Before you continue you should have a basic understanding of the following:

- HTML
- CSS
- JavaScript

**What is PHP?**

- PHP is an acronym for "PHP: Hypertext Preprocessor"
- PHP is a widely-used, open source scripting language
- PHP scripts are executed on the server
- PHP is free to download and use

**PHP is an amazing and popular language!**

It is powerful enough to be at the core of the biggest blogging system on the web (WordPress)!
It is deep enough to run the largest social network (Facebook)!
It is also easy enough to be a beginner's first server side language!

**What is a PHP File?**

- PHP files can contain text, HTML, CSS, JavaScript, and PHP code
- PHP code are executed on the server, and the result is returned to the browser as plain HTML
- PHP files have extension ".php"

---

**What Can PHP Do?**

- PHP can generate dynamic page content
- PHP can create, open, read, write, delete, and close files on the server
- PHP can collect form data
- PHP can send and receive cookies
- PHP can add, delete, modify data in your database
- PHP can be used to control user-access
- PHP can encrypt data

With PHP you are not limited to output HTML. You can output images, PDF files, and even Flash movies. You can also output any text, such as XHTML and XML.

---

**Why PHP?**

- PHP runs on various platforms (Windows, Linux, Unix, Mac OS X, etc.)

- PHP is compatible with almost all servers used today (Apache, IIS, etc.)
- PHP supports a wide range of databases
- PHP is free. Download it from the official PHP resource: www.php.net
- PHP is easy to learn and runs efficiently on the server side

Installation

**To start using PHP, you can:**

- Find a web host with PHP and MySQL support
- Install a web server on your own PC, and then install PHP and MySQL

---

Use a Web Host With PHP Support

If your server has activated support for PHP you do not need to do anything.

Just create some .php files, place them in your web directory, and the server will automatically parse them for you.

You do not need to compile anything or install any extra tools.

Because PHP is free, most web hosts offer PHP support.

---

Set Up PHP on Your Own PC

However, if your server does not support PHP, you must:

- install a web server
- install PHP
- install a database, such as MySQL

The official PHP website (PHP.net) has installation instructions for
PHP: http://php.net/manual/en/install.php

**Syntax**

A PHP script is executed on the server, and the plain HTML result is sent back to the browser.
A PHP script can be placed anywhere in the document.
A PHP script starts with **<?php** and ends with **?>**

A PHP statements end with a semicolon (;).

Basic PHP Syntax

```
<?php
   // PHP code goes here
?>
```

**The default file extension for PHP files is ".php".**

A PHP file normally contains HTML tags, and some PHP scripting code.

Below, we have an example of a simple PHP file, with a PHP script that uses a built-in PHP function "echo" to output the text "Hello World!" on a web page:

Example:

```
<!DOCTYPE html>
<html>
<body>

<h1>My first PHP page</h1>

<?php
echo "Hello World!";
?>

</body>
</html>
```

**Output : My first PHP page**
Hello World!

**Comments in PHP**

A comment in PHP code is a line that is not read/executed as part of the program. Its only purpose is to be read by someone who is looking at the code.

Comments can be used to:

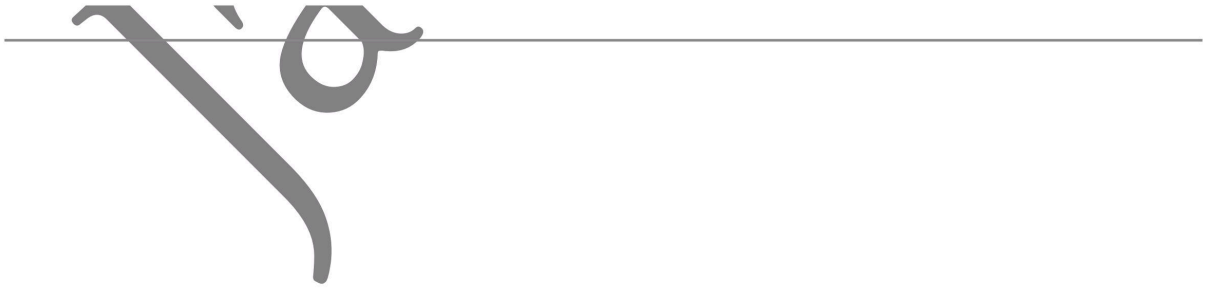
- Let others understand what you are doing
- Remind yourself of what you did - Most programmers have experienced coming back to their own work a year or two later and having to re-figure out what they did. Comments can remind you of what you were thinking when you wrote the code

PHP supports several ways of commenting:

**Example**

```
<!DOCTYPE html>
<html>
<body>

<?php
// This is a single-line comment

# This is also a single-line comment

/*
This is a multiple-lines comment block
that spans over multiple
lines
*/

// You can also use comments to leave out parts of a code line
$x = 5 /* + 15 */ + 5;
echo $x;
?>

</body>
</html>
```

Output : 10



**PHP Case Sensitivity**

In PHP, all keywords (e.g. if, else, while, echo, etc.), classes, functions, and user-defined functions are NOT case-sensitive.

In the example below, all three echo statements below are legal (and equal):

Example

```
<!DOCTYPE html>
<html>
<body>

<?php
ECHO "Hello World!<br>";
echo "Hello World!<br>";
EcHo "Hello World!<br>";
?>

</body>
</html>
```

However; all variable names are case-sensitive.

In the example below, only the first statement will display the value of the $color variable (this is because $color, $COLOR, and $coLOR are treated as three different variables):

Example

```
<!DOCTYPE html>
<html>
<body>
<?php
$color = "red";
echo "My car is " . $color . "<br>";
echo "My house is " . $COLOR . "<br>";
echo "My boat is " . $coLOR . "<br>";
?>
</body>
</html>
```

Output : My car is red
My house is
My boat is

**Variables**

1. Variables are "containers" for storing information.

2. A variable can have a short name (like x and y) or a more descriptive name

(age, carname,total_volume).

3. Unlike other programming languages, PHP has no command for declaring a variable.

   It is created the moment you first assign a value to it.

4. When you assign a text value to a variable, put quotes around the value.

5. PHP variable names are case-sensitive

**Creating (Declaring) PHP Variables**

In PHP, a variable starts with the $ sign, followed by the name of the variable:

Example

```
<?php
$txt = "Hello world!";
$x = 5;
$y = 10.5;
?>
```

After the execution of the statements above, the variable **$txt** will hold the value **Hello world!**, the variable **$x** will hold the value**5**, and the variable **$y** will hold the value **10.5**.

Rules for PHP variables:

- A variable starts with the $ sign, followed by the name of the variable
- A variable name must start with a letter or the underscore character
- A variable name cannot start with a number
- A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _ )
- Variable names are case-sensitive ($age and $AGE are two different variables)

Output Variables

The PHP echo statement is often used to output data to the screen.

The following example will output the sum of two variables:

**Example**

```
<?php
$x = 5;
$y = 4;
echo $x + $y;
?>
```

PHP is a Loosely Typed Language

In the example above, notice that we did not have to tell PHP which data type the variable is.

PHP automatically converts the variable to the correct data type, depending on its value.

In other languages such as C, C++, and Java, the programmer must declare the name and type of the variable before using it.

### Scope Variables

In PHP, variables can be declared anywhere in the script.
The scope of a variable is the part of the script where the variable can be referenced/used.
PHP has three different variable scopes:

- local
- global
- static

### Global and Local Scope

A variable declared **outside** a function has a GLOBAL SCOPE and can only be accessed outside a function:

Example

```
<?php
$x = 5; // global scope

function myTest()

{
   // using x inside this function will generate an error
   echo "<p>Variable x inside function is: $x</p>";
}
myTest();

echo "<p>Variable x outside function is: $x</p>";
?>
```

A variable declared **within** a function has a LOCAL SCOPE and can only be accessed within that function:

Example

```
<?php
function myTest()

{
   $x = 5; // local scope
   echo "<p>Variable x inside function is: $x</p>";
}
myTest();
```

```
// using x outside the function will generate an error
echo "<p>Variable x outside function is: $x</p>";
?>
```

You can have local variables with the same name in different functions, because local variables are only recognized by the function in which they are declared.

**The global Keyword**

The global keyword is used to access a global variable from within a function.

To do this, use the global keyword before the variables (inside the function):

Example

```
<?php
$x = 5;
$y = 10;
function myTest()

{
   global $x, $y;
   $y = $x + $y;
}

myTest();
echo $y;

?>

// outputs 15
```

PHP also stores all global variables in an array called $GLOBALS[*index*]. The *index* holds the name of the variable. This array is also accessible from within functions and can be used to update global variables directly.

The example above can be rewritten like this:

Example

```
<?php
$x = 5;
$y = 10;

function myTest()
```

```
{
  $GLOBALS['y'] = $GLOBALS['x'] + $GLOBALS['y'];
}

myTest();
echo $y; // outputs 15
?>
```

**The static Keyword**

Normally, when a function is completed/executed, all of its variables are deleted. However, sometimes we want a local variable NOT to be deleted. We need it for a further job.To do this, use the **static** keyword when you first declare the variable:

Example

```
<?php
function myTest()

{
  static $x = 0;
  echo $x;
  $x++;
}
myTest();
myTest();
myTest();
?>
```

Then, each time the function is called, that variable will still have the information it contained from the last time the function was called.

**Note:** The variable is still local to the function

**Data Types**

Variables can store data of different types, and different data types can do different things.

PHP supports the following data types:

- String
- Integer
- Float (floating point numbers - also called double)
- Boolean
- Array
- Object
- NULL

**String**

A string is a sequence of characters, like "Hello world!".
A string can be any text inside quotes. You can use single or double quotes:

Example

```
<?php
$x = "Hello world!";
$y = 'Hello world!';

echo $x;
echo "<br>";
echo $y;
?>
```

**Integer**

An integer data type is a non-decimal number between -2,147,483,648 and 2,147,483,647.

Rules for integers:

- An integer must have at least one digit
- An integer must not have a decimal point
- An integer can be either positive or negative
- Integers can be specified in three formats: decimal (10-based), hexadecimal (16-based - prefixed with 0x) or octal (8-based - prefixed with 0)

In the following example $x is an integer. The PHP var_dump() function returns the data type and value:

Example

```
<?php
$x = 5985;
var_dump($x);
?>
```

**Float**

A float (floating point number) is a number with a decimal point or a number in exponential form.

In the following example $x is a float. The PHP var_dump() function returns the data type and value:

Example

```
<?php
$x = 10.365;
var_dump($x);
?>
```

**Boolean**

A Boolean represents two possible states: TRUE or FALSE.

```
$x = true;
$y = false;
```

Booleans are often used in conditional testing. You will learn more about conditional testing in a later chapter of this tutorial.

**Array**

An array stores multiple values in one single variable.

In the following example $cars is an array. The PHP var_dump() function returns the data type and value:

Example

```
<?php
$cars = array("Volvo","BMW","Toyota");
var_dump($cars);
?>
```

**Object**

An object is a data type which stores data and information on how to process that data.

In PHP, an object must be explicitly declared.

First we must declare a class of object. For this, we use the class keyword. A class is a structure that can contain properties and methods:

Example

```
<?php
class Car

{
  function Car()
```

```
    {
      $this->model = "VW";
    }
}

// create an object
$cartype= new Car();

// show object properties
echo $cartype->model;
?>
```

---

**NULL Value**

Null is a special data type which can have only one value: NULL.

A variable of data type NULL is a variable that has no value assigned to it.

If a variable is created without a value, it is automatically assigned a value of NULL.

Variables can also be emptied by setting the value to NULL:

Example

```
<?php
$x = "Hello world!";
$x = null;
var_dump($x);
?>
```

output:NULL

**Operators**

Operators are used to perform operations on variables and values.

PHP divides the operators in the following groups:

- Arithmetic operators
- Assignment operators
- Comparison operators
- Increment/Decrement operators
- Logical operators
- String operators
- Array operators

---

**Arithmetic Operators**

The PHP arithmetic operators are used with numeric values to perform common arithmetical operations, such as addition, subtraction, multiplication etc.

| Operator | Name | Example | Result |
|---|---|---|---|
| + | Addition | $x + $y | Sum of $x and $y |
| - | Subtraction | $x - $y | Difference of $x and $y |
| * | Multiplication | $x * $y | Product of $x and $y |
| / | Division | $x / $y | Quotient of $x and $y |
| % | Modulus | $x % $y | Remainder of $x divided by $y |
| ** | Exponentiation | $x ** $y | Result of raising $x to the $y'th power (Introduced in PHP 5. |

**Assignment Operators**

The PHP assignment operators are used with numeric values to write a value to a variable.

The basic assignment operator in PHP is "=". It means that the left operand gets set to the value of the assignment expression on the right.

| Assignment | Same as... | Description |
|---|---|---|
| x = y | x = y | The left operand gets set to the value of the expression on the right |

| | | |
|---|---|---|
| x += y | x = x + y | Addition |
| x -= y | x = x - y | Subtraction |
| x *= y | x = x * y | Multiplication |
| x /= y | x = x / y | Division |
| x %= y | x = x % y | Modulus |

**Comparison Operators**

The PHP comparison operators are used to compare two values (number or string):

| Operator | Name | Example | Result |
|---|---|---|---|
| == | Equal | $x == $y | Returns true if $x is equal to $y |
| === | Identical | $x === $y | Returns true if $x is equal to $y, and they are of the same type |
| != | Not equal | $x != $y | Returns true if $x is not equal to $y |
| <> | Not equal | $x <> $y | Returns true if $x is not equal to $y |
| !== | Not identical | $x !== $y | Returns true if $x is not equal to $y, or they are not of the same ty |

| | | | |
|---|---|---|---|
| > | Greater than | $x > $y | Returns true if $x is greater than $y |
| < | Less than | $x < $y | Returns true if $x is less than $y |
| >= | Greater than or equal to | $x >= $y | Returns true if $x is greater than or equal to $y |
| <= | Less than or equal to | $x <= $y | Returns true if $x is less than or equal to $y |

**Increment / Decrement Operators**

The PHP increment operators are used to increment a variable's value.

The PHP decrement operators are used to decrement a variable's value.

| Operator | Name | Description |
|---|---|---|
| ++$x | Pre-increment | Increments $x by one, then returns $x |
| $x++ | Post-increment | Returns $x, then increments $x by one |
| --$x | Pre-decrement | Decrements $x by one, then returns $x |
| $x-- | Post-decrement | Returns $x, then decrements $x by one |

**Logical Operators**

The PHP logical operators are used to combine conditional statements.

| Operator | Name | Example | Result |
|---|---|---|---|
| and | And | $x and $y | True if both $x and $y are true |
| or | Or | $x or $y | True if either $x or $y is true |
| xor | Xor | $x xor $y | True if either $x or $y is true, but not both |
| && | And | $x && $y | True if both $x and $y are true |
| \|\| | Or | $x \|\| $y | True if either $x or $y is true |
| ! | Not | !$x | True if $x is not true |

**String Operators**

PHP has two operators that are specially designed for strings.

| Operator | Name | Example | Result |
|---|---|---|---|
| . | Concatenation | $txt1 . $txt2 | Concatenation of $txt1 and $txt2 |
| .= | Concatenation assignment | $txt1 .= $txt2 | Appends $txt2 to $txt1 |

**Array Operators**

The PHP array operators are used to compare arrays.

| Operator | Name | Example | Result |
|---|---|---|---|
| + | Union | $x + $y | Union of $x and $y |
| == | Equality | $x == $y | Returns true if $x and $y have the same value pairs |
| === | Identity | $x === $y | Returns true if $x and $y have the same value pairs in the same o types |
| != | Inequality | $x != $y | Returns true if $x is not equal to $y |
| <> | Inequality | $x <> $y | Returns true if $x is not equal to $y |
| !== | Non-identity | $x !== $y | Returns true if $x is not identical to $y |

### 2.5. Flow-Control Statements

PHP supports a number of traditional programming constructs for controlling the flow of execution of a program.

Conditional statements, such as `if`/`else` and `switch`, allow a program to execute different pieces of code, or none at all, depending on some condition. Loops, such as`while` and `for`, support the repeated execution of particular code.

#### 2.5.1. if

The `if` statement checks the truthfulness of an expression and, if the expression is true, evaluates a statement. An `if` statement looks like:

```
if (expression)
  statement
```

To specify an alternative statement to execute when the expression is false, use the `else` keyword:

```
if (expression)
  statement
else
  statement
```

For example:

```
if ($user_validated)
  echo "Welcome!";
else
  echo "Access Forbidden!";
```

To include more than one statement in an `if` statement, use a *block* —a curly brace-enclosed set of statements:

```
if ($user_validated) {
  echo 'Welcome!";
  $greeted = 1;
} else {
  echo "Access Forbidden!";
  exit;
}
```

PHP provides another syntax for blocks in tests and loops. Instead of enclosing the block of statements in curly braces, end the `if` line with a colon (`:`) and use a specific keyword to end the block (`endif`, in this case). For example:

```
if ($user_validated) :
  echo "Welcome!";
  $greeted = 1;
else :
  echo "Access Forbidden!";
  exit;
endif;
```

Other statements described in this chapter also have similar alternate style syntax (and ending keywords); they can be useful if you have large blocks of HTML inside your statements. For example:

```
<?if($user_validated):?>
  <table>
    <tr>
      <td>First Name:</td><td>Sophia</td>
    </tr>
    <tr>
      <td>Last Name:</td><td>Lee</td>
    </tr>
  </table>
<?else:?>
  Please log in.
<?endif?>
```

Because `if` is a statement, you can chain them:

```
if ($good)
```

```
  print('Dandy!');
else
  if ($error)
    print('Oh, no!');
  else
    print("I'm ambivalent...");
```

Such chains of `if` statements are common enough that PHP provides an easier syntax: the `elseif` statement. For example, the previous code can be rewritten as:

```
if ($good)
  print('Dandy!');
elseif ($error)
  print('Oh, no!');
else
  print("I'm ambivalent...");
```

The ternary conditional operator (`?:`) can be used to shorten simple true/false tests. Take a common situation such as checking to see if a given variable is true and printing something if it is. With a normal `if`/`else` statement, it looks like this:

```
<td><? if($active) echo 'yes'; else echo 'no'; ?></td>
```

With the ternary conditional operator, it looks like this:

```
<? echo '<td>'.($active ? 'yes':'no').'</td>' ?>
```

Compare the syntax of the two:

```
if (expression) true_statement else false_statement
(expression) ? true_expression : false_expression
```

The main difference here is that the conditional operator is not a statement at all. This means that it is used on expressions, and the result of a complete ternary expression is itself an expression. In the previous example, the `echo` statement is inside the `if` condition, while when used with the ternary operator, it precedes the expression.

#### 2.5.2. switch

It often is the case that the value of a single variable may determine one of a number of different choices (e.g., the variable holds the username and you want to do something different for each user). The `switch` statement is designed for just this situation.

A `switch` statement is given an expression and compares its value to all cases in the switch; all statements in a matching case are executed, up to the first `break`keyword it finds. If none match, and a `default` is given, all statements following the `default` keyword are executed, up to the first `break` keyword encountered.

For example, suppose you have the following:

```
if ($name == 'ktatroe')
  // do something
elseif ($name == 'rasmus')
  // do something
elseif ($name == 'ricm')
  // do something
elseif ($name == 'bobk')
  // do something
```

You can replace that statement with the following `switch` statement:

```
switch($name) {
  case 'ktatroe':
    // do something
    break;
  case 'rasmus':
    // do something
    break;
  case 'ricm':
    // do something
    break;
  case 'bobk':
    // do something
    break;
}
```

The alternative syntax for this is:

```
switch($name):
  case 'ktatroe':
    // do something
    break;
  case 'rasmus':
    // do something
    break;
  case 'ricm':
    // do something
    break;
  case 'bobk':
    // do something
    break;
endswitch;
```

Because statements are executed from the matching case label to the next `break` keyword, you can combine several cases in a *fall-through*. In the following example, "yes" is printed when `$name` is equal to "sylvie" or to "bruno":

```
switch ($name) {
  case 'sylvie': // fall-through
  case 'bruno':
    print('yes');
    break;
  default:
    print('no');
    break;
}
```

Commenting the fact that you are using a fall-through case in a `switch` is a good idea, so someone doesn't come along at some point and add a `break`, thinking you had forgotten it.

You can specify an optional number of levels for the break keyword to break out of. In this way, a `break` statement can break out of several levels of nested `switch`statements. An example of using `break` in this manner is shown in the next section.

#### 2.5.3. while

The simplest form of loop is the `while` statement:

```
while (expression)
  statement
```

If the *`expression`* evaluates to `true`, the *`statement`* is executed and then the *`expression`* is reevaluated (if it is `true`, the body of the loop is executed, and so on). The loop exits when the *`expression`* evaluates to `false`.

As an example, here's some code that adds the whole numbers from 1 to 10:

```
$total = 0;
$i = 1;
while ($i <= 10) {
  $total += $i;
}
```

The alternative syntax for `while` has this structure:

```
while (expr):
  statement;
  ...;
endwhile;
```

For example:

```
$total = 0;
$i = 1;
while ($i <= 10):
  $total += $i;
endwhile;
```

You can prematurely exit a loop with the `break` keyword. In the following code, `$i` never reaches a value of `6`, because the loop is stopped once it reaches `5`:

```
$total = 0;
$i = 1;
while ($i <= 10) {
  if ($i == 5)
    break; // breaks out of the loop
  $total += $i;
  $i++;
}
```

Optionally, you can put a number after the `break` keyword, indicating how many levels of loop structures to break out of. In this way, a statement buried deep in nested loops can break out of the outermost loop. For example:

```
$i = 0;
while ($i < 10)
{
  while ($j < 10)
{
    if ($j == 5)
      break 2; // breaks out of two while loops
    $j++;
  }
  $i++;
}
echo $i;
echo $j;
0
5
```

The continue statement skips ahead to the next test of the loop condition. As with the break keyword, you can continue through an optional number of levels of loop structure:

```
while ($i < 10)
{
  while ($j < 10)
{
    if ($j = 5)
      continue 2; // continues through two levels
    $j++;
  }
  $i++;
}
```

In this code, `$j` never has a value above `5`, but `$i` goes through all values from `0` through `9`.

PHP also supports a `do`/`while` loop, which takes the following form:

```
do
  statement
while (expression)
```

Use a `do`/`while` loop to ensure that the loop body is executed at least once:

```
$total = 0;
$i = 1;
do
{
  $total += $i++;
} while ($i <= 10);
```

You can use `break` and `continue` statements in a `do`/`while` statement just as in a normal `while` statement.

The `do`/`while` statement is sometimes used to break out of a block of code when an error condition occurs. For example:

```
do {
  // do some stuff
  if ($error_condition)
    break;
  // do some other stuff
} while (false);
```

Because the condition for the loop is `false`, the loop is executed only once, regardless of what happens inside the loop. However, if an error occurs, the code after the`break` is not evaluated.

#### 2.5.4. for

The `for` statement is similar to the `while` statement, except it adds counter initialization and counter manipulation expressions, and is often shorter and easier to read than the equivalent `while` loop.

Here's a `while` loop that counts from 0 to 9, printing each number:

```
$counter = 0;
while ($counter < 10)
{
  echo "Counter is $counter\n";
  $counter++;
}
```

Here's the corresponding, more concise `for` loop:

```
for ($counter = 0; $counter < 10; $counter++)
  echo "Counter is $counter\n";
```

The structure of a `for` statement is:

```
for (start; condition; increment)
  statement
```

The expression *`start`* is evaluated once, at the beginning of the `for` statement. Each time through the loop, the expression *`condition`* is tested. If it is `true`, the body of the loop is executed; if it is `false`, the loop ends. The expression *`increment`* is evaluated after the loop body runs.

The alternative syntax of a `for` statement is:

```
for (expr1; expr2; expr3):
  statement;
  ...;
endfor;
```

This program adds the numbers from 1 to 10 using a `for` loop:

```
$total = 0;
for ($i= 1; $i <= 10; $i++)
{
  $total += $i;
}
```

Here's the same loop using the alternate syntax:

```
$total = 0;
for ($i = 1; $i <= 10; $i++):
  $total += $i;
endfor;
```

You can specify multiple expressions for any of the expressions in a `for` statement by separating the expressions with commas. For example:

```
$total = 0;
for ($i = 0, $j = 0; $i <= 10; $i++, $j *= 2)
{
  $total += $j;
}
```

You can also leave an expression empty, signaling that nothing should be done for that phase. In the most degenerate form, the `for` statement becomes an infinite loop. You probably don't want to run this example, as it never stops printing:

```
for (;;)
{
  echo "Can't stop me!<br />";
}
```

In `for` loops, as in `while` loops, you can use the `break` and `continue` keywords to end the loop or the current iteration.

#### 2.5.5. foreach

The `foreach` statement allows you to iterate over elements in an array. To loop over an array, accessing each key, use:

```
foreach ($array as $current)
{
  // ...
}
```

The alternate syntax is:

```
foreach ($array as $current):
  // ...
endforeach;
```

To loop over an array, accessing both key and value, use:

```
foreach ($array as $key => $value)
{
  // ...
```

```
}
```

The alternate syntax is:

```
foreach ($array as $key => $value):
  // ...
endforeach;
```

#### 2.5.6. declare

The `declare` statement allows you to specify execution directives for a block of code. The structure of a `declare` statement is:

```
declare (directive)
  statement
```

Currently, there is only one `declare` form, the ticks directive. Using it, you can specify how frequently (measured roughly in number of code statements) a tick function registered with `register_tick_function( )` is called. For example:

```
register_tick_function("some_function");
declare(ticks = 3)
{
  for($i = 0; $i < 10; $i++)
{
    // do something
  }
}
```

In this code, `some_function( )` is called after every third statement is executed.

#### 2.5.7. exit and return

The `exit` statement ends execution of the script as soon as it is reached.
The `return` statement returns from a function or (at the top level of the program) from the script.

The `exit` statement takes an optional value. If this is a number, it's the exit status of the process. If it's a string, the value is printed before the process terminates. The`exit( )` construct is an alias for `die( )`:

```
$handle = @mysql_connect("localhost", $USERNAME, $PASSWORD);
if (!$handle) {
  die("Could not connect to database");
}
```

This is more commonly written as:

```
$handle = @mysql_connect("localhost", $USERNAME, $PASSWORD)
          or die("Could not connect to database");
```

## Including Code

PHP provides two constructs to load code and HTML from another module: require and include. They both load a file as the PHP script runs, work in conditionals and loops, and complain if the file being loaded can't be found. The main difference is that attempting to require a nonexistent file is a fatal error, while attempting toinclude such a file produces a warning but does not stop script execution.

A common use of include is to separate page-specific content from general site design. Common elements such as headers and footers go in separate HTML files, and each page then looks like:

```
<? include 'header.html'; ?>
```

```
content
<? include 'footer.html'; ?>
```

We use include because it allows PHP to continue to process the page even if there's an error in the site design file(s). The require construct is less forgiving and is more suited to loading code libraries, where the page can't be displayed if the libraries don't load. For example:

```
require 'codelib.inc';
mysub( );              // defined in codelib.inc
```

A marginally more efficient way to handle headers and footers is to load a single file and then call functions to generate the standardized site elements:

```
<? require 'design.inc';
   header( );
?>
content
<? footer( ); ?>
```

If PHP cannot parse some part of a file included by include or require, a warning is printed and execution continues. You can silence the warning by prepending the call with the silence operator; for example, @include.

If the allow_url_fopen option is enabled through PHP's configuration file, *php.ini*, you can include files from a remote site by providing a URL instead of a simple local path:

```
include 'http://www.example.com/codelib.inc';
```

If the filename begins with "http://" or "ftp://", the file is retrieved from a remote site and then loaded.

Files included with include and require can be arbitrarily named. Common extensions are *.php*, *.inc*, and *.html*. Note that remotelyfetching a file that ends in *.php*from a web server that has PHP enabled fetches the *output* of that PHP script. For this reason, we recommend you use *.inc* for library files that primarily contain code and *.html* for library files that primarily contain HTML.

If a program uses include or require to include the same file twice, the file is loaded and the code is run or the HTML is printed twice. This can result in errors about the redefinition of functions or multiple copies of headers or HTML being sent. To prevent these errors from occurring, use the include_once and require_onceconstructs. They behave the same as include and require the first time a file is loaded, but quietly ignore subsequent attempts to load the same file. For example, many page elements, each stored in separate files, need to know the current user's preferences. The element libraries should load the user preferences library withrequire_once. The page designer can then include a page element without worrying about whether the user preference code has already been loaded.

Code in an included file is imported at the scope that is in effect where the include statement is found, so the included code can see and alter your code's variables. This can be useful—for instance, a user-tracking library might store the current user's name in the global $user variable:

```
// main page
include 'userprefs.inc';
echo "Hello, $user.";
```

The ability of libraries to see and change your variables can also be a problem. You have to know every global variable used by a library to ensure that you don't accidentally try to use one of them for your own purposes, thereby overwriting the library's value and disrupting how it works.

If the include or require construct is in a function, the variables in the included file become function-scope variables for that function.

Because include and require are keywords, not real statements, you must always enclose them in curly braces in conditional and loop statements:

```
for ($i=0; $i < 10; $i++) {
  include "repeated_element.html";
}
```

Use the get_included_files( ) function to learn which files your script has included or required. It returns an array containing the full system path filenames of each included or required file. Files that did not parse are not included in this array.

**Embedding PHP in Web Pages**

Although it is possible to write and run standalone PHP programs, most PHP code is embedded in HTML or XML files. This is, after all, why it was created in the first place. Processing such documents involves replacing each chunk of PHP source code with the output it produces when executed.

Because a single file contains PHP and non-PHP source code, we need a way to identify the regions of PHP code to be executed. PHP provides four different ways to do this.

As you'll see, the first, and preferred, method looks like XML. The second method looks like SGML. The third method is based on ASP tags. The fourth method uses the standard HTML <script> tag; this makes it easy to edit pages with enabled PHP using a regular HTML editor.

**1. XML Style**

Because of the advent of the eXtensible Markup Language (XML) and the migration of HTML to an XML language (XHTML), the currently preferred technique for embedding PHP uses XML-compliant tags to denote PHP instructions.

Coming up with tags to demark PHP commands in XML was easy, because XML allows the definition of new tags. To use this style, surround your PHP code with <?php and ?>. Everything between these markers is interpreted as PHP, and everything outside the markers is not. Although it is not necessary to include spaces between the markers and the enclosed text, doing so improves readability. For example, to get PHP to print "Hello, world", you can insert the following line in a web page:

```
<?php echo "Hello, world"; ?>
```

The trailing semicolon on the statement is optional, because the end of the block also forces the end of the expression. Embedded in a complete HTML file, this looks like:

```
<html>
<head>
  <title>This is my first PHP program!</title>
</head>
<body>
<p>
  Look, ma! It's my first PHP program:<br />
  <?php echo "Hello, world"; ?><br />
  How cool is that?
</p>
</body>
</html>
```

Of course, this isn't very exciting—we could have done it without PHP. The real value of PHP comes when we put dynamic information from sources such as databases and form values into the web page. That's for a later chapter, though. Let's get back to our "Hello, world" example. When a user visits this page and views its source, it looks like this:

```
<html>
```

```
<head>
  <title>This is my first PHP program!</title>
</head>
<body>
<p>
  Look, ma! It's my first PHP program:<br />
  Hello, world!<br />
  How cool is that?
</p>
</body>
</html>
```

Notice that there's no trace of the PHP source code from the original file. The user sees only its output.

Also notice that we switched between PHP and non-PHP, all in the space of a single line. PHP instructions can be put anywhere in a file, even within valid HTML tags. For example:

```
<input type="text" name="first_name" value="<?php echo "Rasmus"; ?>" >
```

When PHP is done with this text, it will read:

```
<input type="text" name="first_name" value="Rasmus" >
```

The PHP code within the opening and closing markers does not have to be on the same line. If the closing marker of a PHP instruction is the last thing on a line, the line break following the closing tag is removed as well. Thus, we can replace the PHP instructions in the "Hello, world" example with:

```
<?php
 echo "Hello, world"; ?>
<br />
```

with no change in the resulting HTML.

**2. SGML Style**

The "classic" style of embedding PHP comes from SGML instruction processing tags. To use this method, simply enclose the PHP in <? and ?>. Here's the "Hello world" example again:

```
<? echo "Hello, world"; ?>
```

This style, known as *short tags*, is the shortest and least intrusive, and it can be turned off so as to not clash with the XML PI (Process Instruction) tag in the *php.ini*initialization file. Consequently, if you want to write fully portable PHP code that you are going to distribute to other people (who might have short tags turned off ), you should use the longer <?php ... ?> style, which cannot be turned off. If you have no intention of distributing your code, you don't have an issue with telling people who want to use your code to turn on short tags, and you are not planning on mixing XML in with your PHP code, then using this tag style is okay.

### 3. ASP Style

Because neither the SGML nor XML tag style is strictly legal HTML,[3] some HTML editors do not parse it correctly for color syntax highlighting, context-sensitive help, and other such niceties. Some will even go so far as to helpfully remove the "offending" code for you.

Mostly because you are not allowed to use a > inside your tags if you wish to be compliant, but who wants to write code like if( $a > 5 )...?

However, many of these same HTML editors recognize another mechanism (no more legal than PHP's) for embedding code—that of Microsoft's Active Server Pages (ASP). Like PHP, ASP is a method for embedding server-side scripts within documents.

If you want to use ASP-aware tools to edit files that contain embedded PHP, you can use ASP-style tags to identify PHP regions. The ASP-style tag is the same as the SGML-style tag, but with % instead of ?:

```
<% echo "Hello, world"; %>
```

In all other ways, the ASP-style tag works the same as the SGML-style tag.

ASP-style tags are not enabled by default. To use these tags, either build PHP with the --enable-asp-tags option or enable asp_tags in the PHP configuration file.

### 4. Script Style

The final method of distinguishing PHP from HTML involves a tag invented to allow client-side scripting within HTML pages, the <script> tag. You might recognize it as the tag in which JavaScript is embedded. Since PHP is processed and removed from the file before it reaches the browser, you can use the <script> tag to surround PHP code. To use this method, simply specify "php" as the value of the language attribute of the tag:

```
<script language="php">
  echo "Hello, world";
</script>
```

This method is most useful with HTML editors that work only on strictly legal HTML files and don't yet support XML processing commands.

### 5. Echoing Content Directly

Perhaps the single most common operation within a PHP application is displaying data to the user. In the context of a web application, this means inserting into the HTML document information that will become HTML when viewed by the user.

To simplify this operation, PHP provides special versions of the SGML and ASP tags that automatically take the value inside the tag and insert it into the HTML page. To use this feature, add an equals sign (=) to the opening tag. With this technique, we can rewrite our form example as:

```
<input type="text" name="first_name" value="<?="Rasmus"; ?>">
```

If you have ASP-style tags enabled, you can do the same with your ASP tags:

```
<p>This number (<%= 2 + 2 %>)<br />
and this number (<% echo (2 + 2); %>) <br />
Are the same.</p>
```

After processing, the resulting HTML is:

```
<p>This number (4) <br />
and this number (4) <br />
are the same.</p>
```

---

## UNIT : 2

## Functions

A *function* is a named block of code that performs a specific task, possibly acting upon a set of values given to it, or *parameters*, and possibly returning a single value. Functions save on compile time—no matter how many times you call them, functions are compiled only once for the page. They also improve reliability by allowing you to fix any bugs in one place, rather than everywhere you perform a task, and they improve readability by isolating code that performs specific tasks.

### 1. Calling a Function

Functions in a PHP program can be either built-in (or, by being in an extension, effectively built-in) or user-defined. Regardless of their source, all functions are evaluated in the same way:

```
$some_value = function_name( [ parameter, ... ] );
```

The number of parameters a function requires differs from function to function (and, as we'll see later, may even vary for the same function). The parameters supplied to the function may be any valid expression and should be in the specific order expected by the function. A function's documentation will tell you what parameters the function expects and what values you can expect to be returned.

Here are some examples of functions:

```
// strlen( ) is a built-in function that returns the length of a string
$length = strlen("PHP"); // $length is now 3
// sin() and asin( ) are the sine and arcsine math functions
$result = sin(asin(1)); // $result is the sine of arcsin(1), or 1.0
// unlink( ) deletes a file
```

```
$result = unlink("functions.txt"); // false if unsuccessful
```

In the first example, we give an argument, "PHP", to the function strlen( ), which gives us the number of characters in the string it's given. In this case, it returns 3, which is assigned to the variable $length. This is the simplest and most common way to use a function.

The second example passes the result of asin(1) to the sin( ) function. Since the sine and arcsine functions are reflexive, taking the sine of the arcsine of any value will always return that same value.

In the final example, we give a filename to the unlink( ) function, which attempts to delete the file. Like many functions, it returns false when it fails. This allows you to use another built-in function, die( ), and the short-circuiting property of the logic operators. Thus, this example might be rewritten as:

```
$result = unlink("functions.txt") or die("Operation failed!");
```

The unlink( ) function, unlike the other two examples, affects something outside of the parameters given to it. In this case, it deletes a file from the file system. All such side effects of a function should be carefully documented.

**2. Defining a Function**

To define a function, use the following syntax:

```
function [&] function_name ( [ parameter [, ... ] ] )
{
  statement list
}
```

The statement list can include HTML. You can declare a PHP function that doesn't contain any PHP code. For instance, the column( ) function simply gives a convenient short name to HTML code that may be needed many times throughout the page:

```
<? function column( ) { ?>
</td><td>
<? } ?>
```

The function name can be any string that starts with a letter or underscore followed by zero or more letters, underscores, and digits. Function names are case-insensitive; that is, you can call the sin( ) function as sin(1), SIN(1), SiN(1), and so on, because all these names refer to the same function.

Typically, functions return some value. To return a value from a function, use the return statement: put return *expr* inside your function. When a return statement is encountered during execution, control reverts to the calling statement, and the evaluated results of *expr* will be returned as the value of the function. Although it can make for messy code, you can actually include multiple return statements in a function if it makes sense (for example, if you have a switch statement to determine which of several values to return).

If you define your function with the optional ampersand before the name, the function returns a reference to the returned data rather than a copy of the data.

***Example 3-1. String concatenation***

```
function strcat($left, $right)
{
  $combined_string = $left . $right;
  return $combined_string;
}
```

The function takes two arguments, $left and $right. Using the concatenation operator, the function creates a combined string in the variable $combined_string. Finally, in order to cause the function to have a value when it's evaluated with our arguments, we return the value $combined_string.

Because the return statement can accept any expression, even complex ones, we can simplify the program:

***Example 3-2. String concatenation redux***

```
function strcat($left, $right)
{
  return $left . $right;
}
```

If we put this function on a PHP page, we can call it from anywhere within the page. Take a look at Example 3-3.

***Example 3-3. Using our concatenation function***

```
<?php
 function strcat($left, $right)
{
   return $left . $right;
 }
 $first = "This is a ";
 $second = " complete sentence!";
 echo strcat($first, $second);
?>
```

When this page is displayed, the full sentence is shown.

This function takes in an integer, doubles it, and returns the result:

```
function doubler($value)
{
  return $value << 1;
}
```

Once the function is defined, you can use it anywhere on the page. For example:

```
<?= 'A pair of 13s is ' . doubler(13); ?>
```

You can nest function declarations, but with limited effect. Nested declarations do not limit the visibility of the inner-defined function, which may be called from anywhere in your program. The inner function does

not automatically get the outer function's arguments. And, finally, the inner function cannot be called until the outer function has been called.

```
function outer ($a)
{
  function inner ($b)
{
    echo "there $b";
  }
  echo "$a, hello ";
}
outer("well");
inner("reader");
```

**well, hello there reader**

## 3.Variable Scope

Up to this point, if you don't use functions, any variable you create can be used anywhere in a page. With functions, this is no longer always true. Functions keep their own sets of variables that are distinct from those of the page and of other functions.

The variables defined in a function, including its parameters, are not accessible outside the function, and, by default, variables defined outside a function are not accessible inside the function. The following example illustrates this:

```
$a = 3;
function foo( )
{
  $a += 2;
}
foo( );
echo $a;
```

The variable $a inside the function foo( ) is a different variable than the variable $a outside the variable; even though foo( ) uses the add-and-assign operator, the value of the outer $a remains 3 throughout the life of the page. Inside the function, $a has the value 2.

As we discussed earlier the extent to which a variable can be seen in a program is called the *scope* of the variable. Variables created within a function are inside the scope of the function (i.e., have *function-level scope*). Variables created outside of functions and objects have *global scope* and exist anywhere outside of those functions and objects. A few variables provided by PHP have both function-level and global scope.

At first glance, even an experienced programmer may think that in the previous example $a will be 5 by the time the echo statement is reached, so keep that in mind when choosing names for your variables.

### 3.1. Global Variables

If you want a variable in the global scope to be accessible from within a function, you can use the global keyword. Its syntax is:

```
global var1, var2, ...
```

Changing the previous example to include a global keyword, we get:

```
$a = 3;
function foo( )
{
  global $a;
  $a += 2;
}
foo( );
echo $a;
```

Instead of creating a new variable called $a with function-level scope, PHP uses the global $a within the function. Now, when the value of $a is displayed, it will be 5.

You must include the global keyword in a function before any uses of the global variable or variables you want to access. Because they are declared before the body of the function, function parameters can never be global variables.

Using global is equivalent to creating a reference to the variable in the $GLOBALS variable. That is, the following declarations:

```
global $var;
$var = &$GLOBALS['var'];
```

both create a variable in the function's scope that is a reference to the same value as the variable $var in the global scope.

### 3.2. Static Variables

Like C, PHP supports declaring function variables *static*. A static variable is shared between all calls to the function and is initialized during a script's execution only the first time the function is called. To declare a function variable static, use the static keyword at the variable's first use. Typically, the first use of a static variable is to assign an initial value:

```
static var [= value][, ... ];
```

In Example 3-4, the variable $count is incremented by one each time the function is called.

***Example 3-4. Static variable counter***

```
function counter( )
{
  static $count = 0;
  return $count++;
}
for ($i = 1; $i <= 5; $i++)
{
  print counter( );
}
```

When the function is called for the first time, the static variable $count is assigned a value of 0. The value is returned and $count is incremented. When the function ends, $count is not destroyed like a non-static

variable, and its value remains the same until the next time counter( ) is called. The for loop displays the numbers from 0 to 4.

**4. Function Parameters**

Functions can expect, by declaring them in the function definition, an arbitrary number of arguments. There are two different ways of passing parameters to a function. The first, and more common, is by value. The other is by reference.

### 3.4.1. Passing Parameters by Value

In most cases, you pass parameters by value. The argument is any valid expression. That expression is evaluated, and the resulting value is assigned to the appropriate variable in the function. In all of the examples so far, we've been passing arguments by value.

### 3.4.2. Passing Parameters by Reference

Passing by reference allows you to override the normal scoping rules and give a function direct access to a variable. To be passed by reference, the argument must be a variable; you indicate that a particular argument of a function will be passed by reference by preceding the variable name in the parameter list with an ampersand (&).Example 3-5 revisits our doubler( ) function with a slight change.

***Example 3-5. Doubler redux***

```
function doubler(&$value)
{
  $value = $value << 1;
}
$a = 3;
doubler($a);
echo $a;
```

Because the function's $value parameter is passed by reference, the actual value of $a, rather than a copy of that value, is modified by the function. Before, we had to return the doubled value, but now we change the caller's variable to be the doubled value.

Here's another place where a function contains side effects: since we passed the variable $a into doubler( ) by reference, the value of $a is at the mercy of the function. In this case, doubler( ) assigns a new value to it.

A parameter that is declared as being passed by reference can only be a variable. Thus, if we included the statement <?= doubler(7); ?> in the previous example, it would issue an error.

Even in cases where your function does affect the given value, you may want a parameter to be passed by reference. When passing by value, PHP must copy the value. Particularly for large strings and objects, this can be an expensive operation. Passing by reference removes the need to copy the value.

### 3.4.3. Default Parameters

Sometimes, a function may need to accept a particular parameter in some cases. For example, when you call a function to get the preferences for a site, the function may take in a parameter with the name of the preference to retrieve. If you want to retrieve all the preferences, rather than using some special keyword, you can just not supply an argument. This behavior works by using default arguments.

To specify a default parameter, assign the parameter value in the function declaration. The value assigned to a parameter as a default value cannot be a complex expression; it can only be a constant.

```
function get_preferences($which_preference = "all" ) {
   // if $which_preference is "all", return all prefs;
   // otherwise, get the specific preference requested...
}
```

When you call get_preferences( ), you can choose to supply an argument. If you do, it returns the preference matching the string you give it; if not, it returns all preferences.

A function may have any number of parameters with default values. However, they must be listed after all the parameters that do not have default values.

### 3.4.4. Variable Parameters

A function may require a variable number of arguments. For example, the get_preferences( ) example in the previous section might return the preferences for any number of names, rather than for just one. To declare a function with a variable number of arguments, leave out the parameter block entirely.

```
function get_preferences( ) {
  // some code
}
```

PHP provides three functions you can use in the function to retrieve the parameters passed to it. func_get_args( ) returns an array of all parameters provided to the function, func_num_args( ) returns the number of parameters provided to the function, and func_get_arg( ) returns a specific argument from the parameters.

```
$array = func_get_args( );
$count = func_num_args( );
$value = func_get_arg(argument_number);
```

In Example 3-6, the count_list( ) function takes in any number of arguments. It loops over those arguments and returns the total of all the values. If no parameters are given, it returns false.

***Example 3-6. Argument counter***

```
function count_list( ) {
  if(func_num_args( ) == 0) {
    return false;
  }
  else {
    for($i = 0; $i < func_num_args( ); $i++) {
      $count += func_get_arg($i);
    }
    return $count;
  }
}
echo count_list(1, 5, 9);
```

The result of any of these functions cannot directly be used as a parameter to another function. To use the result of one of these functions as a parameter, you must first set a variable to the result of the function, then use that in the function call. The following expression will not work:

```
foo(func_num_args( ));
```

Instead, use:

```
$count = func_num_args( );
foo($count);
```

### 3.4.5. Missing Parameters

PHP lets you be as lazy as you want—when you call a function, you can pass any number of arguments to the function. Any parameters the function expects that are not passed to it remain unset, and a warning is issued for each of them:

```
function takes_two( $a, $b )
{
  if (isset($a)) { echo " a is set\n"; }
  if (isset($b)) { echo " b is set\n"; }
}
echo "With two arguments:\n";
takes_two(1, 2);
echo "With one argument:\n";
takes_two(1);
With two arguments:
 a is set
 b is set
With one argument:
Warning:  Missing argument 2 for takes_two( )
 in /path/to/script.php on line 6
a is set
```

## 5. Return Values

PHP functions can return only a single value with the return keyword:

```
function return_one()
{
   return 42;
}
```

To return multiple values, return an array:

```
function return_two ( )
{
```

```
  return array("Fred", 35);
}
```

By default, values are copied out of the function. A function declared with an & before its name returns a reference (alias) to its return value:

```
$names = array("Fred", "Barney", "Wilma", "Betty");
function & find_one($n)
{
  return $names[$n];
}
$person =& find_one(1);          // Barney
$person = "Barnetta";            // changes $names[1]
```

In this code, the find_one( ) function returns an alias for $names[1], instead of a copy of its value. Because we assign by reference, $person is an alias for$names[1], and the second assignment changes the value in $names[1].

This technique is sometimes used to return large string or array values efficiently from a function. However, PHP's copy-on-write/shallow-copy mechanism usually means that returning a reference from a function is not necessary. There is no point in returning a reference to some large piece of data unless you know you are likely to change that data. The drawback of returning the reference is that it is slower than returning the value and relying on the shallow-copy mechanism to ensure that a copy of that data is not made unless it is changed.

## 6. Variable Functions

As with variable variables, you can call a function based on the value of a variable. For example, consider this situation, where a variable is used to determine which of three functions to call:

```
switch($which)
{
 case 'first':
   first( );
   break;
 case 'second':
   second( );
   break;
 case 'third':
   third( );
   break;
}
```

In this case, we could use a variable function call to call the appropriate function. To make a variable function call, include the parameters for a function in parentheses after the variable. To rewrite the previous example:

```
$which();  // if $which is "first" the function first( ) is called, etc...
```

If no function exists for the variable, a runtime error occurs when the code is evaluated. To prevent this, you can use the built-in function function_exists( ) to determine whether a function exists for the value of the variable before calling the function:

```
$yes_or_no = function_exists(function_name);
```

For example:

```
if(function_exists($which))
{
  $which();  // if $which is "first" the function first( ) is called, etc...
}
```

Language constructs such as echo( ) and isset( ) cannot be called through variable functions:

```
$f = 'echo';
$f('hello, world');  // does not work
```

## 7. Anonymous Functions

Some PHP functions use a function you provide them with to do part of their work. For example, the usort( ) function uses a function you create and pass to it as a parameter to determine the sort order of the items in an array.

Although you can define a function for such purposes, as shown previously, these functions tend to be localized and temporary. To reflect the transient nature of the callback, create and use an *anonymous function* (or lambda function).

You can create an anonymous function using create_function( ). This function takes two parameters—the first describes the parameters the anonymous function takes in, and the second is the actual code. A randomly generated name for the function is returned:

```
$func_name = create_function(args_string, code_string);
```

Example 3-7 shows an example using usort( ).

***Example 3-7. Anonymous functions***

```
$lambda = create_function('$a,$b', 'return(strlen($a) - strlen($b));');
$array = array('really long string here, boy', 'this', 'middling length', 'larger');
```

```
usort($array, $lambda);
print_r($array);
```

The array is sorted by usort( ), using the anonymous function, in order of string length.

## 4.1. Quoting String Constants

There are three ways to write a literal string in your program: using single quotes, double quotes, and the here document (*heredoc*) format derived from the Unix shell. These methods differ in whether they recognize special *escape sequences* that let you encode other characters or interpolate variables.

The general rule is to use the least powerful quoting mechanism necessary. In practice, this means that you should use single-quoted strings unless you need to include escape sequences or interpolate variables, in which case you should use double-quoted strings. If you want a string that spans many lines, use a heredoc.

### 4.1.1. 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 string with the values of those variables. There are two ways to interpolate variables into strings—the simple way and the complex way.

The simple way is to just put the variable name in a double-quoted string or heredoc:

```
$who = 'Kilroy';
$where = 'here';
echo "$who was $where";
Kilroy was here
```

The complex way is to surround the variable being interpolated with curly braces. This method can be used either to disambiguate or to interpolate array lookups. The classic use of curly braces is to separate the variable name from surrounding text:

```
$n = 12;
echo "You are the {$n}th person";
You are the 12th person
```

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

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

```
$bar = 'this is not printed';
$foo = '$bar';     // single quotes
print("$foo");
$bar
```

### 4.1.2. Single-Quoted Strings

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

```
$name = 'Fred';
$str  = 'Hello, $name';     // single-quoted
echo $str;
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. Any other occurrence of a backslash is interpreted simply as a backslash:

```
$name = 'Tim O\'Reilly';    // escaped single quote
echo $name;
$path = 'C:\\WINDOWS';      // escaped backslash
echo $path;
$nope = '\n';               // not an escape
echo $nope;
Tim O'Reilly
C:\WINDOWS
\n
```

### 4.1.3. Double-Quoted Strings

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

*Table 4-1. Escape sequences in double-quoted strings*

| Escape sequence | Character represented |
|---|---|
| \" | Double quotes |
| \n | Newline |
| \r | Carriage return |
| \t | Tab |
| \\ | Backslash |
| \$ | Dollar sign |

| | |
|---|---|
| \{ | Left brace |
| \} | Right brace |
| \[ | Left bracket |
| \] | Right bracket |
| \0 through \777 | ASCII character represented by octal value |
| \x0 through \xFF | ASCII character represented by hex value |

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):

```
$str = "What is \c this?";      // unknown escape sequence
echo $str ;
What is \c this?
```

#### 4.1.4. Here Documents

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

```
$clerihew = <<< End_Of_Quote
Sir Humphrey Davy
Abominated gravy.
He lived in the odium
Of having discovered sodium.
End_Of_Quote;
echo $clerihew;
Sir Humphrey Davy
Abominated gravy.
He lived in the odium
Of having discovered sodium.
```

The <<< *Identifier* tells the PHP parser that you're writing a heredoc. There must be a space after the <<< and before the identifier. You get to pick the identifier. The next line starts the text being quoted by the heredoc, which continues until it reaches a line that consists of nothing but the identifier.

As a special case, you can put a semicolon after the terminating identifier to end the statement, as shown in the previous code. If you are using a heredoc in a more complex expression, you 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 passed through:

```
$dialogue = <<< No_More
"It's not going to happen!" she fumed.
He raised an eyebrow.  "Want to bet?"
No_More;
echo $dialogue;
"It's not going to happen!" she fumed.
He raised an eyebrow.  "Want to bet?"
```

Whitespace in a heredoc is also preserved:

```
$ws = <<< Enough
  boo
  hoo
Enough;
// $ws = "  boo\n  hoo\n";
```

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

```
$s = 'Foo';
// same as
$s = <<< End_of_pointless_heredoc
Foo
End_of_pointless_heredoc;
```

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

```
$s = <<< End
Foo
End;
```

### 4.2. Printing Strings

There are four ways to send output to the browser. The echo 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.

#### 4.2.1. 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 are equivalent:

```
echo "Printy";
echo("Printy");            // also valid
```

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

```
echo "First", "second", "third";
```
**Firstsecondthird**

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

```
// this is a parse 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!");
}
```

Such errors are easily remedied, though, by using the print( ) or printf( ) functions.

#### 4.2.2. print( )

The print( ) function sends one value (its argument) to the browser. It returns true if the string was successfully displayed and false otherwise (e.g., if the user pressed the Stop button on her browser before this part of the page was rendered):

```
if (! print("Hello, world")) {
  die("you're not listening to me!");
}
```
**Hello, world**

#### 4.2.3. 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 in. A % character in the format string indicates a substitution.

##### 4.2.3.1. Format modifiers

Each substitution marker in the template consists of a percent sign (%), possibly 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:

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

- The minimum number of characters that this element should contain. If the result is less than this number of characters, the sign and padding specifier govern how to pad to this length.
- 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.

#### 4.2.3.2. 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.

***Table 4-2. printf( ) type specifiers***

| Specifier | Meaning |
|---|---|
| B | The argument is an integer and is displayed as a binary number. |
| C | The argument is an integer and is displayed as the character with that value. |
| d or I | The argument is an integer and is displayed as a decimal number. |
| e, E, or f | The argument is a double and is displayed as a floating-point number. |
| g or G | The argument is a double with precision and is displayed as a floating-point number. |
| O | The argument is an integer and is displayed as an octal (base-8) number. |
| S | The argument is a string and is displayed as such. |
| U | The argument is an unsigned integer and is displayed as a decimal number. |
| x | The argument is an integer and is displayed as a hexadecimal (base-16) number; lowercase letters are used. |
| X | The argument is an integer and is displayed as a hexadecimal (base-16) number; uppercase letters are used. |



The printf( ) function looks outrageously complex to people who aren't C programmers. Once you get used to it, though, 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/%04y', $month, $day, $year);

**02/15/2002**

- 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 spent $ 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
```

### 4.2.4. print_r( ) and var_dump( )

The print_r( ) construct 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.

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);       print "\n";
1
print_r(false);      print "\n";
print_r(null);       print "\n";
```

For this reason, var_dump( ) is preferable to print_r( ) for debugging. The var_dump( ) function displays any PHP value in a human-readable format:

```
var_dump(true);
bool(true)
var_dump(false);
bool(false);
var_dump(null);
bool(null);
var_dump(array('name' => Fred, 'age' => 35));
array(2)
{
  ["name"]=>string(4) "Fred" ["age"]=>int(35)
}
class P
{
  var $name = 'Nat';
  // ...
}
$p = new P;
var_dump($p);
object(p)(1)
{
  ["name"]=>string(3) "Nat"
}
```

Beware of using print_r( ) or var_dump( ) on a recursive structure such as $GLOBALS (which has an entry for GLOBALS that points back to itself ). The print_r( ) function loops infinitely, while var_dump( ) cuts off after visiting the same element three times.

## 4.3. 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 array 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]);
}
The 0th character is H
The 1th character is e
The 2th character is l
The 3th character is l
The 4th character is o
```

## 4.4. Cleaning Strings

Often, the strings we get from files or users 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).

### 4.4.1. 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 of *`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.

***Table 4-3. Default characters removed by trim( ), ltrim( ), and rtrim( )***

| Character | ASCII value | Meaning |
|---|---|---|
| `" "` | 0x20 | Space |

| | | |
|---|---|---|
| `"\t"` | 0x09 | Tab |
| `"\n"` | 0x0A | Newline (line feed) |
| `"\r"` | 0x0D | Carriage return |
| `"\0"` | 0x00 | NUL-byte |
| `"\x0B"` | 0x0B | Vertical tab |

For example:

```
$title = "   Programming PHP  \n";
$str_1 = ltrim($title);               // $str_1 is "Programming PHP  \n"
$str_2 = rtrim($title);               // $str_2 is "   Programming PHP"
$str_3 = trim($title);                // $str_3 is "Programming PHP"
```

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

```
$record = "  Fred\tFlintstone\t35\tWilma   \n";
$record = trim($record, " \r\n\0\x0B");
// $record is "Fred\tFlintstone\t35\tWilma"
```

#### 4.4.2. 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));
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, use a combination of `strtolower( )` and `ucwords( )`:

```
print(ucwords(strtolower($string1)));
Fred Flintstone
```

## 4.5. 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 page 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 <. PHP has a number of built-in functions to convert to and from these encodings.

### 4.5.1. HTML

Special characters in HTML are represented by *entities* such as & and <. There are two PHP functions for turning special characters in a string into their entities, one for removing HTML tags, and one for extracting only meta tags.

#### 4.5.1.1. Entity-quoting all special characters

The htmlspecialchars( ) 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("Einsturzende Neubauten");
echo $string;
Einstürzende Neubauten
```

The entity-escaped version (ü) correctly displays as ü in the web page. As you can see, the space has not been turned into  .

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

```
$output = htmlentities(input, quote_style, charset);
```

The *charset* parameter, if given, identifies the character set. The default is "ISO-8859-1".
The *quote_style* parameter controls whether single and double quotes are turned into their entity forms. ENT_COMPAT (the default) converts only 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;
```

#### 4.5.1.2. 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 &
- Double quotes (") are converted to "
- Single quotes (') are converted to ' (if ENT_QUOTES is on, as described for htmlentities( ))
- Less-than signs (<) are converted to <
- Greater-than signs (>) are converted to >

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, and you'll have a garbled page.

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

```
$output = htmlspecialchars(input, [quote_style, [charset]]);
```

The *quote_style* and *charset* 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 ofhtmlentities( ):

```
$str = htmlentities("Einstürzende Neubauten");  // now it is encoded
$table = get_html_translation_table(HTML_ENTITIES);
```

```
$rev_trans = array_flip($table);
echo strtr($str,$rev_trans);  // back to normal
```
**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 wantedhtmlentities( ) to also encode spaces to  s, you would do:

```
$table = get_html_translation_table(HTML_ENTITIES);
$table[' '] = '&amp;nbsp;';
$encoded = strtr($original, $table);
```

#### 4.5.1.3. Removing HTML tags

The strip_tags( ) function removes HTML tags from a string:

```
$input  = '<p>Howdy, &amp;quot;Cowboy&amp;quot;</p>';
$output = strip_tags($input);
// $output is 'Howdy, &amp;quot;Cowboy&amp;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.

#### 4.5.1.4. Extracting meta tags

If you have the HTML for a web page in a string, the get_meta_tags( ) function returns an array of the meta tags in that page. The name of the meta tag (keywords,author, description, etc.) becomes the key in the array, and the content of the meta tag becomes the corresponding value:

```
$meta_tags = get_meta_tags('http://www.example.com/');
echo "Web page made by {$meta_tags[author]}";
```
**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.

### 4.5.2. URLs

PHP provides 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 1738) 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 is used in building query strings.

Note that you don't want to use these functions on a complete URL,
like http://www.example.com/hello, as they will escape the colons and slashes to producehttp%3A%2F%2Fwww.example.com%2Fhello. Only encode partial URLs (the bit after http://www.example.com/), and add the protocol and domain name later.

#### 4.5.2.1. RFC 1738 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 convention.

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";
http://localhost/Programming%20PHP
```

The rawurldecode( ) function decodes URL-encoded strings:

```
$encoded = 'Programming%20PHP';
echo rawurldecode($encoded);
Programming PHP
```

#### 4.5.2.2. 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 building query strings and cookie values, but because these values are automatically decoded when they are passed through a form or cookie, you don't need to use these functions to process the current page's query string or cookies. The functions are useful for generating query strings:

```
$base_url = 'http://www.google.com/q=';
$query = 'PHP sessions -cookies';
$url = $base_url . urlencode($query);
echo $url;
http://www.google.com/q=PHP+sessions+-cookies
```

### 4.5.3. 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:

```
$string = <<< The_End
"It's never going to work," she cried,
as she hit the backslash (\\) key.
The_End;
echo addslashes($string);
\"It\'s never going to work,\" she cried,
as she hit the backslash (\\) key.
echo stripslashes($string);
"It's never going to work," she cried,
as she hit the backslash (\) key.
```

Some databases escape single quotes with another single quote instead of a backslash. For those databases, enable magic_quotes_sybase in your *php.ini* file.

### 4.5.4. C-String Encoding

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

***Table 4-4. Single-character escapes recognized by addcslashes( ) and stripcslashes( )***

| ASCII value | Encoding |
|---|---|
| 7 | \a |
| 8 | \b |
| 9 | \t |
| 10 | \n |
| 11 | \v |
| 12 | \f |
| 13 | \r |

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 the ".." construct:

```
echo addcslashes("hello\tworld\n", "\x00..\x1fz..\xff");
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', etc. 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');
// $string is "hello\tworld\n"
```

## 4.6. Comparing Strings

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

### 4.6.1. Exact Comparisons

You can compare two strings for equality with the == and === operators. These operators differ in how they deal with non-string operands. The == operator casts non-string operands to strings, so it reports that 3 and "3" are equal. The === operator does not cast, and returns false if the types of the arguments 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";
}
```

```
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");
}
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, giving 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);
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 forstrcmp( ):

```
$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 **strnatcasecmp( ),** which take the same arguments as strcmp( ) and return the same kinds 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.

***Table 4-5. Natural order versus ASCII order***

| Natural order | ASCII order |
|---|---|

| | |
|---|---|
| pic1.jpg | pic1.jpg |
| pic5.jpg | pic10.jpg |
| pig10.jpg | pic5.jpg |
| pic50.jpg | pic50.jpg |

### 4.6.2. Approximate Equality

PHP provides several functions that let you test whether two strings are approximately equal: soundex( ) , metaphone( ), similar_text(), and levenshtein( ).

```
$soundex_code = soundex($string);
$metaphone_code = metaphone($string);
$in_common = similar_text($string_1, $string_2 [, $percentage ]);
$similarity = levenshtein($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 $query<br>";
} else {
  print "soundex: $known doesn't sound like $query<br>";
}
if (metaphone($known) == metaphone($query)) {
  print "metaphone: $known sounds $query<br>";
} else {
  print "metaphone: $known doesn't sound like $query<br>";
}
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, is a variable in which to store the commonality as a percentage:

```
$string_1 = "Rasmus Lerdorf";
$string_2 = "Razmus Lehrdorf";
$common = similar_text($string_1, $string_2, $percent);
printf("They have %d chars in common (%.2f%%).", $common, $percent);
```

```
They have 13 chars in common (89.66%).
```

The Levenshtein algorithm calculates the similarity of two strings based on how many characters you must add, substitute, or remove to make them the same. For 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 thelevenshtein( ) 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);
```

## 4.7. 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 only 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").

### 4.7.1. Substrings

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

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

The *start* argument is the position in *string* at which to begin copying, with 0 meaning the start of 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 times a smaller string occurs in a larger one, use substr_count( ):

```
$number = substr_count(big_string, small_string);
```

For example:

```
$sketch = <<< End_of_Sketch
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;
End_of_Sketch;
$count = substr_count($sketch, "spam");
print("The word spam occurs $count times.");
```
**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:

```
$greeting = "good morning citizen";
$farewell = substr_replace($greeting, "bye", 5, 7);
// $farewell is "good bye citizen"
```

Use a *length* value of 0 to insert without deleting:

```
$farewell = substr_replace($farewell, "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"
```

### 4.7.2. Miscellaneous String Functions

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

```
$string = strrev(string);
```

For example:

```
echo strrev("There is no cabal");
labac on si erehT
```

The str_repeat( ) 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 horizontal rule:

```
echo str_repeat('-', 40);
```

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 either 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";
[             Fred Flintstone]
[      Fred Flintstone       ]
```

## 4.7.3. 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( ).

### 4.7.3.1. Exploding and imploding

Data often arrives as strings, which must be broken down into an array of values. For instance, you might want to separate out the comma-separated fields from a string such as "Fred,25,Wilma". In these 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 large 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( ).

#### 4.7.3.2. Tokenizing

The strtok( ) function lets you iterate through a string, 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:

```
$first_chunk = strtok(string, separator);
```

To retrieve the rest of the tokens, repeatedly call strtok( ) with only the separator:

```
$next_chunk  = strtok(separator);
```

For instance, consider this invocation:

```
$string = "Fred,Flintstone,35,Wilma";
$token  = strtok($string, ",");
while ($token !== false)
{
  echo("$token<br>");
  $token = strtok(",");
}
```

**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.

#### 4.7.3.3. 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);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";
Fred Flintstone is 35 years old
```

### 4.7.4. String-Searching Functions

Several functions find a string or character within a larger string. They come in three families: strpos( ) 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, these function calls are identical because 44 is the ASCII value of the comma:

```
$pos = strpos($large, ",");          // find last comma
$pos = strpos($large, 44);           // find last comma
```

All the string-searching functions return false if they can't find the substring you specified. If the substring occurs at the start of the string, the functions return 0. Becausefalse 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
}
```

#### 4.7.4.1. Searches returning position

The strpos( ) 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, strpos( ) 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 strpos( ).

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**

If you pass a string as the second argument to strrpos( ), only the first character is searched for. To find the last occurrence of a multicharacter string, reverse the strings and use strpos( ):

```
$long = "Today is the day we go on holiday to Florida";
$to_find = "day";
$pos = strpos(strrev ($long), strrev($to_find));
if ($pos === false) {
  echo("Not found");
} else {
  // $pos is offset into reversed strings
  // Convert to offset into regular strings
  $pos = strlen($long) - $pos - strlen($to_find);;
  echo("Last occurrence starts at position $pos");
}
```

**Last occurrence starts at position 30**

#### 4.7.4.2. 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:

```
$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( )**

Find last occurrence of a character in a string

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

### 4.7.4.3. Searches using masks

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

```
$length = strspn(string, charset);
```

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

```
function is_octal ($str)
{
  return strspn($str, '01234567') == strlen($str);
}
```

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

```
function has_bad_chars ($str)
{
  return strcspn($str, "\n\t\0");
}
```

### 4.7.4.4. 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);
```

## Regular Expressions

A regular expression is 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 functions tell you whether there was a match, while others make changes to the string.

PHP provides support for two different types of regular expressions:

POSIX and Perl-compatible. POSIX regular expressions are less powerful, and sometimes slower, than the Perl-compatible functions, but can be easier to read.

There are three uses for regular expressions:

1. matching, which can also be used to extract information from a string;

2. substituting new text for matching text; and

3. splitting a string into an array of smaller chunks.

PHP has functions for all three behaviors for both Perl and POSIX regular expressions. For instance, ereg( ) does a POSIX match, while preg_match( ) does a Perl match. Fortunately, there are a number of similarities between basic POSIX and Perl regular expressions.

### 4.8.1. The Basics

Most characters in a regular expression are literal characters, meaning that they match only themselves. For instance, if you search for the regular expression "cow" in the string "Dave was a cowhand", you get a match because "cow" occurs in that string.

Some characters, though, 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):

```
ereg('^cow', 'Dave was a cowhand');     // returns false
ereg('^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):

```
ereg('cow$', 'Dave was a cowhand');     // returns false
ereg('cow$', "Don't have a cow");       // returns true
```

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

```
ereg('c.t', 'cat');                     // returns true
ereg('c.t', 'cut');                     // returns true
ereg('c.t', 'c t');                     // returns true
ereg('c.t', 'bat');                     // returns false
ereg('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:

```
ereg('\$5\.00', 'Your bill is $5.00 exactly');     // returns true
ereg('$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 POSIX-style match, you can use the eregi( ) function. With Perl-style regular expressions, you still use preg_match( ), but specify a flag to indicate a case-insensitive match (as you'll see when we discuss Perl-style regular expressions in detail later in this chapter).

So far, we haven't done anything we couldn't have done with the string functions we've already 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 characters, 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 no 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.

### 4.8.2. Character Classes

To specify a set of acceptable characters in your pattern, you can either 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:

```
ereg('c[aeiou]t', 'I cut my hand');     // returns true
ereg('c[aeiou]t', 'This crusty cat');   // returns true
ereg('c[aeiou]t', 'What cart?');        // returns false
ereg('c[aeiou]t', '14ct gold');         // returns false
```

The regular expression engine finds a "c", then checks that the next 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 looking for another "c". If a vowel is found, though, the engine then checks that the next character is a "t". If it is, the engine is at the end of the match and so returns true. If the next character isn't a "t", the engine goes back to looking for another "c".

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

```
ereg('c[^aeiou]t', 'I cut my hand');    // returns false
ereg('c[^aeiou]t', 'Reboot chthon');    // returns true
ereg('c[^aeiou]t', '14ct gold');        // returns 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":

```
ereg('[0-9]%', 'we are 25% complete');          // returns true
ereg('[0123456789]%', 'we are 25% complete');   // returns true
ereg('[a-z]t', '11th');                         // returns false
ereg('[a-z]t', 'cat');                          // returns true
ereg('[a-z]t', 'PIT');                          // returns false
ereg('[a-zA-Z]!', '11!');                       // returns false
ereg('[a-zA-Z]!', 'stop!');                     // returns true
```

When you are specifying a character class, some special characters lose their meaning, while others take on new meaning. In particular, 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. For instance, [^\]] matches any character that is not a closing bracket, while [$.^] matches any dollar sign, period, or caret.

The various regular expression libraries define shortcuts for character classes, including digits, alphabetic characters, and whitespace. The actual syntax for these shortcuts differs between POSIX-style and Perl-style regular expressions. For instance, with POSIX, the whitespace character class is "[[:space:]]", while with Perl it is "\s".

### 4.8.3. Alternatives

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

```
ereg('cat|dog', 'the cat rubbed my legs');      // returns true
ereg('cat|dog', 'the dog rubbed my legs');      // returns true
ereg('cat|dog', 'the rabbit rubbed my legs');   // returns false
```

The precedence of alternation can be a surprise: '^cat|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 regular expression '^(cat|dog)$'.

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

```
ereg('^([a-z]|[0-9])', 'The quick brown fox');  // returns false
ereg('^([a-z]|[0-9])', 'jumped over');          // returns true
ereg('^([a-z]|[0-9])', '10 lazy dogs');         // returns true
```

### 4.8.4. Repeating Sequences

To specify a repeating pattern, you use something called a *quantifier*. The quantifier goes after the pattern that's repeated and says how many times to repeat that pattern. Table 4-6 shows the quantifiers that are supported by both POSIX and Perl regular expressions.

*Table 4-6. Regular expression quantifiers*

| Quantifier | Meaning |
|---|---|
| ? | 0 or 1 |
| * | 0 or more |
| + | 1 or more |
| {*n*} | Exactly *n* times |
| {*n*,*m*} | At least *n*, no more than *m* times |
| {*n*,} | At least *n* times |

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

```
ereg('ca+t', 'caaaaaaat');          // returns true
ereg('ca+t', 'ct');                 // returns false
ereg('ca?t', 'caaaaaaat');          // returns false
ereg('ca*t', 'ct');                 // returns true
```

With quantifiers and character classes, we can actually do something useful, like matching valid U.S. telephone numbers:

```
ereg('[0-9]{3}-[0-9]{3}-[0-9]{4}', '303-555-1212');     // returns true
ereg('[0-9]{3}-[0-9]{3}-[0-9]{4}', '64-9-555-1234');    // returns false
```

### 4.8.5. Subpatterns

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

```
ereg('a (very )+big dog', 'it was a very very big dog'); // returns true
ereg('^(cat|dog)$', 'cat');                 // returns true
ereg('^(cat|dog)$', '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 substrings:

```
ereg('([0-9]+)', 'You have 42 magic beans', $captured);
// returns true and populates $captured
```

The zeroth element of the array is set to 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.

### 4.9. POSIX-Style Regular Expressions

Now that you understand the basics of regular expressions, we can explore the details. POSIX-style regular expressions use the Unix locale system. The locale system provides functions for sorting and identifying characters that let you intelligently work with text from languages other than English. In particular, what constitutes a "letter" varies from language to language (think of à and ç), and there are character classes in POSIX regular expressions that take this into account.

However, POSIX regular expressions are designed for use with only textual data. If your data has a NUL-byte (\x00) in it, the regular expression functions will interpret it as the end of the string, and matching will not take place beyond that point. To do matches against arbitrary binary data, you'll need to use Perl-compatible regular expressions, which are discussed later in this chapter. Also, as we already mentioned, the Perl-style regular expression functions are often faster than the equivalent POSIX-style ones.

#### 4.9.1. Character Classes

As shown in Table 4-7, POSIX defines a number of named sets of characters that you can use in character classes. The expansions given in Table 4-7 are for English. The actual letters vary from locale to locale.

***Table 4-7. POSIX character classes***

| Class | Description | Expansion |
|---|---|---|
| [:alnum:] | Alphanumeric characters | [0-9a-zA-Z] |
| [:alpha:] | Alphabetic characters (letters) | [a-zA-Z] |
| [:ascii:] | 7-bit ASCII | [\x01-\x7F] |
| [:blank:] | Horizontal whitespace (space, tab) | [ \t] |
| [:cntrl:] | Control characters | [\x01-\x1F] |
| [:digit:] | Digits | [0-9] |
| [:graph:] | Characters that use ink to print (non-space, non-control) | [^\x01-\x20] |
| [:lower:] | Lowercase letter | [a-z] |
| [:print:] | Printable character (graph class plus space and tab) | [\t\x20-\xFF] |
| [:punct:] | Any punctuation character, such as the period (.) and the semicolon (;) | [-!"#$%&'( )*+,./:;<=>?@[\\]^_`{\|}~] |

| | | |
|---|---|---|
| [:space:] | Whitespace (newline, carriage return, tab, space, vertical tab) | [\n\r\t \x0B] |
| [:upper:] | Uppercase letter | [A-Z] |
| [:xdigit:] | Hexadecimal digit | [0-9a-fA-F] |

Each [:*something*:] class can be 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 as the endpoint of a range:

```
ereg('[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 this character class:

```
[st[.ch.]]
```

The final POSIX extension to character classes is the *equivalence class*, specified by enclosing the character in [= 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=].

### 4.9.2. 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 POSIX regular expressions.

***Table 4-8. POSIX anchors***

| Anchor | Matches |
|---|---|
| ^ | Start of string |
| $ | End of string |
| [[:<:]] | Start of word |
| [[:>:]] | End of word |

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

```
ereg('[[:<:]]gun[[:>:]]', 'the Burgundy exploded');    // returns false
ereg('gun', 'the Burgundy exploded');                 // returns true
```

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

#### 4.9.3. Functions

There are three categories of functions for POSIX-style regular expressions: matching, replacing, and splitting.

##### 4.9.3.1. Matching

The ereg( ) function takes a pattern, a string, and an optional array. It populates the array, if given, and returns true or false depending on whether a match for the pattern was found in the string:

```
$found = ereg(pattern, string [, captured ]);
```

For example:

```
ereg('y.*e$', 'Sylvie');       // returns true
ereg('y(.*)e$', 'Sylvie', $a); // returns true, $a is array('Sylvie', 'lvi')
```

The zeroth element of the array is set to the entire string being matched against. The first element is the substring that matched the first subpattern, the second element is the substring that matched the second subpattern, and so on.

The eregi( ) function is a case-insensitive form of ereg( ). Its arguments and return values are the same as those for ereg( ).

Example 4-1 uses pattern matching to determine whether a credit-card number passes the Luhn checksum and whether the digits are appropriate for a card of a specific type.

***Example 4-1. Credit-card validator***

```
// The Luhn checksum determines whether a credit-card number is syntactically
// correct; it cannot, however, tell if a card with the number has been issued,
// is currently active, or has enough space left to accept a charge.

function IsValidCreditCard($inCardNumber, $inCardType)
{
  // Assume it's okay
  $isValid = true;

  // Strip all non-numbers from the string
  $inCardNumber = ereg_replace('[^[:digit:]]','', $inCardNumber);

  // Make sure the card number and type match
  switch($inCardType) {
    case 'mastercard':
      $isValid = ereg('^5[1-5].{14}$', $inCardNumber);
      break;
```

```
  case 'visa':
    $isValid = ereg('^4.{15}$|^4.{12}$', $inCardNumber);
    break;

  case 'amex':
    $isValid = ereg('^3[47].{13}$', $inCardNumber);
    break;

  case 'discover':
    $isValid = ereg('^6011.{12}$', $inCardNumber);
    break;

  case 'diners':
    $isValid = ereg('^30[0-5].{11}$|^3[68].{12}$', $inCardNumber);
    break;

    case 'jcb':
    $isValid = ereg('^3.{15}$|^2131|1800.{11}$', $inCardNumber);
    break;
}

// It passed the rudimentary test; let's check it against the Luhn this time
if($isValid)
{
  // Work in reverse
  $inCardNumber = strrev($inCardNumber);

  // Total the digits in the number, doubling those in odd-numbered positions
  $theTotal = 0;
  for ($i = 0; $i < strlen($inCardNumber); $i++)
  {
    $theAdder = (int) $inCardNumber{$i};

    // Double the numbers in odd-numbered positions
    if($i % 2)
  {
      $theAdder << 1;
      if($theAdder > 9) { $theAdder -= 9; }
    }

    $theTotal += $theAdder;
  }

  // Valid cards will divide evenly by 10
  $isValid = (($theTotal % 10) == 0);
}

return $isValid;
```

```
}
```

#### 4.9.3.2. Replacing

The ereg_replace( ) function takes a pattern, a replacement string, and a string in which to search. It returns a copy of the search string, with text that matched the pattern replaced with the replacement string:

```
$changed = ereg_replace(pattern, replacement, string);
```

If the pattern has any grouped subpatterns, the matches are accessible by putting the characters \1 through \9 in the replacement string. For example, we can useereg_replace( ) to replace characters wrapped with [b] and [/b] tags with equivalent HTML tags:

```
$string = 'It is [b]not[/b] a matter of diplomacy.';
echo ereg_replace ('\[b]([^]]*)\[/b]', '<b>\1</b>', $string);
It is <b>not</b> a matter of diplomacy.
```

The eregi_replace( ) function is a case-insensitive form of ereg_replace( ). Its arguments and return values are the same as those for ereg_replace( ).

#### 4.9.3.3. Splitting

The split( ) function uses a regular expression to divide a string into smaller chunks, which are returned as an array. If an error occurs, split( ) returns false. Optionally, you can say how many chunks to return:

```
$chunks = split(pattern, string [, limit ]);
```

The pattern matches the text that *separates* the chunks. For instance, to split out the terms from an arithmetic expression:

```
$expression = '3*5+i/6-12';
$terms = split('[/+*-]', $expression);
// $terms is array('3', '5', 'i', '6', '12)
```

If you specify a limit, the last element of the array holds the rest of the string:

```
$expression = '3*5+i/6-12';
$terms = split('[/+*-]', $expression, 3);
// $terms is array('3', '5', 'i'/6-12)
```

### 4.10. Perl-Compatible Regular Expressions

Perl has long been considered the benchmark for powerful regular expressions. PHP uses a C library called pcre to provide almost complete support for Perl's arsenal of regular expression features. Perl regular expressions include the POSIX classes and anchors described earlier. A POSIX-style character class in a Perl regular expression works and understands non-English

characters using the Unix locale system. Perl regular expressions act on arbitrary binary data, so you can safely match with patterns or strings that contain the NUL-byte (\x00).

### 4.10.1. Delimiters

Perl-style regular expressions emulate the Perl syntax for patterns, which means that each pattern must be enclosed in a pair of delimiters. Traditionally, the 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 when matching strings containing slashes, such as filenames. For example, the following are equivalent:

```
preg_match('/\/usr\/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
```

Section 4.10.8 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 engine strip whitespace and #-marked comments from 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'
```

### 4.10.2. Match Behavior

While Perl's regular expression syntax includes the POSIX constructs we talked about earlier, some pattern components have a different meaning in Perl. In particular, Perl's regular expressions are optimized for matching against single lines of text (although there are options that change this 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 my pants'
```

### 4.10.3. Character Classes

Perl-style regular expressions support the POSIX character classes but also define some of their own, as shown in Table 4-9.

*Table 4-9. Perl-style character classes*

| Character class | Meaning | Expansion |
|---|---|---|
| \s | Whitespace | [\r\n \t] |
| \S | Non-whitespace | [^\r\n \t] |
| \w | Word (identifier) character | [0-9A-Za-z_] |
| \W | Non-word (identifier) character | [^0-9A-Za-z_] |
| \d | Digit | [0-9] |
| \D | Non-digit | [^0-9] |

### 4.10.4. Anchors

Perl-style regular expressions also support additional anchors, as listed in Table 4-10.

*Table 4-10. Perl-style anchors*

| Assertion | Meaning |
|---|---|
| \b | Word boundary (between \w and \W or at start or end of string) |
| \B | Non-word boundary (between \w and \w, or \W and \W) |
| \A | Beginning of string |
| \Z | End of string or before \n at end |
| \z | End of string |
| ^ | Start of line (or after \n if /m flag is enabled) |
| $ | End of line (or before \n if /m flag is enabled) |

### 4.10.5. Quantifiers and Greed

The POSIX quantifiers, which Perl also supports, are always *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 (non-greedy) matching*—that is, quantifiers that match as few times as possible to satisfy the rest of the pattern. Perl provides a parallel set of quantifiers that match minimally. They're easy to remember, because they're the same as the greedy quantifiers, but with aquestion mark (?) appended. Table 4-11 shows the corresponding greedy and non-greedy quantifiers supported by Perl-style regular expressions.

***Table 4-11. Greedy and non-greedy quantifiers in Perl-compatible regular expressions***

| Greedy quantifier | Non-greedy quantifier |
|---|---|
| ? | ?? |
| * | *? |
| + | +? |
| {m} | {m}? |
| {m,} | {m,}? |
| {m,n} | {m,n}? |

Here's how to match a tag using a non-greedy quantifier:

```
preg_match('/(<.*?>)/', 'do <b>not</b> press the button', $match);
// $match[1] is '<b>'
```

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

```
preg_match('/(<[^>]*>)/', 'do <b>not</b> press the button', $match);
// $match[1] is '<b>'
```

### 4.10.6. Non-Capturing Groups

If you enclose a part of a pattern in parentheses, the text that matches that 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'
```

### 4.10.7. 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 second opening parenthesis, and so on.

For instance, this identifies doubled words:

```
preg_match('/([[:alpha:]]+)\s+\1/', 'Paris in the the spring', $m);
// returns true and $m[1] is 'the'
```

You can't capture more than 99 subpatterns.

### 4.10.8. 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!');        // returns true
```

Table 4-12 shows the modifiers from Perl that are supported in Perl-compatible regular expressions.

***Table 4-12. Perl flags***

| Modifier | Meaning |
|---|---|
| /*regexp*/i | Match case-insensitively. |
| /*regexp*/s | Make period (.) match any character, *including* newline (\n). |
| /*regexp*/x | Remove whitespace and comments from the pattern. |
| /*regexp*/m | Make caret (^) match after, and dollar sign ($) match before, internal newlines (\n). |
| /*regexp*/e | If the replacement string is PHP code, eval( ) it to get the actual replacement string. |

PHP's Perl-compatible regular expression functions also support other modifiers that aren't supported by Perl, as listed in Table 4-13.

***Table 4-13. Additional PHP flags***

| Modifier | Meaning |
|---|---|
| /*regexp*/U | Reverses the greediness of the subpattern; * and + now match as little as possible, instead of as much as possible |

| | |
|---|---|
| /*regexp*/u | Causes pattern strings to be treated as UTF-8 |
| /*regexp*/X | Causes a backslash followed by a character with no special meaning to emit an error |
| /*regexp*/A | Causes the beginning of the string to be anchored as if the first character of the pattern were ^ |
| /*regexp*/D | Causes the $ character to match only at the end of a line |
| /*regexp*/S | Causes the expression parser to more carefully examine the structure of the pattern, so it may run slightly faster the next time (such as in a loop) |

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

```
$message = <<< END
To: you@youcorp
From: me@mecorp
Subject: pay up

Pay me or else!
END;
preg_match('/^subject: (.*)/im', $message, $match);
// $match[1] is 'pay up'
```

#### 4.10.9. Inline Options

In addition to specifying patternwide 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:

```
preg_match('/I like (?i:PHP)/', 'I like pHp');          // returns true
```

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

```
preg_match('/eat (?ix:fo    o   d)/', 'eat FoOD');     // returns true
```

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

```
preg_match('/(?-i:I like) PHP/i', 'I like pHp');       // returns true
```

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 do that.

### 4.10.10. Look ahead and Look behind

It's sometimes useful in patterns to be able 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 make sure (without matching it, thus preventing it from being returned) that there's more data after the separator. Similarly, *lookbehind* checks the preceding text.

Lookahead and lookbehind come in two forms: *positive* and *negative*. A positive lookahead or lookbehind says "the next/preceding text must be like this." A negative lookahead or lookbehind says "the next/preceding text must *not* be like this." Table 4-14 shows the four constructs you can use in Perl-compatible patterns. None of the constructs captures text.

***Table 4-14. Lookahead and lookbehind assertions***

| Construct | Meaning |
|---|---|
| (?=subpattern) | Positive lookahead |
| (?!subpattern) | Negative lookahead |
| (?<=subpattern) | Positive lookbehind |
| (?<!subpattern) | Negative lookbehind |

A simple use of positive lookahead is splitting a Unix mbox mail file into individual messages. The word "From" starting a line by itself indicates the start of a new message, so you can split the mailbox into messages by specifying the separator as the point where the next text is "From" at the start of a line:

```
$messages = preg_split('/(?=^From )/m', $mailbox);
```

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];
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 expressions. 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 lengths.

### 4.10.11. Cut

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

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

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

This code snippet takes 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.

### 4.10.12. 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 succeeds, the regular expression engine matches the *yespattern*. With the second form, if the assertion doesn't succeed, the regular expression engine skips the *yespattern* and tries to match the *nopattern*.

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 1-99 (the most backreferences available). The condition uses the pattern in the assertion only if the backreference was matched. If the assertion is not a backreference, it must be a positive or negative lookahead or lookbehind assertion.

### 4.10.13. 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.

#### 4.10.13.1. 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( ) function takes the same arguments and gives the same return value as the ereg( ) 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('Sylvie', 'lvi')
```

While there's an eregi( ) function to match case-insensitively, there's no preg_matchi( ) function. Instead, use the i flag on the pattern:

```
preg_match('y.*e$/i', 'SyLvIe');        // 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, $m2, 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 elements than subpatterns.

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

Example 4-2 fetches the HTML at a particular web address into a string and extracts the URLs from that HTML. For each URL, it generates a link back to the program that will display the URLs at that address.

***Example 4-2. Extracting URLs from an HTML page***

```
<?php
 if (getenv('REQUEST_METHOD') == 'POST')
{
  $url = $_POST[url];
 }
else
{
  $url = $_GET[url];
 }
?>

<form action="<?php $PHP_SELF ?>" method="POST">
URL: <input type="text" name="url" value="<?php $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|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 valid
                    #   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 = $PHP_SELF . '?url=' . urlencode($u);
    echo "<A HREF='$link'>$u</A><BR>\n";
  }
?>
```

#### 4.10.13.2. 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 ]);
```

The most common usage has all the argument strings, except for the integer *limit*. The limit is the maximum number of occurrences of the pattern to replace (the default, and the behavior when a limit of -1 is passed, is all occurrences).

```
$better = preg_replace('/<.*?>/', '!', 'do <b>not</b> press the button');
// $better is 'do !not! press the button'
```

Pass an array of strings as *subject* to make the substitution on all of 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 = "Please 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 jump 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:

```
$html_gunk = array('/<.*?>/', '/&.*?;/');
$html = '&eacute; : <b>very</b> cute';
$stripped  = preg_replace($html_gunk, 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:

```
$stripped = preg_replace($html_gunk, '', $html);
```

The replacement can use backreferences. Unlike backreferences in patterns, though, the preferred syntax for backreferences in replacements is $1, $2, $3, etc.

For example:

```
echo preg_replace('/(\w)\w+\s+(\w+)/', '$2, $1.', 'Fred Flintstone')
```

**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);
```

**It 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 executed is $name and $age. That code evaluates to the value of the variable, which is what's used as the replacement. Whew!

### 4.10.13.3. 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 chunks. By default, the separators are not returned. The optional *limit* specifies the maximum number of chunks to return (-1 is the default, which means all chunks). The *flags* argument is a bitwise OR combination of the

flags PREG_SPLIT_NO_EMPTY (empty chunks are not returned)
and PREG_SPLIT_DELIM_CAPTURE (parts of the string captured in the pattern are returned).

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

```
$ops = preg_split('{[+*/-]}', '3+5*9/2');
// $ops is array('3', '5', '9', '2')
```

To extract the operands and the operators, use:

```
$ops = preg_split('{([+*/-])}', '3+5*9/2', -1, PREG_SPLIT_DELIM_CAPTURE);
// $ops is array('3', '+', '5', '*', '9', '/', '2')
```

An empty pattern matches at every boundary between characters in the string. This lets you split a string into an array of characters:

```
$array = preg_split('//', $string);
```

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 text that matched the pattern, the first is the contents of the first captured subpattern, and so on). For example:

```
function titlecase ($s)
{
  return ucfirst(strtolower($s[0]));
}

$string = 'goodbye cruel world';
$new = preg_replace_callback('/\w+/', 'titlecase', $string);
echo $new;
Goodbye Cruel World
```

#### 4.10.13.4. 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);
```

#### 4.10.13.5. 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)');
\$5\.00 \(five bucks\)
```

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

```
$to_find = '/usr/local/etc/rsync.conf';
$re = preg_quote($filename, '/');
if (preg_match("/$re", $filename))
{
  // found it!
}
```



### 4.10.14. 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 ?.
- Parenthesized submatches within negative assertions are not remembered.
- Alternation branches within a lookbehind assertion can be of different lengths.

## PHP - Regular Expressions

Regular expressions are nothing more than a sequence or pattern of characters itself. They provide the foundation for pattern-matching functionality.

Using regular expression you can search a particular string inside a another string, you can replace one string by another string and you can split a string into many chunks.

PHP offers functions specific to two sets of regular expression functions, each corresponding to a certain type of regular expression. You can use any of them based on your comfort.

- POSIX Regular Expressions
- PERL Style Regular Expressions

## POSIX Regular Expressions

The structure of a POSIX regular expression is not dissimilar to that of a typical arithmetic expression: various elements (operators) are combined to form more complex expressions.

The simplest regular expression is one that matches a single character, such as g, inside strings such as g, haggle, or bag.

Explanation for few concepts being used in POSIX regular expression.

### Brackets

Brackets ([]) have a special meaning when used in the context of regular expressions. They are used to find a range of characters.

| Sr.No | Expression & Description |
|---|---|
| 1 | **[0-9]**<br>It matches any decimal digit from 0 through 9. |
| 2 | **[a-z]**<br>It matches any character from lower-case a through lowercase z. |
| 3 | **[A-Z]**<br>It matches any character from uppercase A through uppercase Z. |
| 4 | **[a-Z]**<br>It matches any character from lowercase a through uppercase Z. |

The ranges shown above are general; we can also use the range [0-3] to match any decimal digit ranging from 0 through 3, or the range [b-v] to match any lowercase character ranging from b through v.

### Quantifiers

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

| Sr.No | Expression & Description |
|---|---|
| 1 | **p+**<br>It matches any string containing at least one p. |
| 2 | **p***<br>It matches any string containing zero or more p's. |
| 3 | **p?**<br>It matches any string containing zero or more p's. This is just an alternative way to use p*. |
| 4 | **p{N}**<br>It matches any string containing a sequence of N p's |
| 5 | **p{2,3}**<br>It matches any string containing a sequence of two or three p's. |
| 6 | **p{2, }**<br>It matches any string containing a sequence of at least two p's. |
| 7 | **p$**<br>It matches any string with p at the end of it. |
| 8 | **^p**<br>It matches any string with p at the beginning of it. |

**Examples**

| Sr.No | Expression & Description |
|---|---|
| 1 | **[^a-zA-Z]**<br>It matches any string not containing any of the characters ranging from a through z and A through Z. |
| 2 | **p.p**<br>It matches any string containing p, followed by any character, in turn followed by another p. |
| 3 | **^.{2}$**<br>It matches any string containing exactly two characters. |
| 4 | **(.*)**<br>It matches any string enclosed within  and . |

| | |
|---|---|
| 5 | **p(hp)***<br>It matches any string containing a p followed by zero or more instances of the sequence php. |

**Predefined Character Ranges**

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

| Sr.No | Expression & Description |
|---|---|
| 1 | **[[:alpha:]]**<br>It matches any string containing alphabetic characters aA through zZ. |
| 2 | **[[:digit:]]**<br>It matches any string containing numerical digits 0 through 9. |
| 3 | **[[:alnum:]]**<br>It matches any string containing alphanumeric characters aA through zZ and 0 through 9. |
| 4 | **[[:space:]]**<br>It matches any string containing a space. |

**PHP's Regexp POSIX Functions**

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

| Sr.No | Function & Description |
|---|---|
| 1 | **ereg()**<br>The ereg() function searches a string specified by string for a string specified by pattern, returning true if the pattern is found, and false otherwise. |
| 2 | **ereg_replace()**<br>The ereg_replace() function searches for string specified by pattern and replaces pattern with replacement if found. |
| 3 | **eregi()**<br>The eregi() function searches throughout a string specified by pattern for a string specified by string. The search is not case sensitive. |
| 4 | **eregi_replace()** |

| | |
|---|---|
| | The eregi_replace() function operates exactly like ereg_replace(), except that the search for pattern in string is not case sensitive. |
| 5 | **split()**<br>The split() function will divide a string into various elements, the boundaries of each element based on the occurrence of pattern in string. |
| 6 | **spliti()**<br>The spliti() function operates exactly in the same manner as its sibling split(), except that it is not case sensitive. |
| 7 | **sql_regcase()**<br>The sql_regcase() function can be thought of as a utility function, converting each character in the input parameter string into a bracketed expression containing two characters. |

## PERL Style Regular Expressions

Perl-style regular expressions are similar to their POSIX counterparts. The POSIX syntax can be used almost interchangeably with the Perl-style regular expression functions. In fact, we can use any of the quantifiers introduced in the previous POSIX section.

### Meta characters

A meta character is simply an alphabetical character preceded by a backslash that acts to give the combination a special meaning.

For instance, we can search for large money sums using the '\d' meta character: **/([\d]+)000/**, Here **\d** will search for any string of numerical character.

Following is the list of meta characters which can be used in PERL Style Regular Expressions.

| Character | Description |
|---|---|
| . | a single character |
| \s | a whitespace character (space, tab, newline) |
| \S | non-whitespace character |
| \d | a digit (0-9) |
| \D | a non-digit |
| \w | a word character (a-z, A-Z, 0-9, _) |
| \W | a non-word character |
| [aeiou] | matches a single character in the given set |
| [^aeiou] | matches a single character outside the given set |
| (foo\|bar\|baz) | matches any of the alternatives specified |

### Modifiers

Several modifiers are available that can make your work with regexps much easier, like case sensitivity, searching in multiple lines etc.

| Modifier | Description |
|---|---|
| i | Makes the match case insensitive |
| m | Specifies that if the string has newline or carriage return characters, the ^ and $ operators will now match against a newline boundary, instead of a string boundary |
| o | Evaluates the expression only once |
| s | Allows use of . to match a newline character |
| x | Allows you to use white space in the expression for clarity |
| g | Globally finds all matches |
| cg | Allows a search to continue even after a global match fails |

**PHP's Regexp PERL Compatible Functions**

PHP offers following functions for searching strings using Perl-compatible regular expressions –

| Sr.No | Function & Description |
|---|---|
| 1 | **preg_match()**<br>The preg_match() function searches string for pattern, returning true if pattern exists, and false otherwise. |
| 2 | **preg_match_all()**<br>The preg_match_all() function matches all occurrences of pattern in string. |
| 3 | **preg_replace()**<br>The preg_replace() function operates just like ereg_replace(), except that regular expressions can be used in the pattern and replacement input parameters. |
| 4 | **preg_split()**<br>The preg_split() function operates exactly like split(), except that regular expressions are accepted as input parameters for pattern. |
| 5 | **preg_grep()**<br>The preg_grep() function searches all elements of input_array, returning all elements matching the regexp pattern. |
| 6 | **preg_ quote()**<br>Quote regular expression characters |