CS 85: PHP Programming

Santa Monica College Computer Science & Information Systems Dept.

Module 4: Strings

Working with Strings

Parse Stings

Compare Stings

Single and multiple strings

PHP is an extremely powerful tool for generating HTML code as well as processing user submitted data. HTML code and use data is commonly in a string format. This module will cover how to format and manipulate strings. By using built-in PHP functions and regular expressions a PHP developer can perform such task as comparing strings, match and replacing substrings, joining or splitting strings.

# What is a String?

A string is any sequence of characters, like "Hello world!", ‘a’, “123”, “a1b2c3”, $singleVar, etc. A string can either be a text string surrounded by either single or double quotes. A variable with a string assigned to it can also be considered a string, because it stores a string.

When working with strings, it is important to remember all strings must be enclosed in a single or double quote and the ending quote and must match the beginning quote. If for example you want to the include a single or double quote in the string, the opposite quotes should be used to declared the string. For example $myQuote = ‘<p>”Never on Tuesday … “<p>’;

# String Operators

There are two string operators in PHP. The first string operator is the concatenation operator ('.') a single period, which returns the concatenation of its right and left arguments. The second is the concatenating assignment operator ('.=') is the period and equal symbol, which appends the string on the right side to the string on the left side. The string in these examples can be a variable with a string value or the actual string.

<?php  
$a = "Hello ";  
$b = $a . "World!"; // now $b contains "Hello World!"  
  
$a = "Hello ";  
$a .= "World!";     // now $a contains "Hello World!"  
?>

# Escape Characters

When working with special characters in PHP strings, special care has to be taken to ensure the string is interpreted by the PHP engine correctly. If a special character is added to a strings, unusual behavior or fetal execution errors might occur.

For example, this line of code will have unexpected behavior. The single quote in the word “it’s” will prematurely end the strings, when the PHP interpreter match the beginning single quote to the “it’s” word single quote. Upon execution of this script, a parse errorwill occur.

//Special character caused script to error out

echo ‘It’s your birthday!’;

By using escape characters in your string, it flags the PHP interpreter that the character that follows the escape character serves a special purpose. In PHP the escape character is the backslash (\). By using the backslash in front of a special character such a single quote (‘) or dollar sign ($) symbol, is telling the PHP interpreter to treat the single quote or dollar symbol as normal characters and not as the beginning or ending of a PHP string or in the case of the dollar symbol ($), not a variable but only as the text symbol of a dollar sign.

//Special Character with Escape Character

echo ‘It\’s your birthday!’;

This is only one example of where the escape character was use to include a special character in a string, but the escape character can also perform an escape sequence where a string related change will be made. For example \t will add a tab space to a string, \n will add a line break to a string. Here is a list of escape sequences:

\’ to escape ‘ within single quoted string

\” to escape “ within double quoted string

\n is replaced by the newline character

\r is replaced by the carriage-return character

\t is replaced by the tab character

\$ is replaced by the dollar sign itself ($)

\\ is replaced by a single backslash (\)

Sample Code:

$guestA = "Peter Smith";

echo "<p>\"Say What!\" exclaimed $ guestA.</p>";

# String Syntax

Single quoted will present strings nearly identically as written. Meaning an escape sequence is not always required. For example, variables and most escape sequences will not be interpreted. The exception to this rule, is the escape sequence is required when single quotes used to enclosed a literal string and the literal string has a single quote in it.

Double quote strings requires more use of escaped characters. All escape characters will be evaluated in a string enclosed in double quotes. Variables in the strings will be evaluated. For example, the variable $type in this statement

echo "This $varAlpha will be displayed" ;

PHP is made up of simple and complex strings. A simple string is any string with only text or text and variables. The simple string is enclosed in double quota quotation marks. The PHP interpreter will automatically evaluate a variable in a string when it encounters a dollar sign.

Sample Code:

$book = **"Green Eggs and Ham"**;  
**echo "<p>Do you have any** $book**?</p>"**;

//Output: <p>Do you have any Green Eggs and Ham?</p>

The complex string syntax is a string that uses the curly brackets { } for use of complex expressions, within the string.

**<?php**$aVar = **'astring'**;  
  
*// Incorrect Output: This is { astring}***echo "This is {** $aVar**}"**;  
  
*// Correct Output: This is astring***echo "This is** {$aVar}**"**;  
**echo "This is** ${aVar}**"**;  
**?>**

# PHP Built in String Functions

PHP provides a wide range of built in functions to alter, compare and evaluated strings. We will now cover some of the more useful PHP string functions.

# Count Characters

First knowing the length of a strings is often required before performing any other functions on the string. For example, testing user input length for an account username or check correct number of integers entered for a Social Security Number or credit card number. There are two built in functions that will return the number of characters in a string, strlen() and str\_word\_count(). The function strlen() will return the number of characters in a string. The function str\_word\_count() will return the number words in the string.

**<?php***//strlen() - count characters in a string***echo** *strlen*(**"Hello"**); *//Output 5*$str = **' ab cd '**;  
**echo** *strlen*($str); *//Output 7***?>  
  
  
<?php***//str\_word\_count() – count words in a string*$str = **"Hello fri3nd, you're  
 looking good today!"**;  
  
**echo** *str\_word\_count*($str); *//Output 7***?>**

# Numeric Strings

Numbers can be stored in variable as strings. There are special cases where a number should be stored as a string. For examples, social security numbers or a mailing addresses. Number sequences can be stored as strings by surrounding the number with either single or double quotation marks. To test if a string only contain numbers, the built in PHP function is\_numeric() can be used.

<?***php***$tests = **array**(  
 **"42"**,  
 1337,  
 1337e0,  
 **"not numeric"**,  
 **array**(),  
 9.1,  
 **null**);  
  
**foreach** ($tests **as** $element) {  
 **if** (*is\_numeric*($element)) {  
 **echo** *var\_export*($element, **true**) . **" is numeric"**, ***PHP\_EOL***;  
 } **else** {  
 **echo** *var\_export*($element, **true**) . **" is NOT numeric"**, ***PHP\_EOL***;  
 }  
}  
**?>**

**Output:**

'42' is numeric

1337 is numeric

1337 is numeric

'not numeric' is NOT numeric

array (

) is NOT numeric

9.0999999999999996 is numeric

NULL is NOT numeric

# Upper/Lowers Case Strings

When working with strings, there will be instances where the developer will require all the letters in the strings to be lowercase or all the letters in the strings to be uppercase. A common example of this is when users are required to complete an online form which includes a US State textbox. Some users might use all lowercase for the state abbreviation, some user might use any mix of upper and lower case. But to maintain consistent data in the company database, the developer wants all the letters to be uppercase. By using the PHP function strtoupper($string), the user input can be converted to all uppercase strings. Similar the function strtolower() will converts all of the letters in a string to lowercase.

**<?php**$str = **"Mary Had A Little Lamb and She LOVED It So"**;  
$str = *strtoupper*($str);  
**echo** $str; *// Prints MARY HAD A LITTLE LAMB AND SHE LOVED IT SO***?>**

# Sentences

When you are working with strings that are long sentences of a spoken language, it is common to capitalize the first letter in the sentence. PHP has a built in function that will perform that exact task. The function ucfirst(), will make the first letter in the string to uppercase.

<?php  
echo ucfirst("hello world!"); //Output: Hello world!  
?>

Other similar functions:

lcfirst() - converts the first character of a string to lowercase

ucwords() - converts the first character of each word in a string to uppercase

strtoupper() - converts a string to uppercase

strtolower() - converts a string to lowercase

# Encoding and Decoding a String

Certain characters in HTML have a special meaning. To be able to display these characters on a webpage, the characters must be encoded using HTML character entities. The characters that have special meaning in HTML are: the ampersand (&), double quotation mark ("), single quotation mark ('), left angle bracket or “less than” symbol (<), and right angle bracket or “greater than” symbol (>). The function htmlspecialchars() will convert any instance of these special characters to HTML character entities when a string is passed as an argument to this function.

& &amp;

" &quot;

' &#039;

< &lt;

> &gt;

# Trim Strings

To remove any white spaces either at the beginner of ending of a string use the PHP trim() function. The trim() function can also remove any predefined characters from the front or end of a string by passing a char\_list argument.

Syntax: trim(string\_name, char\_list)

<?php

$string\_name=' Welcome to My Webpage ';

$new\_string=trim($string\_name);

echo $new\_string;

vardump($new\_string);

?>

# Return Substrings

To return only part of a string, use the function substr(). The substr() function used to cut a part of a string from a string, starting at a specified position.

Syntax: substr(string\_name, start\_pos, length\_to\_cut)

**<?php**$string1=**"Welcome to w3resource.com"**;  
**echo** $string1;  
**echo '<br>'**;  
**echo** *substr*($string1,1);  
**echo '<br>'**;  
**echo** *substr*($string1,1,5);  
**echo '<br>'**;  
**echo** *substr*($string1,0,10);  
**echo '<br>'**;  
**echo** *substr*($string1,-1,1);  
**echo '<br>'**;  
  
**?>**

Output:

Welcome to w3resource.com

elcome to w3resource.com

elcom

Welcome to

m

# Shuffle String

The str\_shuffle() function randomly shuffles all the characters of a string.

Syntax: str\_shuffle(string)

<?php  
echo str\_shuffle("Hello World");  
?>

//Output: Wdo llHreol

# Reserves String Order

The strrev() function reverses a string.

Syntax: strrev(string)

<?php

$main\_string='www.awebsite.com';

echo strrev($main\_string);

?>

//Output: moc.etisbewa.www

# Multiple Strings

PHP supports functions for working with a string into substrings, merging strings, and modifying string due to a match with another. In this section of the reading, we will study how to work with multiple stings.

Locating Substrings

The strpos() function finds the position of the first occurrence of a substring in a string. It is a case sensitive search of the substring. The function will return the position of the first occurrence the substring within the string. The function strops() takes two arguments. The first strpos() function argument is the string you want to search, and the second strpos() function argument contains the substring to search for. If the substring is not found, the strpos() function returns a Boolean value of FALSE.

<?php  
$mystring = 'abcdeghijk';  
$findme   = 'e';  
$pos = strpos($mystring, $findme);  
  
// Note our use of ===.  Simply == would not work as expected  
// because the position of 'e' was the 4th character.  
// Positions start at zero  
if ($pos === false) {  
    echo "The string '$findme' was not found in the string '$mystring'";  
} else {  
    echo "The string '$findme' was found in the string '$mystring'";  
    echo " and exists at position $pos";  
}  
?>

//Output: The string 'e' was found in the string 'abcdeghijk' and exists at position 4

# Similar Functions:

strchr() : The strchr() function searches for the first occurrence of a string inside another string.

Syntax: strstr ( string , search string [, bool $before\_needle = FALSE ] )

<?php  
$email  = 'name@example.com';  
$domain = strstr($email, '@');  
echo $domain; // prints @example.com  
  
$user = strstr($email, '@', true); // As of PHP 5.3.0  
echo $user; // prints name  
?>

strrchr() : The strrchr() function finds the position of the last occurrence of a string within another string, and returns all characters from this position to the end of the string.

Syntax: strrchr ( string , search string)

<?php  
echo strrchr("Hello world! What a beautiful day!",What);  
?>

//Output: What a beautiful day!

str\_replace() : The str\_replace() function replaces some characters with some other characters in a string.

This function works by the following rules:

1. If the string to be searched is an array, it returns an array
2. If the string to be searched is an array, find and replace is performed with every array element
3. If both find and replace are arrays, and replace has fewer elements than find, an empty string will be used as replace
4. If find is an array and replace is a string, the replace string will be used for every find value
5. Note: This function is case-sensitive. Use the str\_replace() function to perform a case-insensitive search.

Syntax: str\_replace(find,replace,string,count)

<!DOCTYPE html>  
<html>  
<body>  
  
<?php  
$arr = array("blue","red","green","yellow");  
print\_r(str\_replace("red","pink",$arr,$i));  
echo "<br>" . "Replacements: $i";  
?>  
  
<p>In this example, we search an array to find the value "red", and then we replace the value "red" with "pink".</p>  
  
</body>  
</html>

# Empty String

The PHP empty() function check if a variable is empty or not. It is sometimes necessary to know if a variable empty before performing certain tasks using the variable. For example, an online form that gets processed in PHP can be submitted by the user with empty fields on the form. When the form fields are assigned to variables, it is possible some of the variables will be empty. In these situations, empty function can be extremely useful.

Syntax: empty(var\_name)   
Returns true if var\_name has an empty and zero value.

To test if a variable is not empty use the NOT (!) operator with the empty() function. The not operator can be used to change FALSE to TRUE and TRUE to FALSE. For example, !(FALSE) is TRUE. The statement !(empty(var\_name)) will return TRUE when a variable is not empty. The alternative is to use the isset() function. The isset () function is used to check whether a variable is set or not. The isset() function return false if testing variable contains a NULL value.

Syntax: isset(variable)

Return TRUE if the variable exists and has a value not equal to NULL. FALSE otherwise.

$var = 0;  
  
*// Evaluates to true because $var is empty***if** (**empty**($var)) {  
 **echo '$var is either 0, empty, or not set at all'**;  
}  
  
*// Evaluates as true because $var is set***if** (**isset**($var)) {  
 **echo '$var is set even though it is empty'**;  
}

# Dividing Strings into Smaller Pieces

Data can be stored and transmitted in many formats on the internet. A common format is a simple text file with a common delimiter. To be able to work with the data, the data will have to broken down into subsections. To separate the data, use PHP built in function strtok(). The function takes in a string and the common delimiter as function argument and return the text from the beginning of the string to the first occurrence of the separator.

Syntax: $variable = strtok(string, separators);

To assign the next token to $variable, call the strtok() function again, but only pass to it a single argument containing the separator. PHP engine will keep track of the current token and assigns the next token to $variable, starting at the first character after the separator, each time the strtok() function is called and until the end of the string is reached.

<?php  
$string = "This is\tan example\nstring";  
/\* Use tab and newline as tokenizing characters as well  \*/  
$tok = strtok($string, " \n\t");  
  
while ($tok !== false) {  
    echo "Word=$tok<br />";  
    $tok = strtok(" \n\t");  
}  
?>

# Strings and Arrays

Spiting strings into token one by one us useful if you want to quickly scan through each token. But a more useful and more controllable alternative is splitting a string into an array, where each element in the array is one portion of the string. The function str\_split() or explode() will split a string into an indexed array. The function

Syntax: $array = str\_split(string[, length]);.

<?php  
  
$str = "Hello Friend";  
  
$arr1 = str\_split($str);  
$arr2 = str\_split($str, 3);  
  
print\_r($arr1); //function print\_r() prints human readable data from $arr1  
print\_r($arr2);  
  
?>

//Output:

Array

(

[0] => H

[1] => e

[2] => l

[3] => l

[4] => o

[5] =>

[6] => F

[7] => r

[8] => i

[9] => e

[10] => n

[11] => d

)

Array

(

[0] => Hel

[1] => lo

[2] => Fri

[3] => end

)

The explode() function breaks a string into an array. Returns an array of strings, each of which is a substring of string formed by splitting it on boundaries formed by the string delimiter. The explode function is required and it specific the string delimiter, the sting is the string to be split and limit is an optional argument that specific the number array elements to return.

Syntax: $array = explode(separator, string, limit)

<?php  
$str = "Hello world. It's a beautiful day.";  
print\_r (explode(" ",$str));  
?>

//Output

Array (

[0] => Hello

[1] => world.

[2] => It's

[3] => a

[4] => beautiful

[5] => day.

)

# Comparing Strings Operator

The most important string operation in any programming language is the comparing functionality. The ability to compare two strings against each other and return a Boolean value of true if they match or the Boolean value of false if they do not match. PHP provides two possible ways of comparing strings. The first technique is using the ( == ) equal operating.

**if**(**'string1'** == **'string1'**)  
{  
 **echo 'Strings match.'**;  
} **else** {  
 **echo 'Strings do not match.'**;  
}

//Output: Strings match.

// Notice the S is now capitalized in the second string

**if**(**'string1'** == **'String1'**)  
{  
 **echo 'Strings match.'**;  
} **else** {  
 **echo 'Strings do not match.'**;  
}

//Output: Strings do not match.

# String Comparison Functions

The second way to compare strings is to use of the built in PHP functions. PHP provide a range of compare functions, depending on the relationship between the strings. The two most widely use basic compare functions are strcasecmp() and strcmp(). The both perform a string compare but strcmp() is case sensitive and strcasecmp() is not.

**<?php**$var1 = **"Hello"**;  
$var2 = **"hello"**;  
**if** (*strcmp*($var1, $var2) !== 0) {  
 **echo '$var1 is not equal to $var2 in a case sensitive string comparison'**;  
}

//Output: $var1 is not equal to $var2 in a case sensitive string comparison  
**?>**  
**<?php**$var1 = **"Hello"**;  
$var2 = **"hello"**;  
**if** (*strcasecmp*($var1, $var2) !== 0) {  
 **echo '$var1 is not equal to $var2 in a case sensitive string comparison'**;  
}  
**else** {  
 **echo '$var1 is equal to $var2 in a non-case sensitive string comparison'**;  
}

//Output: $var1 is equal to $var2 in a non-case sensitive string comparison  
**?>**

# Regular Expressions

A regular expression is a method of representing a string matching pattern. Regular expressions enable strings that match a particular pattern within textual data records to be located and modified and they are often used within utility programs and programming languages that manipulate textual data. Regular expressions are extremely powerful. PHP developers commonly use regular expression to validate user input. Such input can include credit card numbers, phone numbers, zip code, etc.

PHP supports Perl Compatible Regular Expressions (PCRE). Some of the PCRE functions include preg\_match(), preg\_match\_al(), preg\_replace(), preg\_split,

The function preg\_match() is the most commonly used regular expression function. The function takes a regular expression patter as an argument and then the string to be searched. The function will return a 1 if a match was found and 0 if no match was made. Notice how this code is case insensitive.

$String = **"welcome to los angeles"**;  
**if** (*preg\_match*(**"/Los Angeles/"**, $String))  
 **echo "<p>Match found</p>"**;  
**else  
 echo "<p>No match</p>"**;  
  
**//Output: <p>Match Found</p>**

# Writing Regular Expression Patterns

A regular expression is a pattern that is matched against a string from left to right. Most characters stand for themselves in a pattern, and match the corresponding characters in the search string.

When using the PCRE functions, it is required that the pattern is enclosed by delimiters. A delimiter can be any non-alphanumeric, non-backslash, non-whitespace character. Commonly used delimiters are forward slashes (/), hash signs (#) and tildes (~).

Examples:

/abc def/

#^[^0-9]$#

+aword+

%[a-z\*]%

Bracket style delimiters are also possible, where the opening and closing brackets are the starting and ending delimiter, respectively. (), {}, [] and <> are all valid bracket style delimiter pairs. Example: (here [is] a (regexp))

When the delimiter is part of the string that needs to be match, use the backslash to escape the delimiter special meaning.

/www:\/\//

#ftp://

# Meta-characters ¶

The usefulness of regular expressions is its ability to include different and repetitions in the pattern matching. By using meta-characters, a more advance regular expression pattern can be encoded. Meta-characters are interpreted in some special way depending on the text directly in front of or in back of the meta character.

There are two different types of meta-characters. The first type is matches anywhere in the pattern, except in the square brackets. The second type are recognized in square brackets.

|  |  |
| --- | --- |
| Meta-character | Description |
| \ | general escape character with several uses |
| ^ | assert start of subject (or line, in multiline mode) |
| $ | assert end of subject or before a terminating newline (or end of line, in multiline mode) |
| . | match any character except newline (by default) |
| [ | start character class definition |
| ] | end character class definition |
| | | start of alternative branch |
| ( | start subpattern |
| ) | end subpattern |
| ? | extends the meaning of (, also 0 or 1 quantifier, also makes greedy quantifiers lazy (see [repetition](http://php.net/manual/en/regexp.reference.repetition.php)) |
| \* | 0 or more quantifier |
| + | 1 or more quantifier |
| { | start min/max quantifier |
| } | end min/max quantifier |

# Matching Any Character

Use the Meta Character period ( . ) to match any single character. For example, a developer has a string and needs to know if it is 5 characters long. Because the variable only contains 4 characters, the preg\_match() function returns a value of 0.

$ZIPCODE = "915";

preg\_match("/...../", $ZIPCODE); // returns 0

This example has a variable with five characters. The preg\_match will match the regular expression /…../ with the variable with five characters. One characters per period is matched.

$ ZIPCODE = "90045";

preg\_match("/...../", $ ZIPCODE); // returns 1

To take it a step further and required the $ZIPCODE variable to start with a 9 for California zip codes, replace the first period in the regular express with a 9. It is now requiring that the variable $ZIPCODE start with a 9 for the regular expression to match and return 1.

$ ZIPCODE = "90045";

preg\_match("/9..../", $ ZIPCODE); // returns 1

# Matching Characters at the Beginning or End of a String

The metacharater anchors do not match any character at all. Instead, they match a position before, after, or between characters. They can be used to "anchor" the regex match at a certain position. The caret ^ matches the position before the first character in the string. Applying ^a to abc matches a because a is the first letter in the match and ^ is stating the starting position of the string. ^b does not match abc at all, because the b cannot be matched right after the start of the string which matched by ^.

This example specifies that the $URL variable starts with http. The regular express /http/ requires that http appear anywhere in $URL but by adding the ^ to the regular expression /^http/, it now requires that http appear the beginning the string.

$URL = **"http://www.awebsite.com"**;  
$result = *preg\_match*(**"/^http/"**, $URL);  *print\_r*($result); *// returns 1*

By using the dollar symbol ( $ ), a regular expression must match the last characters in the string. c$ matches c in abc, while a$ does not match because a is not the last character.

Here is an example where the code requires that the string ends with .gov domain suffix. This code will return a value of 1, because $URL does end with a .gov domain.

$URL = **"http://www.irs.gov"**;  
$result = *preg\_match*(**"/.gov$/"**, $URL);   
*print\_r*($result); *// returns 1*

----

# Matching Metacharacters

When there is a need to match one of the characters that function as a regular expression meta character, use the escape character ( \ ) to flag the PHP interpreter to not treat the character that follows as a meta character but as a normal character. For example, if a regular expression need to check for a period to appear in a URL, then the period must first be escaped. /\.gov/

$URL = **"http://www.irs-gov"**;

*// returns 1 because . was not escaped*  
**echo** *preg\_match*(**"/.gov$/"**, $URL);

$URL = **"http://www.irs-gov"**;

*// returns 0 because now \. must represent a .*  
**echo** *preg\_match*(**"/**\.**gov$/"**, $URL);

To print the dollar symbol in a string in PHP is slightly tricky. The $ symbol represent a variable in PHP. To escape the dollar symbol, there are two options. The first is to used double around the strings and then use three backslashes to (\\\$) in the string. The second method is to use single quotes around the string and use only one escape backslash (\$).

$dollarAmt="$123.45";

echo preg\_match('/^\$/', $dollarAmt); // returns 1

echo preg\_match("/^\\\$/", $ dollarAmt); // returns 1

# Repetition

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

* ? //Preceding character is optional
* + //Specifies that preceding character must match
* \* //Zero or more must match
* {n} //The number of matches required

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

Example: z{2,4} matches "zz", "zzz", or "zzzz".

A closing brace on its own is not a special character. If the second number is omitted, but the comma is present, there is no upper limit; if the second number and the comma are both omitted, the quantifier specifies an exact number of required matches. Thus [aeiou]{3,} matches at least 3 successive vowels.

There is also the use of a question mark quantifier that represents an optional pattern directly in front of the question mark. For example, in the below example the s in https is an optional character. Both http and https will match with the use of this regular express.

$URL = "http://www.whitehouse.gov";

preg\_match("/^https?/", $URL); // returns 1

The addition quantifier (+) specifies that one or more sequential occurrences of the preceding characters match, whereas the asterisk quantifier (\*) specifies that zero or more sequential occurrences of the preceding characters match. As a simple example, the following code demonstrates how to ensure that data has been entered in a required field.

$Name = "John";

preg\_match("/.+/", $Name); // returns 1

Similarly, because a numeric string might contain leading zeroes, the following code demonstrates how to check whether the $NumberString variable contains zero or more leading zeroes:

$NumberString = "04544";

preg\_match("/^0\*/", $NumberString); // returns 1

The { } quantifiers allow you to more precisely specify the number of times that a character must repeat sequentially. The following code shows a simple example of how to use the { } quantifiers to ensure that a ZIP code consists of at least five characters:

preg\_match("/ZIP: .{5}$/", " ZIP: 01562"); // returns 1

The preceding code uses the period metacharacter and the { } quantifiers to ensure that the $ZIP variable contains a minimum of five characters. The following code specifies that the $ZIP variable must consist of at least five characters but a maximum of 10 characters, in case the ZIP code contains the dash and four additional numbers that are found in a ZIP+4 number:

preg\_match("/(ZIP: .{5,10})$/", "ZIP: 01562-2607"); // returns 1

# Specifying Subexpressions

As you learned earlier, regular expression patterns can include literal values; any strings you validate against a regular expression must contain exact matches for the literal values contained in the pattern. You can also use parentheses metacharacters (( and )) to specify the characters required in a pattern match. Characters contained in a set of parentheses within a regular expression are referred to as a subexpression or subpattern. Subexpressions allow you to determine the format and quantities of the enclosed characters as a group. As an example, consider the following pattern, which defines a regular expression for a telephone number: "/^(1 )?(\(.{3}\) )?(.{3})(\-.{4})$/"

The first and second groups in the preceding pattern include the ? quantifier. This allows a string to optionally include a 1 and the area code. If the string does include these groups, they must be in the exact format of “1 ” for the first pattern and “(nnn) ” for the second pattern, including the space following the area code. Similarly, the telephone number itself includes two groups that require the number to be in the format of “nnn” and “–nnnn.” Because the “1 ” and the area code pattern are optional, all of the following statements return a value of 1:

preg\_match("/^(1 )?(\(.{3}\) )?(.{3})(\-.{4})$/", "555- 1234");

preg\_match("/^(1 )?(\(.{3}\) )?(.{3})(\-.{4})$/", "(707) 555-1234");

preg\_match("/^(1 )?(\(.{3}\) )?(.{3})(\-.{4})$/", "1 (707) 555-1234");