**Chapter 4**

**Strings**

Most data you encounter as you program will be sequences of characters or strings. Strings can hold people’s names, passwords, addresses, credit card numbers, links to photographs, purchase histories and more. For that reason, PHP has an extensive selection of functions for working with strings .\

This chapter shows the many ways to create strings in your programs, including the sometomes tricky subject of interpolation (placing a variable’s value into a string), then covers functions for changing, quoting, manipulating, and searching strings. By the end of this chapter, you’ll be a string-handling expert.

**Quoting String Constants**

There are four ways to write a string literal in your PHP code: using single quotes, double quotes, the here document (heredoc) format derived for the Unix shell, and its “cousin” now document (nowdoc). Theses methods differ in whether they recognize special exceapte sequences that let you encode other character or interpolate variables.

**Variable Interpolation**

When you define a string literal using double quotes or a heredoc, the string is subject to variable interpolation. Interpolation is the process of replacing variable names in the strings with their contained values. There are two ways to interpolate variables into strings.

The simpler of the two ways is to put the variable named in a double-quoted string or in a heredoc:

<?php

        $who = 'Kilroy';

        $where = 'here';

        echo "$who was $where";

        //output Kilroy was here

        ?>

The other way to surround the variable bneing interpolated with curly braces. Using this syntax ensures the correct vbariable is interpolated. The classic uuse of curly braces its to disambiguate the variable named from any surrounding text:

$n = 12;

        echo "You are the {$n}th person";

        //output You are the 12th person

Without the cury braces, PHP would try to print the value of the $nth variable.

Unlike in some shell environments, in PGP, strings are not repeatedly processed for interpolation. Instead, any interpolations in a double-quoted string are processed first and the result is used as the value of the string:

  $bar = 'this is not printed';

        $foo = '$bar'; // single quotes

        print("$foo");

**Single-Quoted Strings**

Single-quoted strings and nowdocs do not interpolate variables. Thus, the variable named in the following string is not expanded because the string literal in which it occurs is single-qouted:

$name = 'Fred';

        $str = 'Hello, $name';// single-quoted

        echo $str;

        //Output Hello, $name

The only escape sequences that work in single-quoted strings are \’, which puts a single quote in a single-quoted string, and \\, which puts a backslash in a single quoted string. Ant other occurrence of a backslash is interpreted simply as a backslash:

echo $name;

        $path = 'C:\\WINDOWS'; // escaped backslash

        echo $nope;

        //Output Luka O' Bostick

        //c:\WINDOWS

**// \n**

**Double-Quoted Strings**

Double-quoted strings interpolate variables and expand the many PHP escape sequences. Table 4-1 lisits the escape sequences recognized by PHP in double-quoted strings.

A table with text and a list of characters

Description automatically generated

If an unknown escape sequence (i.e, a backslash followed by a character that is not one of those in table 4-1) is found in a double-quoted string literal, it is ignored (if you have the warning level E\_Notice set, a warning is generated for such unknown escape sequences):

**Here Documents**

You can easily put multiline strings into your program with a heredoc, as follows:

$clerihew = <<< EndOfQuote

    Sir Humphrey Davy

    Abominated gravy.

    He lived in the odium

    of having discovered sodium.

    EndOfQuote;

    /\*  OutPut

    Sir Humphrey Davy

    Abominated gravy.

    He lived in the odium

    of having discovered sodium.

    \*/

    echo $clerihew;

The <<< identifier token tells the PHP parser that you’re writing a heredoc. You get to pick the identifier (“EndOfQoute” in this case), and you can put it in double quotes if you wish (e.g “EndOfQuote”). The next line starts the text being quoted by the heredoc, which continues until it reaches a line containing only the identifier. To ensure the quoted text is displayed in the output area exactly as you’ve laid it out, turn on plain-text mode by adding this command at the top of your code file:

header('Content-Type: text/plain;');

Alternately, if you have control of your server settings, you could set default\_mime type to plain in the php.ini file:

default\_mimetype = "text/plain"

This is not recommended, however, as it puts all output from the server in plain-text mode, which would affect the layout of most of your webcode.

If you do not set plain-text mode for your heredoc, the default is typically HTML mode, which simply displayed the output all on one line.

When using a heredoc for a simple expression, you can put a semicolon after the terminating identifier to end the statement (as shown in the first example). If you are using a heredoc in a more complex expression, however, you’ll need to continue the expression on the next line, as shown here:

printf(<<< Template %s is %d years old. Template, "Fred", 35);

Single and double quotes in a heredoc are preserved:

$dialogue = <<< NoMore

    "It's not going to happen!" she fumed.

    He raised an eyebrow. "Want to bet?"

    NoMore;

    echo $dialogue;

    //"It's not going to happen!" she fumed.

    //He raised an eyebrow. "Want to bet?"

As it whitespace:

$ws = <<< Enough

    boo

    hoo

    Enough;

    //$ws = "boo\n hoo";

New to PHP 7.3 is the indentation of the heredoc terminator. This allows for more legible formatting in the case of embedded code, as in the following function:

<?php

    function sayIt(){

        $ws = <<< "StufftoSay"

        the quick brown fox

        Jumps over the lazy dog.

        StufftoSay;

        return $ws;

    }

    echo sayIt();

    /\*

    The quick brown fox

    Jumps over the lazy dog.

    \*/

    ?>

The newline before the trailing terminator is removed, so these two assignments are identical:

$s = 'Foo';

    // same as

    $s = <<< EndOfPointlessHeredoc

    foo

    EndOfPointlessHeredoc;

If you want a newline to end your heredoc-quoted string, you’ll need to add one yourself:

$s = <<< End

    foo

    End;

**Printing Strings**

There are four ways to send output to the browser. The scho construct lets you print many values at once, while print() prints only one value. The printf() function builds a formatted string by inserting values into a template. The print\_r() function is useful for debugging; it prints the contents of arrays, objects, and other things in a more or less human-readable form.

**Echo**

To put a string into the HTML of a PHP-generated page, use echo. While it looks and for the most part behaves – like a function, echo is a language construct. This means that you can omit the parentheses, so the following expression are equivalent:

echo "Printy";

       echo("Printy"); //also valid

You can specify multiple items to print by separating them with commas:

echo "First", "second", "third";

       //output Firstsecondthird

It is a parse error to use parentheses when trying to echo multiple values:

// this is a pares error

       echo("Hello", "world");

Because echo is not a true function, you can’t use it as part of a larger expression:

// parse error

       if(echo("test")){

        echo("It worked!");

       }

You can easily remedy such errors by using the print() or printf() functions.

**Print()**

The print() function sends one value(its argument) to the browser:

if(print("test\n")){

                print("It worked!");

            }

            //Output

            //test

            //It worked!

**Printf()**

The printf() function outputs a string built by substituting values into a template (the format string). It is derived from the function of the same name in the standard C library. The first argument to printf() is the format string. The remaining arguments are the values to be substituted. A % character in the format string indicates a substitution.

**Format modifiers**

Each substitution marker in the template consists of a percent sign (%), possible followed by modifiers from the following list, and ends with a type specifier. (Use %% to get a single percent character in the output.) The modifiers must appear in the order in which they are listed here:

1. A padding specifier denoting the character to use to pad the results to the appropriate string size. Specify 0, a space, or any character prefixed with a single quote. Passing with spaces is the default.

2. A sign. This has a different effect on strings than on numbers. For strings, a minus (-) here forces the string to be left-justified (the default is right-justified). For numbers, a plus (+) here forces positive numbers to be printed with a leading plus sign (e.g, 35 will be printed as +35).

3. The minimum number of character that this element should contain. If the result would be less than this number of characters, the sign and passing specifier govern how to pad to this length.

4. For floating-point numbers, a precision specifier consisting of a period and a number; this dictates how many decimal digits will be displayed. For types other than double, this specifier is ignored.

**Type Specifiers**

The type specifier tells printf() what type of data is being substituted. This determines the interpretation of the previously listed modifiers. There are eight types, as listed in Table 4-2.

A screenshot of a computer

Description automatically generated

The printf() function looks outrageously complex to people who aren’t C programmers. Once you get used to it, you’ll find it a powerful formatting tool. Here are some examples.

* A floating-point number to two decimal places:

printf('%.2f', 27.452);

            //27.45

* Decimal and hexadecimal output:

printf('The hex value of %d is %x', 214,214);

            // The hex value of 214 is d6

* Padding an integer to three decimal places:

printf('Bond. James Bond. %03d.',7);

            //Bond. James Bond. 007.

* Formatting a date:

printf('%02d/%02d/%04d', $month, $day, $year);

            // 7/25/2023

* A percentage:

printf('%.2f%% Complete',2.1);

            // 2.10% Complete

* Padding a floating-point number:

printf('You\'ve spent $%5.2f so far',4.1);

            //You've spend $ 4.10 so far

The sprintf() function takes the same arguments as printf() but returns the built-up string instead of printing it. This lets you save the string in a variable for later use:

$date = sprintf("%02d/%02d/%04d", $month, $day, $year);

            // now we can interpolate $date wherever we need a date

**print\_r() and var\_dump()**

The print\_r() function intelligently displays what is passed to it, rather than casting everything to a string, as echo and print() do. Strings and numbers are simply printed. Arrays appear as parenthesized lists of keys and values, prefaced by Array:

$a = array('name' => 'Fred', 'age' => 35, 'wife'=> 'Wilma');

        print\_r($a);

        /\*

        Array(

            [name]=> Fred

            [age] => 35

            [wife]=> Wilma)\*/

Using print\_r() on an array moves the internal iterator to the position of the last element in the array. See Chapter 5 for more on iterators and arrays. When you print\_r() an object, you see the word Object, followed by the initialized properties of the object displayed as an array:

class p{

                var $name = 'nat';

                // ...

            }

            $p = new p;

            print\_r($p);

            /\*\*

             \*  Object ([name] => nat)

             \*

             \*/

Boolean values and Null are not meaningfully displayed by print\_r():

  print\_r(true); // prints "1"

             print\_r(true); // prints ""

             print\_r(true); // prints ""

For this reason, var\_dump() is preferred over print\_r() for debugging. The var\_dump() function displays any PHP values in a human-readable format:

var\_dump(true); //1

             var\_dump(false);//2

             var\_dump(null);//3

             var\_dump(array('name' => "Fred", 'age' => 35));//4

             var\_dump($p);//5

             /\*\* bool (true) 1

              \* bool (false) 2

              \* bool (null)  3

              \* array(2){    4

              \* ["name"]=> string(4) "Fred"

              \* ["age]=> int(35)

              \* }

              \*

              \*object(p)(1){ 5

              \* ["name"]=>

              \* string(3) "Nat"

              \*  }\*/

**Accessing Individual Characters**

The strlen() function returns the number of characters in a string:

$string = 'Hello, world';

        $length = strlen($string); //$length is 12

You can use the string offset syntax on a string to address individual characters:

 $string = 'Hello';

        for ($i = 0; $i < strlen($string); $i++) {

        printf("the %dth character is %s\n", $i, $string[$i]);

            }

**Cleaning Strings**

Often, the strings we get from files or uses need to be cleaned up before we can use them. Two common problems with raw data are the presence of extraneous whitespace and incorrect capitalization (uppercase versus lowercase).

**Removing Whitespace**

You can remove leading or trailing whitespace with the trim(), ltrim(), and rtrim() functions:

$trimmed = trim(string [, charlist]);

        $trimmed = ltrim(string [, charlist]);

        $trimmed = rtrim(string [, charlist]);

trim() returns a copy og string with whitespace removed from the beginning and the end. ltrim() (the l is for left) does the same, but removes whitespace only from the start of the string. rtrim() (the r is for right) removes whitespace only from the end of the string. The optional charlist argument is a string that specifies all the characters to strip. The default characters to strip are given in Table 4-3.

A screenshot of a computer

Description automatically generated

For example:

$title = "Programming PHP \n";

        $str1 = ltrim($title); // $str1 is "Programming PHP \n"

        $str2 = rtrim($title); // $str2 is " Programming PHP"

        $str2 = trim($title); // $str2 is "Programming PHP"

Given a line of tab-separated data, use the charlist argument to remove leading or trailing whitespace without deleting the tabs:

$record = " Fred\tFlinstone\t35\tWilma\t \n";

        $record = trim($record, "\r\n\0\x0B");

        // $record is "Fred\tFlintstone\t35\tWilma"

**Changing Case**

PHP has several functions for changing the case of strings: strtolower() and strtoupper() operate on entire strings, ucfirst() operates only on the first character of the string, and ucwords() operates on the first character of each word in the string. Each function takes a string to operate on as an argument and returns a copy of that string, appropriately changed. For example:

$string1 = "FRED flintstone";

        $string2 = "barney rubble";

        print(strtolower($string1));

        print(strtoupper($string1));

        print(ucfirst($string2));

        print(ucwords($string2));

        /\*\*Output

         \* fred flintstone

         \* FRED FLINTSTONE

         \* Barney rubble

         \* Barney Rubble

         \*/

If you’ve got a mixed-case string that you want to convert to “title case”, where the first letter of each word is in uppercase and the rest of the letters are in lowercase (and you’re not sure what case the string is in to begin with), use a combination of strtolower() and ucwords():

print(ucwords(strtolower($string1)));

         //Fred Flintstone

**Encoding and Escaping**

Because PHP programs often interact with HTML pages, web addresses (URLs), and databases, there are functions to help you work with those types of data. HTML, web addresses, and database commands are all strings, but they each require different characters to be escaped in different ways. For instance, a space in a web address must be written as %20, while a literal less-than sign (<) in an HTML document must be written as &lt;. PHP has a number of built-in functions to convert to and from these encodings.

**HTML**

Special characters in HTML are represented by entities such as &amp; (&) and &lt; (<). There are two PHP functions that turn special characters in a string into their entities: one for removing HTML tags, and one for extracting only meta tags.

**Entity-quoting all special characters**

The htmlentities() function changes all characters with HTML entity equivalents into those equivalents (with the exception of the space character). This includes the less-than sign (<), the greater-than sign (>), the ampersand (&), and accented characters. For example:

$string = htmlentities("Einstürzende Neubauten");

        echo $string;

        //Output

        // Einstürzende Neubauten

The entity-escaped version, &uuml; (seen by viewing the source), correctly displays as ü in the rendered web page. As you can see, the space has not been turned into &nbsp;.

The htmlentities() function actually takes up to three arguments:

$output = htmlentities(input,flags,encoding);

The encoding parameter, if given, identifies the character set. The default is “UTF-8.” The flags parameter controls whether single and double quotes, ENT\_QUOTES converts both types of quotes, and ENT\_NOQUOTES converts neither. There is no option to convert only single quotes. For example

 $input = <<< End

        "Stop pulling my hair!" Jane's eyes flashed. <p>

        End;

        $double = htmlentities($input);

        // &quot;Stop pulling my hair!&quot; Jane's eyes flashed.&lt;p&gt;

        $both = htmlentities($input, ENT\_QUOTES);

        // &quot;Stop pulling my hair!&quot; Jane&#039;s eyes flashed.&lt;p&gt;

        $neither = htmlentities($input, ENT\_NOQUOTES);

        //"Stop pulling my hair!" Jane's eyes flashed.&lt;p&gt;

**Entity-quoting only HTML syntax characters**

The htmlspecialchars() function converts the smallest set of entities possible to generate valid HTML. The following entities are converted:

* Ampersands (&) are converted to &amp;
* Double quotes(“) are converted to &quot;
* Single quotes (‘) are converted to &#039; (if ENT\_QUOTES is on, as described for htmlentities())
* Less-than signs (<) are converted to &lt;
* Greater-than signs (>) are converted to &gt;

If you have an application that displays data that a user has entered in a form, you need to run that data through htmlspecialchars() before displaying or saving it. If you don’t, and the user enters a string like “angle < 30” or “sturm & drang”, the browser will think the special characters are HTML, resulting in a garbled page.

Like htmlentities(), htmlspecialchars() can take up to three arguments:

$output = htmlspecialchars(input,[flags, [encoding]]);

The flags and encoding arguments have the same meaning that they do for htmlentities().

There are no functions specifically for converting back from the entities to the original text, because this is rarely needed. There is a relatively simple way to do this, though. Use the get\_html\_translation\_table() function to fetch the translation table used by either of these functions in a given quote style. For example, to get the translation table that htmlentities() uses, do this:

$table = get\_html\_translation\_table(HTML\_ENTITIES);

To get the table for htmlspecialchars() in ENT\_NOQUOTES mode, use:

$table = get\_html\_translation\_table(HTML\_SPECIALCHARS, ENT\_NOQUOTES);

A nice trick is to use this translation table, flip it using array\_flip(), and feed it to strtr() to apply it to a string, thereby effectively doing the reverse of htmlentities():

$str = htmlentities("Einstürzende Neubauten");// now it is encoded

        $table = get\_html\_translation\_table(HTML\_ENTITIES);

        $revTrans = array\_flip($table);

        echo strtr($str, $revTrans);

        //OUTPUT: Einstürzende Neubauten

You can, of course, also fetch the translation table, add whatever other translations you want to it, and then do the strtr(). For example, if you wanted htmlentities() to also encode each space to &nbsp;. You would do:

$table = get\_html\_translation\_table(HTML\_ENTITIES);

        $table[' '] = '&nbsp;';

        $encoded = strtr($original, $table);

**Removing HTML tags**

The strip\_tags() function removes HTML tags from a string:

$input = '<p>Howdy, &quot;Cowboy&quot;</p>';

            $output = strip\_tags($input);

            //$output is 'Howdy, &quot;Cowboy&quot;'

The function may take a second argument that specifies a string of tags to leave in the string. List only the opening forms of the tags. The closing forms of tags listed in the second parameter are also preserved:

$input = 'The <b>bold</b> tags will <i>stay</i><p>';

            $output = strip\_tags($input, '<b>');

            // $output is 'The <b>bold</b> tags will stay'

Attributes in preserved tags are not changed by strip\_tags(). Because attributes such as style and onmouseover can affect the look and behavior of web pages, preserving some tags with strip\_tags() won’t necessarily remove the potential for abuse.

**Extracting meta tags**

The get\_meta\_tags() function returns an array of the meta tags for an HTML page, specified as a local filename or URL. The name of meta tag (keywords, author, description, etc) becomes the key in the array and the content of the meta tag becomes the corresponding value:

$metaTags = get\_meta\_tags('http://www.example.com/');

        echo "Web page made by {$metaTags['author']}";

        //output Web page made by John Doe

The general form of the function is:

$array = get\_meta\_tags(filename [, use\_include\_path]);

Pass a true value for use\_include\_path to let PHP attempt to open the file using the standard include path.

**URLs**

PHP rovides functions to convert to and from URL encoding, which allows you to build and decode URLs. There are actually two types of URL encoding, which differ in how they treat spaces. The first (specified by RFC 3986) treats a space as just another illegal character in a URL and encodes it as %20. The second (implementing the application/x-www-form-urlencoded system) encodes a space as a + and it used in building query strings.

Note that you don’t want to use these functions on a complete URL, such as <http://www.example.com/hello>, as they will escape the colons and slashes to produce

http%3A%2F%2Fwww.example.com%2Fhello

Endode only partial URLs (the bit after <http://www/example.com/>) and add the protocol and domain name later.

**RFC 3986 encoding and decoding**

To encode a string according to the URL conventions, use rawurlencode();

$output = rawurlencode(input);

This function takes a string and returns a copy with illegal URL characters encoded in the %dd conventation.

If you are dynamically generating hypertext references for links in a page, you need to convert them with rawurlencode();

$name = "Programming PHP";

        $output = rawurlencode($name);

        echo "http://localhost{$output}";

        //Output http://localhost/Programming%20PHP

The rawurldecode() function decodes URL-encoded strings:

$encoded = 'Programming%20PHP';

        echo rawurldecode($encoded);

        //Output Programming PHP

**Query-string encoding**

The urlencode() and urldecode() functions differ from their raw counterparts only in that they encode spaces as plus signs(+) instead of as the sequence %20. This is the format for builfing query strings and cookie values. These functions can be useful in supplying form-like URLs in the HTML. PHP automatically decodes query strings and cookie values, so you don’t need to use these functions to process those values. The functions are useful for generating query strings:

$baseUrl = 'http://www.google.com/q=';

        $query = 'PHP sessions -cookies';

        $url = $baseUrl . urlencode($query);

        echo$url;

        //Output: http://www.google.com/q=PHP+sessions+-cookies

**SQL**

Most database systems require that string literals in your SQL queries be escaped. SQL’s encoding scheme is pretty simple-single quotes, double quotes, NUL-bytes, and backslashes need to be preceded by a backslash. The addslashes() function adds these slashes, and the stripslashes() function removes them:

$string = <<< EOF

        "It's never going to work," she cried,

        as she hit the backslash (\) key.

        EOF;

        $string = addslashes($string);

        echo $string;

        echo stripslashes($string);

        /\*\* Output

         \* \"It\'s never going to work,\" she cried,

         \* as she hit the backslash (\\) key.

         \* "It's never going to work," she cried, as

         \*  she hit the backslash (\)key.

         \*/

**C-String Encoding**

The addslashes() function escapes arbitrary characters by placing backslashes before them. With the exception of the characters in Table4-4, characters with ASCII values less than 32 or above 126 are encoded with their octal values (e.g., “\002”). The addcslashes() and stripceslashes() functions are used with nonstandard database systems that have their own ideas of which characters need to be escaped.

A white background with black text

Description automatically generated

Call addcslashes() with two arguments – the string to encode and the characters to escape:

$escaped = addcslashes(string,charset);

Specify a range of characters to escape with “..” construct:

echo addcslashes("hello\tworld\n", "\x00..\x1fz..\xff");

        //Output: hello\tworld\n

Beware of specifying ‘0’, ‘a’, ‘b’, ‘f’, ‘n’, ‘r’, ‘t’, or ‘v’ in the character set, as they will be turned into ‘\0’, ‘\a’, and so on. These escapes are recognized by C and PHP and may cause confusion.

Stripcslashes() takes a string and returns a copy with the escapes expanded:

$string = stripcslashes(escaped);

For example:

$string = stripcslashes('hello\tworld\n');

        //Output $string is "hello\tworld\n"

**Comparing Strings**

PHP has two operators and six functions for comparing strings to each other.

**Exact Comparisons**

You can compare two strings for equality with the == and === operators. These operators differ in how they deal with nonstring operands. The == operantor casts string operands to numbers, so it reports that 3 and “3” are equal. Due to the rules for casting strings to numbers, it would also report that 3 and “3b” are equal, as only the portion of the string up to a non-number character is used in casting. The === operator does not cast, and returns false if the data types of the argument differ:

$o1 = 3;

        $o2 = "3";

        if($o1 == $o2){

            echo("== returns true<br>");

        }

        if($o1 === $o2){

            echo("=== returns true<br>");

        }

        // == returns true

The comparison operators (<,<=,>,>=) also work on strings:

$him = "Fred";

        $her = "Wilma";

        if($him < $her){

            print "{$him} comes before {$her} in the alphabet. \n";

        }

        //Output Fred comes before Wilma in the alphabet

However, the comparison operators give unexpected results when comparing strings and numbers:

$string = "PHP Rocks";

        $number = 5;

        if($string < $number){

            echo("{$string} < {$number}");

        }

        //Output PHP Rocks < 5

When one argument to a comparison operator is a number, the other argument is cast to a number. This means that “PHP Rocks” is cast to a number, given 0 (Since the string does not start with a number). Because 0 is less than 5, PHP prints “PHP Rocks < 5”.

To explicitly compare two strings as strings, casting numbers to strings if necessary, use the strcmp() function:

$relationship = strcmp(string\_1, string\_2);

The function returns a number less than 0 if string\_1 sorts before string\_2, greater than 0 if string\_2 sorts before string\_1, or 0 if they are the same:

$n = strcmp("PHP Rocks", 5);

        echo($n);

        //output: 1

A variation on strcmp() is strcasecmp(), which converts strings to lowercase before comparing them. Its arguments and return values are the same as those for strcmp();

$n = strcasecmp("Fred", "frED"); // $n is 0

Another variation on string comparison is to compare only the first few characters of the string. The strncmp() and strncasecmp() functions take an additional argument, the initial number of characters to use for the comparisons:

$relationship = strncmp(string\_1, string\_2, len);

        $relationship = strncasecmp(string\_1, string\_2, len);

The final variation on these functions is natural-order comparison with strnatcmp() and strnatcasesmp(), which take the same arguments as strcmp() and returns the same kind of values. Natural-order comparison identifies numeric portions of the strings being compared and sorts the string parts separately from the numeric parts.

Table 4-5 shows strings in natural order and ASCII order:

A screenshot of a computer

Description automatically generated

**Approximate Equality**

PHP provides several functions that let you test whether two strings are approximately equal—soundex(), metaphone(), similar\_text(), and levenshtein():

$soundexCode = soundex($string);

        $metaphoneCode = metaphone($string);

        $inCommon = similar\_text($string\_1, $string\_2 [, $percentage]);

        $similarity = levenshtenin($string\_1, $string\_2);

        $similarity = levenshtein($string\_1, $string\_2 [, $cost\_ins, $cost\_rep, $cost\_del]);

The Soundex and Metaphone algorithms each yield a string that represents roughly how a word is pronounced in English. To see whether two strings are approximately equal with these algorithms, compare their pronunciations. You can compare Soundex values only to Soundex values and Metaphone values only to Metaphone values. The Metaphone algorithm is generally more accurate, as the following example demonstrates:

$known = "Fred";

        $query = "Phred";

        if(soundex($known) == soundex($query))

        {

            print "soundex: {$known} sounds like {$query} <br>";

        }

        else{

            print "soundex: {$known} doesn't sound like {$query}<br>";

        }

        if(metaphone($known) == metaphone($query))

        {

            print "metaphone: {$known} sounds like {$query}<br>";

        }else {

            print"metaphone: {$known} doesn't sound like {$query}<br>";

        }

        //OutPut

        //soundex: Fred doesn't sound like Phred

        //metaphone: Fred sounds like Phred

The similar\_text() function returns the number of characters that its two string arguments have in common. The third argument, if present, it a variable in which to store the commonality as a percentage:

$string1 = "Rasmus Lerdorf";

        $string2 = "Razmus Lehrdorf";

        $common = similar\_text($string1, $string2, $percent);

        printf("They have %d chars in common (%.2f%%).", $common, $percent);

        //Output: They have 13 chars in common (89.66%)

The Levenshtenin alforithm calculates the similarity of two strings based on how many characters you nust add, substitute, or remove to make them the same. For example instance, “cat” and “cot” have a Levenshtein distance of 1, because you need to change only one character (the “a” to an “o”) to make them the same:

$similarity = levenshtein("cat", "cot");// $similarity is 1

This measure of similarity is generally quicker to calculate than that used by the similar\_text() function. Optionally, you can pass three values to the levenshtein() function to individually weight insertions, deletions, and replacements – for instance, to compare a word against a contraction.

This example excessively weights insertions when comparing a string against its possible contraction, because contractions should never insert characters:

echo levenshtein('would not', 'wouldn\'t', 500,1,1);

**Manipulating and Searching Strings**

PHP has many functions to work with strings. The most commonly used functions for searching and modifying strings are those that use regular expressions to describe the string in question. The functions described in this section do not use regular expressions—they are faster than regular expressions, but they work when you’re looking for a fixed string (for instance if you’re looking for “12/11/01” rather than “any numbers separated by slashes”).

**Substrings**

If you know where the data that you are interested in lies in a larger string, you can copy it out with substr() function:

$piece = substr(string, start[, length]);

The start argument is the position in the string at which to begin copying, with 0 meaning the start ofg the string. The length argument is the number of characters to copy (the default is to copy until the end of the string). For example:

$name = "Fred Flintstone";

        $fluff = substr($name,6,4); //$fluff is "lint"

        $sound = substr($name, 11); //$sound is "tone"

To learn how many time a smaller string occurs within a larger one, use substr\_count();

$number = substr\_count(big\_string, small\_string);

For example:

$sketch = <<< EndOfSketch

        Well, there's egg and bacon; egg sausage and bacon; egg and spam;

        egg bacon and spam; egg bacon sausage and spam; spam bacon sausage

        and spam; spam egg spam spam bacon and spam; spam sausage spam spam

        bacon spam tomato and spam;

        EndOfSketch;

        $count = substr\_count($sketch, "spam");

        print("The wor;d spam occurs {$count} times.");

        //Output The word spam occurs 14 times.

The substr\_replace() function permits many kinds of string modifications:

$string = substr\_replace(original, new, start [, length]);

The function replaces the part of original indicated by the start (0 means the start of the string) and length values with the string new. If no fourth argument is given, substr\_replace() removes the text from start to the end of the string. For instance:

  $greetings "good morning citizen";

        $farewell = substr\_replace($greetings, "bye", 5,7);

        //$farewell is "good bye citizen"

Use a length of 0 to insert without deleting:

$farewell = substr\_replace($farwell, "kind ", 9, 0);

        // $farewell is "good bye kind citizen"

Use a replacement of “” to delete without inserting:

$farewell= substr\_replace($farewell,"", 8);

        // $farewell is "good bye"

Here’s how you can insert at the beginning of the string:

$farewell = substr\_replace($farewell, "now it's time to say ", 0, 0);

        // $farewell is "Now it's time to say good bye"

A negative value for start indicates the number of characters from the end of the string from which to start the replacement:

$farewell = substr\_replace($farewell, "riddance",-3);

        // $farewell is "now it's time to say good riddance"

A negative length indicates the number of characters from the end of the string at which to stop deleting:

$farewell = substr\_replace($farewell, "", -8, -5);

        // $farewell is "now it's time to say good dance"

**Miscellaneous String Functions**

The strrev() function takes a string and returns a revered copy of it:

$string = strrev(string);

For example:

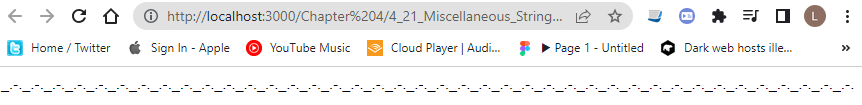
echo strrev("There is no cabal");

        //labac on si erehT

The str\_repear() function takes a string and a count and returns a new string consisting of the argument string repeated count times:

$repeated = str\_repeat(string, count);

For example, to build a crude wavy horizontal rule:



The str\_pad() function pads one string with another. Optionally, you can say what string to pad with, and whether to pad on the left, right, or, both:

$padded = str\_pad(to\_pad, length [, with [, pad\_type]]);

The default is to pad on the right with spaces:

$string = str\_pad('Fred Flintstone', 30);

        echo "{$string}:35:Wilma";

        // Fred Flintstone :35:Wilma

The optional third argument is the string to pad with:

$string = str\_pad('Fred Flintstone', 30, '. ');

        echo "{$string}35";

        //Fred Flintstone. . . . . . . .35

The optional fourth argument can be STR\_PAD\_RIGHT (the default), STR\_PAD\_LEFT, or STR\_PAD\_BOTH (to center). For example:

echo '[' . str\_pad('Fred Flintstone', 30, ' ', STR\_PAD\_LEFT) . "]\n";

echo '[' . str\_pad('Fred Flintstone', 30, ' ', STR\_PAD\_BOTH) . "]\n";

        /\*\*Output

         \* [Fred Flintstone]

         \* [Fred Flintstone]

         \*/

**Decomposing a String**

PHP provides several functions to let you break a string into smaller components. In increasing order of complexity, they are explode(), strtok(), and sscanf().

**Exploding and imploding**

Data often arrives as string, which must be broken down into an array of values. For instance, you might want to split up the comma-separated fields from a string such as “Fred,25,Wilma.” In thses situations, use the explode() function:

$array = explode(separator,string[, limit]);

The first argument, separator, is a string containing the field separator. The second argument, string, is the string to split. The optional third argument, limit, is the maximum number of values to return in the array. If the limit is reached, the last element of the array contains the remainder of the string:

$input = 'Fred,25,Wilma';

        $fields = explode(',', $input);

        // $fields is array('Fred', '25', 'Wilma')

        $fields = explode(',', $input,2);

        // $fields is array('Fred', '25,Wilma')

The implode() function does the exact opposite of explode()—it creates a larger string from an array of smaller strings:

$string = implode(separator, array);

The first argument, separator, is the string to put between the elements of the second argument, array. To reconstruct the simple comma-separated value string, simply say:

$fields = array('Fred','25','Wilma');

$string = implode(',', $fields);// $string is 'Fred,25,Wilma'

The join() function is an alias for implode().

**Tokenizing**

The strtok() function lets you iterate through a stringm getting a new chunk(token) each time. The first time you call it, you need to pass two arguments: the string to iterate over and the token separator. For example:

$firstChunk = strtok(string, separator);

For instance, consider this invocation:

$string = "Fred,Flintstone,35,Wilma";

        $token = strtok($string, ",");

        while($token !== false)

        {

            echo("{$token}<br />");

            $token = strtok(",");

        }

        /\*\* Output

         \* Fred

         \* Flintstone

         \* 35

         \* Wilma

         \*/

The strtok() function returns false when there are no more tokens to be returned. Call strtok() with two arguments to reinitialize the iterator. This restarts the tokenizer from the start of the string.

**sscanf()**

The sscanf() function decomposes a string according to a printf()-like template:

$array = sscanf(string,template);

        $count =sscanf(string, template, var1, ...);

If used without the optional variables, sscanf() returns an array of fields:

$string = "Fred\tFlintstone (35)";

        $a = sscanf($string, "s\t%s (%d)");

        print\_r($a);

        /\*\* Output

         \* Array (

         \* [0] => Fred

         \* [1] => Flintstone

         \* [2] => 35)

         \*/

Pass references to variables to have the fields stored in those variables. The number of fields assigned is returned:

$string = "Fred\tFlintstone (35)";

        $n = sscanf($string, "%s\t%s (%d)", $first, $last, $age);

        echo "Matched {$n} fields: {$first}{$last} is {$age} years old";

        //Output Matched 3 fields: Fred Flintstone is 35 years old

**String-Searching Functions**

Several functions find a string or character within a larger string. They come in three families: strops() and strrpos(), which return a position; strstr(), strchr(), and friends, which return the string they find; and strspn() and strcspn(), which return how much of the start of the string matches a mask.

In all cases, if you specify a number as the “string” to search for, PHP treats that number as the ordinal value of the character to search for. Thus, thses function calls are identical because 44 is ASCII value of the comma:

$pos = strpos($large, ","); // find first comma

$pos = strpos($large, 44);  // also find first comma

All the string-searching functions return false if they can’t find the substring you specified. If the substring occurs at the beginning of the string, the functions return 0. Because false casts to the number 0, always compare the return value with === when testing for failure:

if($pos === false){

                // wasn't found

            }

            else{

                //was found, $pos is offset into string

            }

**Searches Return Position**

The strops() function finds the first occurrence of a small string in a larger string:

$position = strpos(large\_string, small\_string);

If the small string isn’t found, strops() returns false.

The strrpos() function finds the last occurrence of a character in a string. It takes the same arguments and returns the same type of value as strops().

For instance:

$record = "Fred,Flintstone,35,Wilma";

        $pos = strrpos($record, ",");// find last comma

        echo("The last comma in the record is at position {$pos}");

        //The last comma in the record is at position 18

**Searches returning rest of string**

The strstr() function finds the first occurrence of a small string in a larger string and returns from that small string on. For instance:

1 $record = "Fred,Flintstone,35,Wilma";

        $rest = strstr($record, ",");//$rest is ",Flintstone,35,Wilma"

The variations on strstr()are:

* stristr()

Case-insensitive strstr()

* strchr()

Alias for strstr()

* strrchr()

Finds last occurrence of a character in a string

As with strrpos(), strrchr() searches backward in the string, but only for a single character, not for an entire string.

**Searches using masks**

If you thought strrchr() was esoteric, you haven’t seen anything yet. The strspn() and strcspn() functions tell you how many character at the beginning of a string are composed of certain character:

$length = strspn(string, charset);

For example, this function tests whether a string holds an octal number:

 function  isOctal($str){

        return strspn($str,'01234567') == strlen($str);

        }

The c in strcspn() stands for complement—it tells you how much of the start of the string in not composed of the characters in the character set. Use it when the number of interesting characers is greater than the number of uninteresting characters. For example, this function tests whether a string has any NUL-bytes, tabs, or carriage returns:

**Decomposing URLs**

The parse\_url() function returns an array of components of a URL:

$array = parse\_url(url);

For example:

$bits = parse\_url("http://me:secret@example.com/cgi-bin/board?user=fred");

        print\_r($bits);

        /\*\*Array (

         \* [scheme] => http

         \* [host] => example.com

         \* [user] => me

         \* [pass] => secret

         \* [path] => /cgi-bin/board

         \* [query] => user=fred )

         \*/

The possible keys of the hash are scheme, host, port, user, pass, path, query, and fragment.

**Regular Expressions**

If you need more complex searching functionality than the previous methods provide, you can use a regular expression\_\_ a string that represents a pattern, The regular expression functions compare that pattern to another string and see if any of the string matches the pattern. Some function tell you whether there was a match, while other make changes to the string.

There are three uses for regular expressions: matching, which can also be used to extract information from a string; substituting new text for matching text; and splitting a string into an array of smaller chunks. PHP has functions for all. For instance, preg\_match() does a regular expressions match.

Perl has long been condifered the benchmark for powerful regular expressions. PHP uses a C library called pcre to provide almost complete support for Per’s arsebal of regular expressions fearures. Perl regular expressions act on arbitrary binary data, so you can safely match patterns or strings that contain the NUL-byte(\x00).

**The Basics**

Most characters in a regular expression are literal characters, meaning that they match only themselves. For instance, if you search for regular expression “/cow/” in the string “dave as a cowhand”, you get a match because “cow” occurs in that string.

Some characters have special meanings in regular expressions. For instance, a caret (^) at the beginning of a regular expression indicates that it must match the beginning of the string (or, more precisely, anchors the regular expression to the beginning of the string):

preg\_match("/^cow/", "Dave was a cowhand"); // returns false

preg\_match("/^cow/", "cowabunga!");// returns true

Similarly, a dollar sign ($)at the end of a regular expression means that it must match the end of the string(i.e, anchors the regular expression to the end of the string):

preg\_match("/cow$/", "Dave was a cowhand"); // returns false

preg\_match("/cow$/", "Don't have a cow");// returns true

A period(.) in a regular expression matches any single character:

preg\_match("/c.t/", "cat"); // returns true

        preg\_match("/c.t/", "cut"); // returns true

        preg\_match("/c.t/", "c t"); // returns true

        preg\_match("/c.t/", "bar"); // returns false

        preg\_match("/c.t/", "ct");  // returns false

If you want to match one of these special characters (called a metacharacter), you have to escape it with a backslash:

preg\_match("/\$5.00/", "Your bill is $5.00 exactly");// returns true

preg\_match("/$5.00/", "Your bill is $5.00 exactly");// returns false

Regular expressions are case-sensitive by default, so the regular expression “/cow/” doesn’t match the string “COW”. If you want to perform a case-insensitive match, you specify a flag to indicate that (as you’ll see later in this chatper).

So far, we haven’t done anything we couln;t have done with string functions we’ve akready seen, like strstr(). The real power of regular expressions comes from their ability to specify abstract patterns that can match many different character sequences. You can specify three basic types of abstract patterns in a regular expression:

* A set of acceptable characters that can appear in the string (e.g, alphabetic characters, numeric character, specific punctuation characters)
* A set of alternatives for the string (e.g, “com”, “edu”, “net”, or “org”)
* A repeating sequence in the string (e.g, at least one but not more than five numeric characters)

These three kinds of patterns can be combined in countless ways to create regular expressions that match such things as valid phone numbers and URLs.

**Character Classes**

To specify a set of acceptable characters in your pattern, you can wither build a character class yourself or use a predefined one. You can build your own character class by enclosing the acceptable characters in square brackets:

preg\_match("/c[aeiou]t/","I cut my hand");// returns true

        preg\_match("/c[aeiou]t/","This crust cat");//returns true

        preg\_match("/c[aeiou]t/","What cart");//returns false

        preg\_match("/c[aeiou]t/","14ct gold");//returns false

The regular expression engine finds a “c”, then checks that thenext character is one of “a”,”e”,”I”.”o”, or “u”. IF it isn’t a vowel, the match fails and the engine goes back to look for another “c”. IF a vowel is found, the engine checks that the next character is a “t”. If it is, the engine is at the end of the match and returns true. If the next character isn’t a “t”, the engine goes back to look for another “c”.

You can negate a character class with a caret (^) at the start:

  preg\_match("/c[^aeiou]/", "I cut my hand");//return false

        preg\_match("/c[^aeiou]/", "Roboot cthon");//return true

        preg\_match("/c[^aeiou]/", "14ct gold");//return false

In this case, the regular expression engine is looking for a “c” followed by a character that isn’t a vowel, followed by a “t”.

You can define a range of characters with a hyphen (-). This simplifies character classes like “all letters” and “all digits”:

preg\_match("/[0-9]%/", "we are 25% complete");//returns true

preg\_match("/[0123456789]%/", "we are 25% complete");//returns true

preg\_match("/[0-9]%/", "11th");//returns false

preg\_match("/[0-9]%/", "cat");//returns true

preg\_match("/[0-9]%/", "PIT");//returns false

preg\_match("/[0-9]%/", "11!");//returns false

preg\_match("/[0-9]%/", "stop!");//returns true

When you are specifying a character class, some special character lose their meaning, while other take on new meanings. Inparticular, the $ anchor and the period lose their meaning in a character class, while the ^ character is no longer an anchor but negates the character class if it is the first character after the open bracket. FCor instance, [^\]] matches any non closing bracket character, while [$.^] matches any dollar sign, period, or caret.

**Alternatives**

You can use the vertical pipe (|) character to specify alternatives in a regular expression:

preg\_match("/car|dog/", "the cat rubbed my legs");// returns true

preg\_match("/car|dog/", "the dog rubbed my legs");//returns true preg\_match("/car|dog/","the rabbit rubbed my legs");//returns false

The precedence of alternation can be a surprise: “/^car|dog$/” selects from “^cat” and “dog$”, meaning that it matches a line that either starts with “cat” or ends with “dog”. If you want a line that contains just “cat” or “dog”, you need to use the regulare expression “/^(cat|dog)$/”.

You can combine character classes and alternation to, for example, check for string that don’t start with a capital letter:

preg\_match("/^([a-z]|[0-9])/", "the quick brown fox");// returns false

preg\_match("/^([a-z]|[0-9])/", "jumped over");//returns true

preg\_match("/^([a-z]|[0-9])/","10 lazy dogs");//returns true

**Repeating Sequences**

To specify a repeating pattern, you use a quantifier. The quantifier goes after the pattern that’s reeated and says how many times to repeat that pattern. Table 4-6 shows the quantifiers that are supported by PHP’s regular expression.

**A black and white text with black text

Description automatically generated**

To repeat a single character, simply put the quantifier after the character:

preg\_match("/ca+t/". "caaaaaaat");// retuns true

        preg\_match("/ca+t/", "ct");//returns false

        preg\_match("/ca?t/","caaaaaaat");//returns false

        preg\_match("/ca+\*t/","ct");//returns true

When quantifiers and character classes we can actually do something useful, like matching valid US telephone number:

preg\_match("/[0-9]{3}-[0-9]{3}-[0-9]{4}/","303-555-1212");//returns true

preg\_match("/[0-9]{3}-[0-9]{3}-[0-9]{4}/","64-9-555-1234");//returns false

**Subpatterns**

You can use parentheses to group bits of a regular expression together tobe treated as a single unit called a subpattern:

preg\_match("/a (very )+big dog/", "it was a very very big dog"); //returns true

        preg\_match("/^(cat|dog)$/", "cat"); //returns true

        preg\_match("/^(cat|dog)$/", "dog"); //returns true

The parentheses also cause the substring that matches the subpattern to be captured. If you pass an array as the third argument to a match function, the array is populated with any captured substring:

preg\_match("/[0-9]+)/", "you have 42 magic beans", $captured);

//returns true and populates $captured

The zeroth element of the array is set tot the entire string being matched against. The first element is the substring that matched the first subpattern (if there is one), the second element is the substring that matched the second subpattern, and so on.  
**Delimiters**

Pearl-style regular expressions emulate the Perl syntax for patterns, which means that each pattern must be enclosed in a pair of delimiters. Traditionally, the forward slash (/) character is used; for example, /pattern/. However, any nonalphanumeric character other than the backslash character (\) can be used to delimit a Perl-style pattern./ This is useful for matching strings containing slashes, such as filenames. For example the following are equivalent:

 preg\_match("/\/use\/local\//", "/usr/local/bin/perl"); // returns true

 preg\_match("#/usr/local/#", "/usr/local/bin/perl"); // returns true

Parentheses (()), curly braces ({}), square brackets ([]), and angle brackets (<>) can be used as pattern delimiters:

preg\_match("{/usr/local/}", "/usr/local/bin/perl");//returns true

The section “Trailing Options” on page {tbd} discusses the single-character modifiers you can put after the closing delimiter to modify the behavior of the regular expression engine. A very useful one is x, which makes the regular expression before matching. These two patterns are the same, but one is much easier to read:

'/([[:alpha:]]+)\s+\1/'

        '/( # start capture

        [[:alpha:]]+ # a word

        \s+ # whitespace

        \1 # the same word again

        ) # end capture

        /x'

**Match Behavior**

The period(.) matches any character except for a newline (\n). The dollar sign ($) matches at the end of the string or, if the string ends with a newline, just before that newline:

preg\_match("/is (.\*)$/","the key is in my pants", $captured);

        // $captured[1] is 'in mt pants'.

**Character Classes**

As shown in Table 4-7, Perl-compatible regular expressions define a number of named sets of characters that you can use in character classes. The expansions in Table 4-7 are for English. The actial letters varu from, locale to locale.

aEach [: something :] classes can used in place of a character in a character class. For instance, to find any character that’s a digit, an uppercase letter, or an “at” sign (@), use the following regular expression:

 [@[:digit:][:upper:]]

However, you can’t use a character class at the endpoint of a range:

 preg\_match("/[A-[:lower:]]/", "string");// invalid regular expression

Some locales consider certain character sequences as if they were a single character these are called collating sequences. To match one of these multicharacter sequences in a character class, enclose it with [. and .]. For example, if your locale has the collating sequence ch, you can match s, t, or ch with character class:

[st[.ch.]]

The final extension to character classes is the equivalence class, which you specify by enclosing the character within [= and =]. Equivalence classes match characters that have the same collating order, as defined in the current locale. For example, a locale may define a, á, and ä as having the same sorting precedence. TO match any one of them, the equivalence class is [=a=].

**A screen shot of a computer

Description automatically generated**

**Anchors**

An anchor limits a match to a particular location in the string (anchors do not match actual characters in the target string). Table 4=8 lists the anchors supported by regular expressions.

A black and white text box

Description automatically generated

A black and white text

Description automatically generated

A word boundary is defined as the point between a whitespace character and an identifier (alphanumeric or underscore) character:

preg\_match("/[[:<:]]gun[[:>:]]/", "the Burgundy exploded");//returns false

preg\_match("/gun/", "the Burgundy exploded");// returns true

Note that the beginning and end of a string also qualify as word boundaries.

**Quantifiers and Greed**

Regular expression quantifiers are typically greedy. That is, when faced with a quantifier, the engine matches as much as it can while still satisfying the rest of the pattern. For instance:

preg\_match("/(<.\*>)/","do <b>not</b> press the button",$match);

        // $match[1] is '<b>not</b>'

The regular expression matches from the first less-than sign to the last greater-than sign. In effect, the .\* matches everything after the first less-than sign, and the engine backtracks to make it match less and less until finally there’s a greater-than sign to be matched.

This greediness can be a problem. Sometimes you need minimal (nongreedy)

Match-ing this is, quantifiers that match as few times as possible to satisfy the resr of the pattern. Perl provides a parallel set of quantifiers that match minimally. They’re easy to remember because there the same as the greedy quantifiers, but with a question mark(?) Appended. Table 4-9 shows the corresponding greedy and nongreedy quantifiers supported by Perl-style regular expressions.

A screenshot of a computer

Description automatically generated

Here’s how to match a tag using a nongreedy quantifier:

 preg\_match("/(<.\*?>)/","do <b>not</b> press the button",$match);

        // $match[1] is '<b>not</b>'

Another, faster way us to use a character class to match every non-greater-than character up to the next greaser-than sign:

preg\_match("/(<[^>]\*>)/","do <b> not </b> press the button", $match);

        // $match[1] is '<b>'

**Noncapturing Groups**

If you enclose a part of a pattern in parentheses, the text that matches the subpattern is captured and can be accessed later. Sometimes, though, you want to create a subpattern without capturing the matching text. In Perl-compatible regular expressions, you can do this using the (?: subpattern) construct:

preg\_match("/(?:ello)(.\*)/", "jello biafra", $match);

        // $match[1] is " biafra"

**Backreferences**

You can refer to text captured earlier in a pattern with a backreference: \1 refers to the contents of the first subpattern, \2 refers to the second, and so on. If you nest subpatterns, the first begins with the first opening parenthesis, the second begins with the seconds opening parenthesis, and so on.

For instance, this identifiers double words:

preg\_match("/([[:alpha:]]+)\s+\1/","Paris in the the spring", $m);

        // returns true and $m[1] is "the"

The preg\_match() function captures at most 99 subpatterns; subpatterns after the 99th are ignored.

**Trailing Options**

Perl-style regular expressions let you put single-letter options (flags) after the regular expression pattern to modify the interpretation, or behavior, of the match. For instance, to match case-insensitively, simply use the i flag:

 preg\_match("/cat/i","Stop, Catherine!"); // return true

Table 4-10 shows which Perl modifiers are suppored in Perl-compatible regular expressions.

A screenshot of a computer

Description automatically generated

PHP’s Perl-compatible regular expression functions also support other modifiers the aren’t supported by Perl as listed in Table 4-11.

A screenshot of a computer

Description automatically generated

It’s possible to use more than one option in a single pattern, as demonstrated in the following example:

preg\_match("/cat/i","Stop, Catherine!"); // return true

        $message = <<< END

        To: you@youcorp

        From: me@mecorp

        Subject: pay up

        Pay me or else!

        END;

        preg\_match("/^subject: (.\*)/im",$message, $match);

        print\_r($match);

        //Output Array ([0] => SubjectL pay up [1] => pay up)

**Inline Options**

In addition to specifying pattern-wide options after the closing pattern delimiter, you can specify options within a pattern to have them apply only to part of the pattern. The syntax for this is:

(?flags:subpattern)

For example, only the word “PHP” is case-insensitive in this example:

echo preg\_match(‘/I like (?i:PHP)/’, 'I like pHp', $match);

        print\_r($match);

        //returns true (echo: 1)

        //$match[0] is 'I like pHp'

The I,m,s,U,x and X options can be applied internally in this fashion. You can use multiple oprions at once:

preg\_match('/eat (?ix:foo d)/', 'eat FoOD'); // returns true

Prefix an option with a hyphen (-) to turn it off:

echo preg\_match('/I like (?-i:PHP)/', 'I like pHp', $match);

        print\_r($match);

        //returns false (echo: 0)

        // $match[0] is ''

An alternative form enables or disables the flags until the end of the enclosing subpattern or pattern:

preg\_match('/I like (?i)PHP/', 'I like pHp'); // returns true

preg\_match('/I (like (?i)PHP) a lot/', 'I like pHp a lot', $match);

        //$match[1] is 'like pHp'

Inline flags do not enable capturing. You need an additional set of capturing parentheses to do that.

**Lookahead and Lookbehind**

In patterns, it’s sometimes useful to say “match here if this is next.” This is particularly common when you are splitting a string. The regular expression describes the separator, which is not returned. You can use lookahead to ensure (without matching it, thus preventing it from being returned) that there’s more data after the separator. Similarly, lookbehind checks the preceding text.

A black and white text

Description automatically generated

A simple use of negative lookbehind is to extract quoted strings that contain quoted delimiters. For instance, here’s how to extract a single-quoted string (note that the regular expression is commented using the x modifier):

$input = <<< END

        name = 'Tim O\'Reilly';

        END;

        $pattern = <<< END

        ' # opening quote

        ( # begin capturing

         .\*? # the string

         (?<!\\\\) # skip escaped quotes

        ) # end capturing

        ' # closing quote

        END;

        preg\_match("($pattern)x", $input,$match);

        echo $match[1];

        //OUTPUT Tim O\'Reilly

The only tricky part is that to get a pattern that looks behind to see if the last character was a backslash, we need to escape the backslash to prevent the regular expression engine from seeing \), which would mean a literal close parenthesis. In other words, we have to backslash that backslash: \\). But PHP’s string-quoting rules say that \\ produces a literal single backslash, so we end up requiring four backslashes to get one through the regular expression! This is why regular expressions have a reputation for being hard to read.

Perl limits lookbehind to constant-width expression. That is, the expressions cannot contain quantifiers, and if you use alternation, all the choices must be the same length. The Perl-compatible regular expression engine also forbids quantifiers in lookbehind, but does permit alternatives of different length.

**Cut**

The rarely used once-only subpattern, or cut, prevents worst-case behavior by the regular expression engine on some kinds of patterns. The subpattern is never backed out of once matched.

The common use for the once-only subpattern is when you have repeated expression that may itself be repeated:

/(a+|b+)\*\.+/

This code snippet take several seconds to report failure:

$p = '/(a+|b+)\*\.+$/';

        $s = 'abababababbabbbabbaaaaaabbbbabbababababababbba..!';

        if(preg\_match($p, $s)){

            echo "Y";

        }else {

            echo "N";

        }

This is because the regular expression engine tries all the different places to start the match, but has to backtrack out of each one, which takes time. If you know that once something is matched it should never be backed out of, you should mark it with (?>subpattern);

$p = '/(?>a+|b+)\*\.+$/';

The cut never changes the outcome of the match it simply makes it fail faster.

**Conditional Expressions**

A conditional expression is like an if statement in a regular expression. The general form is:

(?(condition)yespattern)

            (?(condition)yespattern|nopattern)

If the assertion can be one of two types: either a backreference, or a lookahead or lookbehind match. To reference a previously matched substring, the assertion is a number from 1 to 99 (the most backreferences available). The condition uses the pattern in the assertion only if the backreference was matched. If the assertion is not a backrederence, it must be a positive or negative lookahead or lookbehind assertion.

**Functions**

There are five classes of functions that work with Perl-compatible regular expressions: matching, replacing, splitting, filtering, and a utility function for quoting text.

**Matching**

The preg\_match() function performs Perl-style pattern matching on a string. It’s the equivalent of the m// operator in Perl. The preg\_match\_all() functions takes the same arguments and gives the same return value as the preg\_match() function, except that it takes a Perl-style pattern instead of a standard pattern:

$found = preg\_match(pattern, string [, captured]);

For example:

preg\_match('/y.\*e$/','Sylvie');// returns true

preg\_match('/y(.\*)e$/', 'Sylvie.$m')// $m is array('ylvie', 'lvi')

While there’s a preg\_match() function to match case-insensitiviely, there;s no preg\_matchi() function. Instead, use the I flag on the pattern:

preg\_match('y.\*e$/i', 'Sylvie',$m);//returns true

The preg\_match\_all() function repeatedly matches from where the last match ended, until no more matches can be made:

$found = preg\_match\_all(pattern, string, matches [, order ])

The order value, either PREG\_PATTERN\_ORDER or PREG\_SET\_ORDER, determines the layout of matches. We’ll look at both, using this code as a guide:

$string = <<< END

        13 dogs

        12 rabbits

        8 cows

        1 goat

        END;

        preg\_match\_all('/(\d+) (/S+)/', $string, $m1, PREG\_PATTERN\_ORDER);

        preg\_match\_all('/(\d+) (/S+)/', $string, $m1, PREG\_SET\_ORDER);

With PREG\_PATTERN\_ORDER (the default), each element of the array corresponds to a particular capturing subpattern. So $m1[0] is an array of all the substrings that matched the pattern, $m1[1] is an array of all the substrings that matched the first subpattern (the numbers), and $m1[2] is an array of all the substrings that matched the second subpattern (the words). The array $m1 has one more elemnt then it has subpatterns.

With PREF\_SET\_ORDER, each element of the array corresponds to the next attempt to match the whole patter. So $m2[0] is an array of the first set of matches (’13 dogs’, ‘13’, ‘dogs’), $m2[1] is an array of the seonds set of matches (’12 rabbits’, ‘12’, ’rabbits’), and so on. The array $m2 has as many elements as there were successful matches of the entire pattern.

Example fetches the HTML at a particular web address into a string and extracts the IR:s from the HTML. For each URL, it generates a link back to the program that will display the IRLs at that address.

    /\*\*Example 4-1 Extracting URLs from an HTML page \*/

    <?php

    $url = '';

    if ($\_SERVER['REQUEST\_METHOD'] === 'POST') {

        // Check if the 'url' key exists in the $\_POST array before accessing it

        if (isset($\_POST['url'])) {

            $url = $\_POST['url'];

        }

    } else {

        // Check if the 'url' key exists in the $\_GET array before accessing it

        if (isset($\_GET['url'])) {

            $url = $\_GET['url'];

        }

    }

?>

    <form action="<?php echo $\_SERVER['PHP\_SELF']; ?> " method="POST">

    <p>URL: <input type="text" name="url" value="<?php echo $url ?>"/><br />

    <input type="submit">

    </form>

    <?php

        if($url){

            $remote = fopen($url, 'r'); {

                $html = fread($remote, 1048576); // read up to 1 MB of HTML

            }

            fclose($remote);

            $urls='(http|https|telnet|gopher|file|wais|ftp)';

            $ltrs ='\w';

            $gunk ='/#~:.?+=&%@!\-';

            $punc='.:?\-';

            $any="{$ltrs}{$gunk}{$punc}";

            preg\_match\_all("{

                \b # start at word boundary

                {$urls}: # need resource and a colon

                [{$any}] +? # followed by one or more of any value

                #characters-but be conservative

                #and take only what you need

                (?= # the match ends at

                [{$punc}]\* # punctuation

                [^{$any}] # followed by a non-URL character

                | # or

                \$ #the end of the string

                )

                }x",$html,$matches);

            printf("I found %d URLs<P>\n", sizeof($matches[0]));

            foreach($matches[0] as $u){

                $link = $\_SERVER['PHP\_SELF']. '?url=' . urlencode($u);

                echo "<a href=\"{$link}\">{$u}</a><br />\n";

            }

        }

    ?>

A screenshot of a computer

Description automatically generated

**Replacing**

The preg\_replace() function behaves like the search-and-replace operation in your text editor. It finds all occurrences of a pattern in a string and changes those occurrences to something else:

$new = preg\_replace(pattern, replacement, subject [, limit ]);

Pass an array of strings as subject to make the substitution on all the them. The new strings are returned from preg\_replace():

names = array('Fred Flintstone',

        'Barney Rubble',

        'Wilma Flintstone',

        'Betty Rubble');

        $tidy = preg\_replace('/(\w)\w\* (/w+)/', '\1 \2', $names);

        // $tidy is array ('F Flintstone', 'B Rubble', 'W Flintstone', 'B Rubble')

To perform multiple substitutions on the same string or array of strings with one call to preg\_replace(), pass arrays of patterns and replacements:

$contractions = array("/don't/i", "/won't/i", "/can't/i");

        $expansions = array('do not', 'will not', 'can not');

        $string = "{;ease don;t yell - I can;t jump while you won't speak";

        $longer = preg\_replace($contractions, $expansions, $string);

        // $longer is 'Please do not yell - I can not j8ump while you will not speak';

If you give fewer replacements than patterns, text matching the extra patterns is deleted. This is a handy way to delete a lot of things at once:

$htmlGunk = array('/<.\*?>/', '/&.\*?;/');

        $html = '&eacute; : <b>very</b> cute';

        $stripped = preg\_replace($htmlGunk, array(), $html);

        // stripped is ' : very cute'

If you give an array of patterns but a single string replacement, the same replacement is used for every pattern:

$stripped = preg\_replace($htmlGunk, '',$html);

The replacement can use backreferences. Unlike backreferences in patterns, though, the preferred syntax for backreferences in replacements is $1,$2,$3, and so on. For example:

echo preg\_replace('/(\w)\w+\s+(\w+)/', '$2, $1', 'Fred Flintstone')

        //output Flintstone, F.

The /e modifier makes preg\_replace() treat the replacement string as PHP code that returns the actual string to use in the replacement. For example, this converts every Celsius temperature to Fahrenheit:

$string = 'It was 5C outside, 20C inside';

        echo preg\_replace('/(\d+)C\b/e', '$1\*9/5+32', $string);

        //OutputIt was 41 outside, 68 inside

This more complex example expands variables in a string:

$name = 'Fred';

        $age = 35;

        $string = '$name is $age';

        preg\_replace('/\$(\w+)/e', '$$1', $string);

Each match isolates the name of a variable ($name, $age). The $1 in the replacement refers to those names, so the PHP code actually executes is $name and $age. That code evaluates to the value of the variable, which is what’s used as the replacement. Whew!

A variation on preg\_replace() is preg\_replace\_callback(). This calls a function to get the replacement string. The function is passed an array of matches (the zeroth element is all the txt that matched the pattern, the first is the contents of the first captured subpattern, and so on). For example:

$string = 'goodbye cruel world';

        $new = preg\_replace\_callback('/\w+/', 'titlecase', $string);

        echo $new;

        //output Goodbye Cruel World

**Splitting**

Whereas you use preg\_match\_all() to extract chunks of a string when you know what those chunks are, use preg\_split() to extract chunks when you know what separates the chunks from each other:

$chunks = preg\_split(pattern,string[, limit [, flags]]);

The pattern matches a separator between two chunck. By default, the separators are not returned. The optional limit specifies the maximum number of chunck to return (-1 is the default, which means all chunks). The flags argument is a bitwise OR combination of the flags PREG\_SPLIT\_NO\_EMPRY (empty chunks are not returned).

For example, to extract just the operands from a simple numeric expression, use:

$ops = preg\_split('{[+\*/-]}', '3+5\*9/2');

        //output: $pos is array ('3','5','9','2')

To extract pattern matches at every boundary between characters in the string, and at the start and end of the string. This lets you split a string into an array of characters:

$array = preg\_split('//', $string);

**Filtering an array with a regular expression**

The preg\_grep() function returns those elements of an array that match a given pattern:

$matching = preg\_grep(pattern, array);

For instance, to get only the filenames that end in .txt, use:

 $textfiles = preg\_grep('/\.txt$/', $filenames);

**Quoting for regular expressions**

The preg\_quote() function creates a regular expression that matches only a given string:

$re =preg\_quote(string [, delimiter]);

Every character in string that has special meaning inside a regular expression (e.g, \* or $) is prefaced with a backslash:

echo preg\_quote('$5.00 (five bucks)');

        //output \$5\.00\(five bucks\)

The oprional second argument is an extra character to be quoted. Usually, pass your regular expression delimiter here:

**Differences from Perl Regular Expressions**

Although very similar, PHP’s implementation of Perl-style regular expressions has a few minor differences from actual Perl regular expressions:

* The NULL character (ASCII 0) is not allowed as a literal character within a pattern string. You can reference it in other ways, however (\000,\x00,etc).
* The \E, \G, \L, \l, \Q, \u, and \U options are not supported.
* The (?{some perl code}) construct is not supported.
* The /D, /G, /U, /u, /A, and /X modifiers are supported
* The vertical tab \v counts as a whitespace character
* Lookahead and lookbehind assertions cannot be repeated using \*, +, or ?.
* Parenthesize submatches within negative assertions are not remembered.
* Alternation branches within a lookbehind assertion can be of different lengths.

**What’s Next**

Now that you know everything there is to know about strings and working with them, thenext major part of PHP we’ll focus on is arrays. These compound data types will challenge you, but you need to get well acquainted with them, as PHP works with them in many areas. Learning how to add array elements, sort arrays, and deal with multidimensional forms of arrays is essential to being a good PHP developer.