo $thing{1};
   echo $thing{2};
?>
```

This example yields the same results as the original version.

The square bracket syntax has been around so long that it's unlikely to go away any time soon, if ever. Nonetheless, in the spirit of clean programming practice, it's suggested that you migrate to the curly bracketing syntax style for future applications.

Regular Expressions

Regular expressions provide the foundation for describing or matching data according to defined syntax rules. A regular expression is nothing more than a pattern of characters itself, matched against a certain parcel of text. This sequence may be a pattern with which you are already familiar, such as the word "dog," or it may be a pattern with specific meaning in the context of the world of pattern matching, `<(?)>.*<\ /.?>` for example.

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, this chapter provides just a basic introduction to both, leaving 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 regular expression pro, feel free to skip past the tutorial to the section "PHP's Regular Expression Functions (POSIX Extended)."

Regular Expression Syntax (POSIX)

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

The simplest regular expression is one that matches a single character, such as `g`, which would match 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.

You can also test for several different expressions simultaneously by using the pipe (`|`) operator. For example, you could test for `php or zend` via the regular expression `php|zend`.

Prior to introducing PHP's POSIX-based regular expression functions, we'll introduce three syntactical variations that POSIX supports for easily locating different character sequences: *brackets*, *quantifiers*, and *predefined character classes*.

Brackets

Brackets (`[]`) have a special meaning when used in the context of regular expressions, which are used to find a range of characters. Contrary to the regular expression `php`, which will find strings containing the explicit string `php`, the regular expression `[php]` will find any string containing the character `p` or `h`. Bracketing plays a significant role in regular expressions, because 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-Za-z]` matches any character from uppercase `A` through lowercase `z`.

Of course, the ranges shown here 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, with each special character having a specific connotation. The `+`, `*`, `?`, `{occurrence_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 one `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.
- `<b>(.*)</b>` matches any string enclosed within `<b>` and `</b>` (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. If you want 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. Standard classes include:

- `[:alpha:]`: Lowercase and uppercase alphabetical characters. This can also be specified as `[A-Za-z]`.
- `[:alnum:]`: Lowercase and uppercase alphabetical characters and numerical digits. This can also be specified as `[A-Za-z0-9]`.
- `[:cntrl:]`: Control characters such as a tab, escape, or backspace.

- [:digit:]: Numerical digits 0 through 9. This can also be specified as [0-9].
- [:graph:]: Printable characters found in the range of ASCII 33 to 126.
- [:lower:]: Lowercase alphabetical characters. This can also be specified as [a-z].
- [:punct:]: Punctuation characters, including ~ ` ! @ # $ % ^ & * ( ) - _ + = { } [ ] : ; ' < > , . ? and /.
- [:upper:]: Uppercase alphabetical characters. This can also be specified as [A-Z].
- [:space:]: Whitespace characters, including the space, horizontal tab, vertical tab, new line, form feed, or carriage return.
- [:xdigit:]: Hexadecimal characters. This can also be specified as [a-fA-F0-9].

PHP's Regular Expression Functions (POSIX Extended)

PHP currently offers seven functions for searching strings using POSIX-style regular expressions: ereg(), ereg_replace(), eregi(), eregi_replace(), split(), spliti(), and sql_regcase(). These functions are discussed in this section.

ereg()

boolean ereg (string *pattern*, string *string* [, array *regs*])

The ereg() function executes a case-sensitive search of string for pattern, returning TRUE if the pattern is found and FALSE otherwise. Here's how you could use ereg() to ensure that a username consists solely of lowercase letters:

```
<?php
   $username = "jasoN";
   if (ereg("([^a-z])",$username)) echo "Username must be all lowercase!";
?>
```

In this case, ereg() will return TRUE, causing the error message to output.

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, you could segment a URL into several pieces, as shown here:

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

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

   echo $regs[0];     // outputs the entire string "http://www.apress.com"
   echo "<br>";
   echo $regs[1];     // outputs "http://www"
   echo "<br>";
```

```
   echo $regs[2];     // outputs "apress"
   echo "<br>";
   echo $regs[3];     // outputs "com"
?>
```

This returns:

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

eregi()

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

The `eregi()` function searches `string` for `pattern`. Unlike `ereg()`, the search is case insensitive. This function can be useful when checking the validity of strings, such as passwords. This concept is illustrated in the following example:

```
<?php
   $pswd = "jasongild";
   if (!eregi("^[a-zA-Z0-9]{8,10}$", $pswd))
   echo "The password must consist solely of alphanumeric characters,
         and must be 8-10 characters in length!";
?>
```

In this example, the user must provide an alphanumeric password consisting of 8 to 10 characters, or else an error message is displayed.

ereg_replace()

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

The `ereg_replace()` function operates much like `ereg()`, except that the functionality is extended to finding and replacing `pattern` with `replacement` instead of simply locating it. If no matches are found, the string will remain unchanged. Like `ereg()`, `ereg_replace()` is case sensitive. Consider an example:

```
<?php
   $text = "This is a link to http://www.wjgilmore.com/.";
   echo ereg_replace("http://([a-zA-Z0-9./-]+)$", "<a href=\"\\0\">\\0</a>",
                     $text);
?>
```

This returns:

```
This is a link to
<a href="http://www.wjgilmore.com/">http://www.wjgilmore.com</a>.
```

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-zA-Z/]+)",
                    "<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_replace()

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

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

split()

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

The `split()` function divides `string` into various elements, with the boundaries of each element based on the occurrence of `pattern` in `string`. The optional input parameter `limit` is used to specify 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 break a string into pieces based on occurrences of horizontal tabs and newline characters:

```
<?php
   $text = "this is\tsome text that\nwe might like to parse.";
   print_r(split("[\n\t]",$text));
?>
```

This returns:

```
Array ( [0] => this is [1] => some text that [2] => we might like to parse. )
```

spliti()

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

The `spliti()` function operates exactly in the same manner as its sibling `split()`, except that it is case insensitive.

sql_regcase()

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

The `sql_regcase()` function converts each character in `string` into a bracketed expression containing two characters. If the character is alphabetic, the bracket will contain both forms; otherwise, the original character will be left unchanged. This function is particularly useful when PHP is used in conjunction with products that support only case-sensitive regular expressions. Here's how you would use `sql_regcase()` to convert a string:

```
<?php
   $version = "php 4.0";
   print sql_regcase($version);
   // outputs [Pp] [Hh] [Pp] 4.0
?>
```

Regular Expression Syntax (Perl Style)

Perl has long been considered one of the greatest parsing languages ever written, and it 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 derivation of the POSIX implementation, resulting in considerable similarities between the two. You can use any of the quantifiers introduced in the previous POSIX section. The remainder of this section is devoted to a brief introduction of Perl regular expression syntax. Let's start with a simple example of a Perl-based regular expression:

```
/food/
```

Notice that the string `food` is enclosed between two forward slashes. Just like with POSIX regular expressions, 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`.

Modifiers

Often, you'll want to tweak the interpretation of a regular expression; for example, you may want to tell the regular expression to execute a case-insensitive search or to ignore comments embedded within its syntax. These tweaks are known as *modifiers,* and they go a long way toward helping you to write short and concise expressions. A few of the more interesting modifiers are outlined in Table 9-1.

Table 9-1. *Six Sample Modifiers*

Modifier	Description
`i`	Perform a case-insensitive search.
`g`	Find all occurrences (perform a global search).
`m`	Treat a string as several (`m` for multiple) lines. By default, the `^` and `$` characters match at the very start and very end of the string in question. Using the `m` modifier will allow for `^` and `$` to match at the beginning of any line in a string.
`s`	Treat a string as a single line, ignoring any newline characters found within; this accomplishes just the opposite of the `m` modifier.
`x`	Ignore whitespace and comments within the regular expression.
`U`	Stop at the first match. Many quantifiers are "greedy"; they match the pattern as many times as possible rather than just stop at the first match. You can cause them to be "ungreedy" with this modifier.

These modifiers are placed directly after the regular expression; for example, `/string/i`. Let's consider a few examples:

- `/wmd/i`: Matches `WMD`, `wMD`, `WMd`, `wmd`, and any other case variation of the string `wmd`.
- `/taxation/gi`: Case insensitivity locates all occurrences of the word *taxation*. You might use the global modifier to tally up the total number of occurrences, or use it in conjunction with a replacement feature to replace all occurrences with some other string.

Metacharacters

Another useful thing you can do with Perl regular expressions is use various metacharacters to search for matches. A *metacharacter* is simply an alphabetical character preceded by a backslash that symbolizes special meaning. A list of useful metacharacters follows:

- `\A`: Matches only at the beginning of the string.
- `\b`: Matches a word boundary.
- `\B`: Matches anything but a word boundary.

- `\d`: Matches a digit character. This is the same as `[0-9]`.
- `\D`: Matches a nondigit character.
- `\s`: Matches a whitespace character.
- `\S`: Matches a nonwhitespace character.
- `[]`: Encloses a character class. A list of useful character classes was provided in the previous section.
- `()`: Encloses a character grouping or defines a back reference.
- `$`: Matches the end of a line.
- `^`: Matches the beginning of a line.
- `.`: Matches any character except for the newline.
- `\`: Quotes the next metacharacter.
- `\w`: Matches any string containing solely underscore and alphanumeric characters. This is the same as `[a-zA-Z0-9_]`.
- `\W`: Matches a string, omitting the underscore and alphanumeric characters.

Let's consider a few examples:

```
/sa\b/
```

Because the word boundary is defined to be on the right side of the strings, this will match strings like `pisa` and `lisa`, but not `sand`.

```
/\blinux\b/i
```

This returns the first case-insensitive occurrence of the word `linux`.

```
/sa\B/
```

The opposite of the word boundary metacharacter is `\B`, matching on anything but a word boundary. This will match strings like `sand` and `Sally`, but not `Melissa`.

```
/\$\d+\g
```

This returns all instances of strings matching a dollar sign followed by one or more digits.

PHP's Regular Expression Functions (Perl Compatible)

PHP offers seven functions for searching strings using Perl-compatible regular expressions: `preg_grep()`, `preg_match()`, `preg_match_all()`, `preg_quote()`, `preg_replace()`, `preg_replace_callback()`, and `preg_split()`. These functions are introduced in the following sections.

preg_grep()

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

The preg_grep() function searches all elements of the array input, returning an array consisting of all elements matching pattern. Consider an example that uses this function to search an array for foods beginning with *p*:

```
<?php
   $foods = array("pasta", "steak", "fish", "potatoes");
   $food = preg_grep("/^p/", $foods);
   print_r($food);
?>
```

This returns:

```
Array ( [0] => pasta [3] => potatoes )
```

Note that the array corresponds to the indexed order of the input array. If the value at that index position matches, it's included in the corresponding position of the output array. Otherwise, that position is empty. If you want to remove those instances of the array that are blank, filter the output array through the function array_values(), introduced in Chapter 5.

The optional input parameter flags was added in PHP version 4.3. It accepts one value, PREG_GREP_INVERT. Passing this flag will result in retrieval of those array elements that do ***not*** match the pattern.

preg_match()

```
int preg_match (string pattern, string string [, array matches]
               [, int flags [, int offset]]])
```

The preg_match() function searches string for pattern, returning TRUE if it exists and FALSE otherwise. The optional input parameter pattern_array can 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:

```
<?php
   $line = "Vim is the greatest word processor ever created!";
   if (preg_match("/\bVim\b/i", $line, $match)) print "Match found!";
?>
```

For instance, this script will confirm a match if the word Vim or vim is located, but not simplevim, vims, or evim.

preg_match_all()

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

The preg_match_all() function matches all occurrences of pattern in string, assigning each occurrence to array pattern_array in the order you specify via the optional input parameter order. The order parameter accepts two values:

- 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 regular expression, and so on.
- PREG_SET_ORDER orders the array a bit differently than the default setting. $pattern_array[0] contains elements matched by the first parenthesized regular expression, $pattern_array[1] contains elements matched by the second parenthesized regular expression, and so on.

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

```
<?php
   $userinfo = "Name: <b>Zeev Suraski</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";
?>
```

This returns:

```
Zeev Suraski
PHP Guru
```

preg_quote()

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

The function preg_quote() inserts a backslash delimiter before every character of special significance to regular expression syntax. These special characters include: $ ^ * ( ) + = { } [ ] | \\ : < >. The optional parameter delimiter is used to specify what delimiter is used for the regular expression, causing it to also be escaped by a backslash. Consider an example:

```
<?php
   $text = "Tickets for the bout are going for $500.";
   echo preg_quote($text);
?>
```

This returns:

```
Tickets for the bout are going for \$500\.
```

preg_replace()

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

The preg_replace() function operates identically to ereg_replace(), except that it uses a Perl-based regular expression syntax, replacing all occurrences of pattern with replacement, and returning the modified result. The optional input parameter limit specifies how many matches should take place. Failing to set limit or setting it to -1 will result in the replacement of all occurrences. Consider an example:

```
<?php
   $text = "This is a link to http://www.wjgilmore.com/.";
   echo preg_replace("/http:\/\/(.*)\//", "<a href=\"\${0}\">\${0}</a>", $text);
?>
```

This returns:

```
This is a link to
<a href="http://www.wjgilmore.com/">http://www.wjgilmore.com/</a>.
```

Interestingly, the pattern and replacement input parameters can also be arrays. This function will cycle through each element of each array, making replacements as they are found. Consider this example, which we could market as a corporate report generator:

```
<?php
   $draft = "In 2006 the company faced plummeting revenues and scandal.";
   $keywords = array("/faced/", "/plummeting/", "/scandal/");
   $replacements = array("celebrated", "skyrocketing", "expansion");
   echo preg_replace($keywords, $replacements, $draft);
?>
```

This returns:

```
In 2006 the company celebrated skyrocketing revenues and expansion.
```

preg_replace_callback()

```
mixed preg_replace_callback(mixed pattern, callback callback, mixed str
  [, int limit])
```

Rather than handling the replacement procedure itself, the preg_replace_callback() function delegates the string-replacement procedure to some other user-defined function. The pattern parameter determines what you're looking for, while the str parameter defines the string you're searching. The callback parameter defines the name of the function to be used for the replacement task. The optional parameter limit specifies how many matches should take place. Failing to set limit or setting it to -1 will result in the replacement of all occurrences. In the following example, a function named acronym() is passed into preg_replace_callback() and is used to insert the long form of various acronyms into the target string:

```
<?php
   // This function will add the acronym long form
   // directly after any acronyms found in $matches
   function acronym($matches) {
      $acronyms = array(
         'WWW' => 'World Wide Web',
         'IRS' => 'Internal Revenue Service',
         'PDF' => 'Portable Document Format');
      if (isset($acronyms[$matches[1]]))
         return $matches[1] . " (" . $acronyms[$matches[1]] . ")";
      else
         return $matches[1];
   }

       // The target text
   $text = "The <acronym>IRS</acronym> offers tax forms in
                <acronym>PDF</acronym> format on the <acronym>WWW</acronym>.";
   // Add the acronyms' long forms to the target text
   $newtext = preg_replace_callback("/<acronym>(.*)<\/acronym>/U", 'acronym',
                                    $text);

   print_r($newtext);
?>
```

This returns:

```
The IRS (Internal Revenue Service) offers tax forms
in PDF (Portable Document Format) on the WWW (World Wide Web).
```

preg_split()

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

The `preg_split()` function operates exactly like `split()`, except that `pattern` can also be defined in terms of a regular expression. If the optional input parameter `limit` is specified, only `limit` number of substrings are returned. Consider an example:

```
<?php
   $delimitedText = "+Jason+++Gilmore++++++++++Columbus+++OH";
   $fields = preg_split("/\+{1,}/", $delimitedText);
    foreach($fields as $field) echo $field."<br />";
?>
```

This returns the following:

```
Jason
Gilmore
Columbus
OH
```

■ **Note** Later in this chapter, the section titled "Alternatives for Regular Expression Functions" offers several standard functions that can be used in lieu of regular expressions for certain tasks. In many cases, these alternative functions actually perform much faster than their regular expression counterparts.

Other String-Specific Functions

In addition to the regular expression–based functions discussed in the first half of this chapter, PHP offers over 100 functions collectively capable of manipulating practically every imaginable aspect of a string. To introduce each function would be out of the scope of this book and would only repeat much of the information in the PHP documentation. This section is devoted to a categorical FAQ of sorts, focusing upon the string-related issues that seem to most frequently appear within community forums. The section is divided into the following topics:

- Determining string length
- Comparing string length
- Manipulating string case
- Converting strings to and from HTML
- Alternatives for regular expression functions
- Padding and stripping a string
- Counting characters and words

Determining the Length of a String

Determining string length is a repeated action within countless applications. The PHP function `strlen()` accomplishes this task quite nicely.

strlen()

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

You can determine the length of a string with the `strlen()` function. This function returns the length of a string, where each character in the string is equivalent to one unit. The following example verifies whether a user password is of acceptable length:

```
<?php
   $pswd = "secretpswd";
   if (strlen($string) < 10) echo "Password is too short!";
?>
```

In this case, the error message will not appear, because the chosen password consists of 10 characters, whereas the conditional expression validates whether the target string consists of less than 10 characters.

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()`, and `strcspn()`. These functions are discussed in the following sections.

strcmp()

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

The `strcmp()` function performs a binary-safe, case-sensitive comparison of the strings `str1` and `str2`, returning one of three possible values:

- 0 if `str1` and `str2` are equal
- -1 if `str1` is less than `str2`
- 1 if `str2` is less than `str1`

Web sites often require a registering user to enter and confirm his chosen password, lessening the possibility of an incorrectly entered password as a result of a typing error. Because passwords are often case sensitive, `strcmp()` is a great function for comparing the two:

```
<?php
   $pswd = "supersecret";
   $pswd2 = "supersecret";
   if (strcmp($pswd,$pswd2) != 0) echo "Your passwords do not match!";
?>
```

Note that the strings must match exactly for `strcmp()` to consider them equal. For example, `Supersecret` is different from `supersecret`. If you're looking to compare two strings case-insensitively, consider `strcasecmp()`, introduced next.

Another common point of confusion regarding this function surrounds its behavior of returning 0 if the two strings are equal. This is different from executing a string comparison using the == operator, like so:

```
if ($str1 == $str2)
```

While both accomplish the same goal, which is to compare two strings, keep in mind that the values they return in doing so are different.

strcasecmp()

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

The `strcasecmp()` function operates exactly like `strcmp()`, except that its comparison is case insensitive. The following example compares two e-mail addresses, an ideal use for `strcasecmp()` because casing does not determine an e-mail address's uniqueness:

```
<?php
   $email1 = "admin@example.com";
   $email2 = "ADMIN@example.com";

   if (! strcasecmp($email1, $email2))
      print "The email addresses are identical!";
?>
```

In this case, the message is output, because `strcasecmp()` performs a case-insensitive comparison of `$email1` and `$email2` and determines that they are indeed identical.

strspn()

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

The `strspn()` function returns the length of the first segment in `str1` containing characters also in `str2`. Here's how you might use `strspn()` to ensure that a password does not consist solely of numbers:

```
<?php
   $password = "3312345";
   if (strspn($password, "1234567890") == strlen($password))
      echo "The password cannot consist solely of numbers!";
?>
```

In this case, the error message is returned, because `$password` does indeed consist solely of digits.

strcspn()

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

The `strcspn()` function returns the length of the first segment in `str1` containing characters not found in `str2`. Here's an example of password validation using `strcspn()`:

```
<?php
   $password = "a12345";
   if (strcspn($password, "1234567890") == 0) {
      print "Password cannot consist solely of numbers! ";
   }
?>
```

In this case, the error message will not be displayed, because `$password` does not consist solely of numbers.

Manipulating String Case

Four functions are available to aid you in manipulating the case of characters in a string: `strtolower()`, `strtoupper()`, `ucfirst()`, and `ucwords()`. These functions are discussed in this section.

strtolower()

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

The `strtolower()` function converts `str` to all lowercase letters, returning the modified string. Nonalphabetical characters are not affected. The following example uses `strtolower()` to convert a URL to all lowercase letters:

```
<?php
   $url = "http://WWW.EXAMPLE.COM/";
   echo strtolower($url);
?>
```

This returns:

```
http://www.example.com/
```

strtoupper()

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

Just as you can convert a string to lowercase, you can convert it to uppercase. This is accomplished with the function `strtoupper()`. Nonalphabetical characters are not affected. This example uses `strtoupper()` to convert a string to all uppercase letters:

```
<?php
   $msg = "i annoy people by capitalizing e-mail text.";
   echo strtoupper($msg);
?>
```

This returns:

```
I ANNOY PEOPLE BY CAPITALIZING E-MAIL TEXT.
```

ucfirst()

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

The ucfirst() function capitalizes the first letter of the string str, if it is alphabetical. Nonalphabetical characters will not be affected. Additionally, any capitalized characters found in the string will be left untouched. Consider this example:

```
<?php
   $sentence = "the newest version of PHP was released today!";
   echo ucfirst($sentence);
?>
```

This returns:

```
The newest version of PHP was released today!
```

Note that while the first letter is indeed capitalized, the capitalized word "PHP" was left untouched.

ucwords()

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

The ucwords() function capitalizes the first letter of each word in a string. Nonalphabetical characters are not affected. This example uses ucwords() to capitalize each word in a string:

```
<?php
   $title = "O'Malley wins the heavyweight championship!";
   echo ucwords($title);
?>
```

This returns:

```
O'Malley Wins The Heavyweight Championship!
```

Note that if "O'Malley" was accidentally written as "O'malley," ucwords() would not catch the error, as it considers a word to be defined as a string of characters separated from other entities in the string by a blank space on each side.

Converting Strings to and from HTML

Converting a string or an entire file into a form suitable for viewing on the Web (and vice versa) is easier than you would think. Several functions are suited for such tasks, all of which are introduced in this section. For convenience, this section is divided into two parts: "Converting Plain Text to HTML" and "Converting HTML to Plain Text."

Converting Plain Text to HTML

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

nl2br()

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

The `nl2br()` function converts all newline (`\n`) characters in a string to their XHTML-compliant equivalent, `<br />`. The newline characters could be created via a carriage return, or explicitly written into the string. The following example translates a text string to HTML format:

```
<?php
   $recipe = "3 tablespoons Dijon mustard
   1/3 cup Caesar salad dressing
   8 ounces grilled chicken breast
   3 cups romaine lettuce";
   // convert the newlines to <br />'s.
   echo nl2br($recipe);
?>
```

Executing this example results in the following output:

```
3 tablespoons Dijon mustard<br />
1/3 cup Caesar salad dressing<br />
8 ounces grilled chicken breast<br />
3 cups romaine lettuce
```

htmlentities()

```
string htmlentities (string str [, int quote_style [, int charset]])
```

During the general course of communication, you may come across many characters that are not included in a document's text encoding, or that are not readily available on the keyboard. Examples of such characters include the copyright symbol (©), cent sign (¢), and the French accent grave (è). To facilitate such shortcomings, a set of universal key codes was devised, known as *character entity references*. When these entities are parsed by the browser, they will be converted into their recognizable counterparts. For example, the three aforementioned characters would be presented as `&copy;`, `&cent;`, and `&Egrave;`, respectively.

The `htmlentities()` function converts all such characters found in `str` into their HTML equivalents. Because of the special nature of quote marks within markup, the optional `quote_style` parameter offers the opportunity to choose how they will be handled. Three values are accepted:

- `ENT_COMPAT`: Convert double-quotes and ignore single quotes. This is the default.
- `ENT_NOQUOTES`: Ignore both double and single quotes.
- `ENT_QUOTES`: Convert both double and single quotes.

A second optional parameter, `charset`, determines the character set used for the conversion. Table 9-2 offers the list of supported character sets. If `charset` is omitted, it will default to ISO-8859-1.

Table 9-2. *htmlentities()'s Supported Character Sets*

Character Set	Description
BIG5	Traditional Chinese
BIG5-HKSCS	BIG5 with additional Hong Kong extensions, traditional Chinese
cp866	DOS-specific Cyrillic character set
cp1251	Windows-specific Cyrillic character set
cp1252	Windows-specific character set for Western Europe
EUC-JP	Japanese
GB2312	Simplified Chinese
ISO-8859-1	Western European, Latin-1
ISO-8859-15	Western European, Latin-9
KOI8-R	Russian
Shift-JIS	Japanese
UTF-8	ASCII-compatible multibyte 8 encode

The following example converts the necessary characters for Web display:

```
<?php
   $advertisement = "Coffee at 'Cafè Française' costs $2.25.";
   echo  htmlentities($advertisement);
?>
```

This returns:

```
Coffee at 'Caf&egrave; Fran&ccedil;aise' costs $2.25.
```

Two characters were converted, the accent grave (è) and the cedilla (ç). The single quotes were ignored due to the default `quote_style` setting `ENT_COMPAT`.

htmlspecialchars()

```
string htmlspecialchars (string str [, int quote_style [, string charset]])
```

Several characters play a dual role in both markup languages and the human language. When used in the latter fashion, these characters must be converted into their displayable equivalents.

For example, an ampersand must be converted to &, whereas a greater-than character must be converted to >. The htmlspecialchars() function can do this for you, converting the following characters into their compatible equivalents:

- & becomes &
- " (double quote) becomes "
- ' (single quote) becomes '
- < becomes <
- > becomes >

This function is particularly useful in preventing users from entering HTML markup into an interactive Web application, such as a message board.

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

```
<?php
   $input = "I just can't get <<enough>> of PHP!";
   echo htmlspecialchars($input);
?>
```

Viewing the source, you'll see:

```
I just can't get &lt;&lt;enough&gt;&gt; of PHP &amp;amp!
```

If the translation isn't necessary, perhaps a more efficient way to do this would be to use strip_tags(), which deletes the tags from the string altogether.

■ **Tip** If you are using gethtmlspecialchars() in conjunction with a function like nl2br(), you should execute nl2br() after gethtmlspecialchars(); otherwise, the <br /> tags that are generated with nl2br() will be converted to visible characters.

get_html_translation_table()

```
array get_html_translation_table (int table [, int quote_style])
```

Using get_html_translation_table() is a convenient way to translate text to its HTML equivalent, returning one of the two translation tables (HTML_SPECIALCHARS or HTML_ENTITIES) specified by table. This returned value can then be used in conjunction with another predefined function, strtr() (formally introduced later in this section), to essentially translate the text into its corresponding HTML code.

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

```
<?php
   $string = "La pasta é il piatto piú amato in Italia";
   $translate = get_html_translation_table(HTML_ENTITIES);
   echo strtr($string, $translate);
?>
```

This returns the string formatted as necessary for browser rendering:

```
La pasta &eacute; il piatto pi&uacute; amato in Italia
```

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, you assigned it to the variable `$translated_string`.

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

```
<?php
   $entities = get_html_translation_table(HTML_ENTITIES);
   $translate = array_flip($entities);
   $string = "La pasta &eacute; il piatto pi&uacute; amato in Italia";
   echo strtr($string, $translate);
?>
```

This returns the following:

```
La pasta é il piatto piú amato in italia
```

strtr()

```
string strtr (string str, array replacements)
```

The `strtr()` function converts all characters in `str` to their corresponding match found in `replacements`. This example converts the deprecated bold (`<b>`) character to its XHTML equivalent:

```
<?php
   $table = array("<b>" => "<strong>", "</b>" => "</strong>");
   $html = "<b>Today In PHP-Powered News</b>";
   echo strtr($html, $table);
?>
```

This returns the following:

```
<strong>Today In PHP-Powered News</strong>
```

Converting HTML to Plain Text

You may sometimes need to convert an HTML file to plain text. The following function can help you accomplish this.

strip_tags()

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

The `strip_tags()` function removes all HTML and PHP tags from `str`, leaving only the text entities. 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:

```
<?php
   $input = "Email <a href='spammer@example.com'>spammer@example.com</a>";
   echo strip_tags($input);
?>
```

This returns the following:

```
Email spammer@example.com
```

The following sample strips all tags except the `<a>` tag:

```
<?php
   $input = "This <a href='http://www.example.com/'>example</a>
             is <b>awesome</b>!";
   echo strip_tags($input, "<a>");
?>
```

This returns the following:

```
This <a href='http://www.example.com/'>example</a> is awesome!
```

■ **Note** Another function that behaves like `strip_tags()` is `fgetss()`. This function is described in Chapter 10.

Alternatives for Regular Expression Functions

When you're 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 speed up the process considerably. Each of these functions is described in this section.

strtok()

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

The `strtok()` function parses the string `str` based on the characters found in `tokens`. One oddity about `strtok()` is that it must be continually called in order to completely tokenize a string; each call only tokenizes the next piece of the string. However, the `str` parameter needs to be specified only once, because the function keeps track of its position in `str` until it either completely tokenizes `str` or a new `str` parameter is specified. Its behavior is best explained via an example:

```
<?php
   $info = "J. Gilmore:jason@example.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>";
      // Don't include the first argument in subsequent calls.
      $tokenized = strtok($tokens);
   }
?>
```

This returns the following:

```
Element = J. Gilmore
Element = jason@example.com
Element = Columbus
Element = Ohio
```

parse_str()

```
void parse_str (string str [, array arr]))
```

The `parse_str()` function parses `string` into various variables, setting the variables in the current scope. If the optional parameter `arr` is included, the variables will be placed in that array instead. This function is particularly useful when handling URLs that contain HTML forms or other parameters passed via the query string. 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:

```
<?php
   // suppose that the URL is http://www.example.com?ln=gilmore&zip=43210
   parse_str($_SERVER['QUERY_STRING']);
   // after execution of parse_str(), the following variables are available:
   // $ln = "gilmore"
   // $zip = "43210"
?>
```

Note that `parse_str()` is unable to correctly parse the first variable of the query string if the string leads off with a question mark. Therefore, if you use a means other than `$_SERVER['QUERY_STRING']` for retrieving this parameter string, make sure you delete that preceding question mark before passing the string to `parse_str()`. The `ltrim()` function, introduced later in the chapter, is ideal for such tasks.

explode()

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

The `explode()` function divides the string `str` into an array of substrings. The original string is divided into distinct elements by separating it based on the character separator specified by `separator`. The number of elements can be limited with the optional inclusion of `limit`. Let's use `explode()` in conjunction with `sizeof()` and `strip_tags()` to determine the total number of words in a given block of text:

```
<?php
$summary = <<< summary
In the latest installment of the ongoing Developer.com PHP series,
I discuss the many improvements and additions to
<a href="http://www.php.net">PHP 5's</a> object-oriented architecture.
summary;
$words = sizeof(explode(' ',strip_tags($summary)));
echo "Total words in summary: $words";
?>
```

This returns:

```
Total words in summary: 22
```

The `explode()` function will always be considerably faster than `preg_split()`, `split()`, and `spliti()`. Therefore, always use it instead of the others when a regular expression isn't necessary.

implode()

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

Just as you can use the explode() function to divide a delimited string into various array elements, you concatenate array elements to form a single delimited string. This is accomplished with the implode() function. This example forms a string out of the elements of an array:

```
<?php
   $cities = array("Columbus", "Akron", "Cleveland", "Cincinnati");
   echo implode("|", $cities);
?>
```

This returns:

```
Columbus|Akron|Cleveland|Cincinnati
```

■ **Note** join() is an alias for implode().

strpos()

```
int strpos (string str, string substr [, int offset])
```

The strpos() function finds the position of the first case-sensitive occurrence of substr in str. The optional input parameter offset specifies the position at which to begin the search. If substr is not in str, strpos() will return FALSE. The optional parameter offset determines the position from which strpos() will begin searching. The following example determines the timestamp of the first time index.html is accessed:

```
<?php
   $substr = "index.html";
$log = <<< logfile
192.168.1.11:/www/htdocs/index.html:[2006/02/10:20:36:50]
192.168.1.13:/www/htdocs/about.html:[2006/02/11:04:15:23]
192.168.1.15:/www/htdocs/index.html:[2006/02/15:17:25]
logfile;

   // what is first occurrence of the time $substr in log?
   $pos = strpos($log, $substr);

   // Find the numerical position of the end of the line
   $pos2 = strpos($log,"\n",$pos);

   // Calculate the beginning of the timestamp
   $pos = $pos + strlen($substr) + 1;
```

```
   // Retrieve the timestamp
   $timestamp = substr($log,$pos,$pos2-$pos);

   echo "The file $substr was first accessed on: $timestamp";
?>
```

This returns the position in which the file `index.html` was first accessed:

```
The file index.html was first accessed on: [2006/02/10:20:36:50]
```

stripos()

```
int stripos(string str, string substr [, int offset])
```

The function `stripos()` operates identically to `strpos()`, except that that it executes its search case-insensitively.

strrpos()

```
int strrpos (string str, char substr [, offset])
```

The `strrpos()` function finds the last occurrence of `substr` in `str`, returning its numerical position. The optional parameter `offset` determines the position from which `strrpos()` will begin searching. Suppose you wanted to pare down lengthy news summaries, truncating the summary and replacing the truncated component with an ellipsis. However, rather than simply cut off the summary explicitly at the desired length, you want it to operate in a user-friendly fashion, truncating at the end of the word closest to the truncation length. This function is ideal for such a task. Consider this example:

```
<?php
   // Limit $summary to how many characters?
   $limit = 100;

$summary = <<< summary
In the latest installment of the ongoing Developer.com PHP series,
I discuss the many improvements and additions to
<a href="http://www.php.net">PHP 5's</a> object-oriented
architecture.
summary;

   if (strlen($summary) > $limit)
      $summary = substr($summary, 0, strrpos(substr($summary, 0, $limit),
                        ' ')) . '...';
   echo $summary;
?>
```

This returns:

```
In the latest installment of the ongoing Developer.com PHP series,
I discuss the many...
```

str_replace()

```
mixed str_replace (string occurrence, mixed replacement, mixed str [, int count])
```

The `str_replace()` function executes a case-sensitive search for `occurrence` in `str`, replacing all instances with `replacement`. If `occurrence` is not found in `str`, then `str` is returned unmodified. If the optional parameter `count` is defined, then only `count` `occurrences` found in `str` will be replaced.

This function is ideal for hiding e-mail addresses from automated e-mail address retrieval programs:

```
<?php
   $author = "jason@example.com";
   $author = str_replace("@","(at)",$author);
   echo "Contact the author of this article at $author.";
?>
```

This returns:

```
Contact the author of this article at jason(at)example.com.
```

str_ireplace()

```
mixed str_ireplace(mixed occurrence, mixed replacement, mixed str [, int count])
```

The function `str_ireplace()` operates identically to `str_replace()`, except that it is capable of executing a case-insensitive search.

strstr()

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

The `strstr()` function returns the remainder of `str` beginning at the first `occurrence`. This example uses the function in conjunction with the `ltrim()` function to retrieve the domain name of an e-mail address:

```
<?php
   $url = "sales@example.com";
   echo ltrim(strstr($url, "@"),"@");
?>
```

This returns the following:

```
example.com
```

substr()

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

The `substr()` function returns the part of `str` located between the `start` and `start + length` positions. 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 `str`. There are four points to keep in mind when using this function:

- If `start` is positive, the returned string will begin at the `start` position of the string.
- If `start` is negative, the returned string will begin at the string length – `start` position of the string.
- If `length` is provided and is positive, the returned string will consist of the characters between `start` and `(start + length)`. If this distance surpasses the total string length, then only the string between `start` and the string's end will be returned.
- If `length` is provided and is negative, the returned string will end `length` characters from the end of `str`.

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

Consider a basic example:

```
<?php
   $car = "1944 Ford";
   echo substr($car, 5);
?>
```

This returns the following:

```
Ford
```

The following example uses the `length` parameter:

```
<?php
   $car = "1944 Ford";
   echo substr($car, 0, 4);
?>
```

This returns the following:

```
1944
```

The final example uses a negative `length` parameter:

```
<?php
   $car = "1944 Ford";
   $yr = echo substr($car, 2, -5);
?>
```

This returns:

```
44
```

substr_count()

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

The `substr_count()` function returns the number of times `substring` occurs in `str`. The following example determines the number of times an IT consultant uses various buzzwords in his presentation:

```
<?php
   $buzzwords = array("mindshare", "synergy", "space");
$talk = <<< talk
I'm certain that we could dominate mindshare in this space with our new product,
establishing a true synergy between the marketing and product development teams.
We'll own this space in three months.
talk;
   foreach($buzzwords as $bw) {
      echo "The word $bw appears ".substr_count($talk,$bw)." time(s).<br />";
   }
?>
```

This returns the following:

```
The word mindshare appears 1 time(s).
The word synergy appears 1 time(s).
The word space appears 2 time(s).
```

substr_replace()

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

The `substr_replace()` function replaces a portion of `str` with `replacement`, beginning the substitution at `start` position of `str`, and ending at `start` + `length` (assuming that the optional input parameter `length` is included). Alternatively, the substitution will stop on the complete placement of `replacement` in `str`. There are several behaviors you should keep in mind 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 (`str` 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 (`str` length – `length`) characters.

Suppose you built an e-commerce site, and within the user profile interface, you want to show just the last four digits of the provided credit card number. This function is ideal for such a task:

```
<?php
   $ccnumber = "1234567899991111";
   echo substr_replace($ccnumber,"************",0,12);
?>
```

This returns:

```
************1111
```

Padding and Stripping a String

For formatting reasons, you sometimes need to modify the string length via either padding or stripping characters. PHP provides a number of functions for doing so. We'll examine many of the commonly used functions in this section.

ltrim()

```
string ltrim (string str [, string charlist])
```

The `ltrim()` function removes various characters from the beginning of `str`, including whitespace, the horizontal tab (`\t`), newline (`\n`), carriage return (`\r`), NULL (`\0`), and vertical tab (`\x0b`).You can designate other characters for removal by defining them in the optional parameter `charlist`.

rtrim()

```
string rtrim(string str [, string charlist])
```

The rtrim() function operates identically to ltrim(), except that it removes the designated characters from the right side of str.

trim()

```
string trim (string str [, string charlist])
```

You can think of the trim() function as a combination of ltrim() and rtrim(), except that it removes the designated characters from both sides of str.

str_pad()

```
string str_pad (string str, int length [, string pad_string [, int pad_type]])
```

The str_pad() function pads str to length characters. If the optional parameter pad_string is not defined, str will be padded with blank spaces; otherwise, it will be padded with the character pattern specified by pad_string. By default, the string will be padded to the right; however, the optional parameter pad_type may be assigned the values 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():

```
<?php
   echo str_pad("Salad", 10)." is good.";
?>
```

This returns the following:

```
Salad     is good.
```

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

```
<?php
   $header = "Log Report";
   echo str_pad ($header, 20, "=+", STR_PAD_BOTH);
?>
```

This returns:

```
=+=+=Log Report=+=+=
```

Note that str_pad() truncates the pattern defined by pad_string if length is reached before completing an entire repetition of the pattern.

Counting Characters and Words

It's often useful to determine the total number of characters or words in a given string. Although PHP's considerable capabilities in string parsing has long made this task trivial, two functions were recently added that formalize the process. Both functions are introduced in this section.

count_chars()

```
mixed count_chars(string str [, mode])
```

The function `count_chars()` offers information regarding the characters found in `str`. Its behavior depends upon how the optional parameter `mode` is defined:

- `0`: Returns an array consisting of each found byte value as the key and the corresponding frequency as the value, even if the frequency is zero. This is the default.
- `1`: Same as `0`, but returns only those byte-values with a frequency greater than zero.
- `2`: Same as `0`, but returns only those byte-values with a frequency of zero.
- `3`: Returns a string containing all located byte-values.
- `4`: Returns a string containing all unused byte-values.

The following example counts the frequency of each character in `$sentence`:

```
<?php
   $sentence = "The rain in Spain falls mainly on the plain";
   // Retrieve located characters and their corresponding frequency.
   $chart = count_chars($sentence, 1);

   foreach($chart as $letter=>$frequency)
      echo "Character ".chr($letter)." appears $frequency times<br />";
?>
```

This returns the following:

```
Character  appears 8 times
Character S appears 1 times
Character T appears 1 times
Character a appears 5 times
Character e appears 2 times
Character f appears 1 times
Character h appears 2 times
Character i appears 5 times
Character l appears 4 times
Character m appears 1 times
Character n appears 6 times
Character o appears 1 times
Character p appears 2 times
Character r appears 1 times
```

```
Character s appears 1 times
Character t appears 1 times
Character y appears 1 times
```

str_word_count()

```
mixed str_word_count (string str [, int format])
```

The function `str_word_count()` offers information regarding the total number of words found in `str`. If the optional parameter `format` is not defined, it will simply return the total number of words. If `format` is defined, it modifies the function's behavior based on its value:

- `1`: Returns an array consisting of all words located in `str`.
- `2`: Returns an associative array, where the key is the numerical position of the word in `str`, and the value is the word itself.

Consider an example:

```
<?php
$summary = <<< summary
In the latest installment of the ongoing Developer.com PHP series,
I discuss the many improvements and additions to PHP 5's
object-oriented architecture.
summary;
   $words = str_word_count($summary);
   echo "Total words in summary: $words";
?>
```

This returns the following:

```
Total words in summary: 23
```

You can use this function in conjunction with `array_count_values()` to determine the frequency in which each word appears within the string:

```
<?php
$summary = <<< summary
In the latest installment of the ongoing Developer.com PHP series,
I discuss the many improvements and additions to PHP 5's
object-oriented architecture.
summary;
   $words = str_word_count($summary,2);
   $frequency = array_count_values($words);
   print_r($frequency);
?>
```

This returns the following:

```
Array ( [In] => 1 [the] => 3 [latest] => 1 [installment] => 1 [of] => 1
[ongoing] => 1 [Developer] => 1 [com] => 1 [PHP] => 2 [series] => 1
[I] => 1 [discuss] => 1 [many] => 1 [improvements] => 1 [and] => 1
[additions] => 1 [to] => 1 [s] => 1 [object-oriented] => 1
[architecture] => 1 )
```

Taking Advantage of PEAR: Validate_US

Regardless of whether your Web application is intended for use in banking, medical, IT, retail, or some other industry, chances are that certain data elements will be commonplace. For instance, it's conceivable you'll be tasked with inputting and validating a telephone number or state abbreviation, regardless of whether you're dealing with a client, patient, staff member, or customer. Such repeatability certainly presents the opportunity to create a library that is capable of handling such matters, regardless of the application. Indeed, because we're faced with such repeatable tasks, it follows that so are other programmers. Therefore, it's always prudent to investigate whether somebody has already done the hard work for us and made a package available via PEAR.

■ **Note** If you're unfamiliar with PEAR, then take some time to review Chapter 11 before continuing.

Sure enough, our suspicions have proved fruitful, because a quick PEAR search turns up `Validate_US`, a package that is capable of validating various informational items specific to the United States. Although still in beta at press time, `Validate_US` is already capable of syntactically validating phone numbers, social security numbers, state abbreviations, and ZIP codes. This section introduces `Validate_US`, showing you how to install and implement this immensely useful package.

Installing Validate_US

To take advantage of `Validate_US`, you need to install it. The process for doing so follows:

```
%>pear install -f Validate_US
Warning: Validate_US is state 'beta' which is less stable than state 'stable'
downloading Validate_US-0.5.0.tgz ...
Starting to download Validate_US-0.5.0.tgz (5,611 bytes)
.....done: 5,611 bytes
install ok: Validate_US 0.5.0
```

Note that because `Validate_US` is still a beta release, you need to pass the `-f` option to the `install` command in order to force installation. Once you have installed the package, proceed to the next section.

Using Validate_US

The Validate_US package is extremely easy to use; simply instantiate the Validate_US() class and call the appropriate validation method. In total there are seven methods, three of which are relevant to this discussion, including:

- phoneNumber(): Validates a phone number, returning TRUE on success and FALSE otherwise. It accepts phone numbers in a variety of formats, including xxx xxx-xxxx, (xxx) xxx-xxxx, and similar combinations without dashes, parentheses, or spaces. For example, (614)999-9999, 6149999999, and (614)9999999 are all valid, whereas (6149999999, 614-999-9999, and 614999 are not.
- postalCode(): Validates a ZIP code, returning TRUE on success and FALSE otherwise. It accepts ZIP codes in a variety of formats, including xxxxx, xxxxxxxxx, xxxxx-xxxx, and similar combinations without the dash. For example, 43210 and 43210-0362 are both valid, whereas 4321 and 4321009999 are not.
- region(): Validates a state abbreviation, returning TRUE on success and FALSE otherwise. It accepts two-letter state abbreviations as supported by the United States Postal Service (http://www.usps.com/ncsc/lookups/usps_abbreviations.html). For example, OH, CA, and NY are all valid, whereas CC, DUI, and BASF are not.
- ssn(): Validates a social security number (SSN) by not only checking the SSN syntax but also reviewing validation information made available via the Social Security Administration Web site (http://www.ssa.gov/), returning TRUE on success and FALSE otherwise. It accepts SSNs in a variety of formats, including xxx-xx-xxxx, xxx xx xxx, xxx/xx/xxxx, xxx\txx\txxxx (\t = tab), xxx\nxx\nxxxx (\n = newline), or any nine-digit combination thereof involving dashes, forward slashes, tabs, or newline characters. For example, 479-35-6432 and 591467543 are valid, whereas 999999999, 777665555, and 45678 are not.

Once you have an understanding of the method definitions, implementation is trivial. For example, suppose you want to validate a phone number. Just include the Validate_US class and call phoneNumber() like so:

```
<?php
   include "Validate/US.php";
   $validate = new Validate_US();
   echo $validate->phoneNumber("614-999-9999");
?>
```

Because phoneNumber() returns a boolean, in this example a 1 will be returned. Contrast this with supplying 614-876530932 to phoneNumber(), which will return FALSE.

Summary

Many of the functions introduced in this chapter will be among the most commonly used within your PHP applications, as they form the crux of the language's string-manipulation capabilities.

In the next chapter, we'll turn our attention toward another set of well-worn functions: those devoted to working with the file and operating system.