## Syntax

**PHP’s syntax is derived from many languages—predominantly the C language, but Perl has also had a lot of influence on its syntax. With the latest object-oriented additions, more Java-like syntax is creeping in as well. Despite incorporating elements of so many other languages, PHP’s syntax remains simple and easy to understand**

## **Source Files and PHP Tags**

Even though it is often used as a pure language, PHP is primarily designed as a text processor (hence its name). To facilitate this role, PHP code can be inserted directly into a text file using a special set of tags; the interpreter will then output any text outside the tags as-is, and execute the code that is between the tags. There are four types of tags available

|  |  |  |
| --- | --- | --- |
| Standard Tags | <?php  ... code  ?> | The default opening and closing tags; they are the best solution for portability and backwards compatibility, because they are guaranteed to be available and cannot be disabled by changing PHP’s configuration file. |
| Short Tags | <?  ... code  ?>  <?= $variable ?> | The standard in the PHP world; however, they do have the major drawback of conflicting with XML headers and, therefore, have somewhat fallen by the wayside. Their other advantage is the availability of the short form <?=$variable ?> syntax, which allows you to print the result of an expression directly to the script’s output. |
| Script Tags | <script language=“php”>  ... code  </script> | Were introduced so that HTML editors which were able to ignore JavaScript but were unable to cope with the standard PHP tags could also ignore the PHP code. |
| ASP Style Tags | <%  ... code  %> | Nobody quite understands why ASP tags were introduced—however, if you are so inclined you can turn on this optional configuration option, and you are free to use them. |

Short tags, script tags and ASP tags are all considered deprecated and their use is strongly discouraged.

## Newline Characters

It is important to remember that every character outside of PHP tags is copied as-is by the interpreter to the script’s output—and this includes newline characters. Newlines are, normally, ignored by browsers, as they are non-semantic characters in HTML. However, they are also used as separators between the header portion of a web server’s HTTP response and the actual data; therefore, outputting a newline character before all of the headers have been written to the output can cause some rather unpleasant (and unintended) consequences. To mitigate this problem, the first newline directly after a closing tag (?> only) is stripped by the parser. Doing so also solves a problem introduced by the fact that a number of popular text editors will automatically prepend a newline to the end of your file, thus interfering with include files which are not supposed to output any text. An easy way to prevent spurious output from an include file is to omit the closing tag at the end, which the parser considers this perfectly legal.

## Anatomy of a PHP Script

Every PHP script is made up of statements, like function calls, variable assignments, data output, directives, and so on. Except in very few cases, each of these instructions must be terminated—just like in C, Perl and JavaScript—with a semicolon. This requirement is not always strict—for example, the last instruction before a closing tag does not require a semicolon; however, these should be primarily considered quirks in the parser’s logic, and you should always terminate your instructions with a semicolon

## Comments

It is a good programming practice to comment every function, class, method or property in your code (although you will likely come across lots of code that is poorly commented—or not at all). Remember—any code that took thought to write will take thought to reread after several days, months or in some cases, years.

As with tags, PHP gives you multiple choices for your comments:

// Single line comment

# Single line comment

/

\*

Multi-line comment

\*

/

/

\*\*

\*

API Documentation Example

\*

\*

@param string $bar

\*

/

function foo($bar) { }

Both types of single line comments, // and #, can be ended using a newline (\r, \n or \r\n) or by ending the current PHP block using the PHP closing tag—?>.

Because the closing tag ?> will end a comment, code like // Do not show this ?> or this will output or this, which is not the intended behaviour.

## Whitespace

PHP is whitespace-insensitive, except in a few key areas. This means that there are no requirements to use (or not to use) a specific type of whitespace character (e.g.: tabs rather than spaces), or a particular number of whitespace characters. However, there are a few limitations

• You can’t have any whitespace between <? and php

• You cannot break apart keywords (e.g.: whi le, fo r, and funct ion)

• You cannot break apart variable names and function names, (e.g.: $var name and   
function foo bar())

## Code Block

A code block is simply a series of statements enclosed between two braces

{

// Some comments

f(); // a function call

}

Code blocks are handy for creating groups of script lines that must all be executed under specific circumstances, such as a function call or a conditional statement. Code blocks can be nested.

## Language Constructs

Constructs are elements that are built-into the language and, therefore, follow special rules. Perhaps the most common of them is the **echo** statement, which allows you to write data to the script’s output

echo 10; // will output 10

It’s important to understand that **echo is not a function** and, as such, it does not have a return value. If you need to output data as part of a more complex expression, you can use **print()** instead, which whilst also a language construct, **behaves like a function**, as it has a return value (which is always 1).

echo 10;

print (10);

Another very important construct is **die(),** which is itself an **alias of exit()**. It allows you to terminate the script’s output and either output a string or return a numeric status to the process that called the script.

## Data Types

PHP supports many different data types, but they are generally divided in two categories: scalar and composite. A scalar value contains only one value at a time. PHP supports four scalar types

**Boolean, a value that can only either be true or false**

**Int, a signed numeric integer value**

**Float, a signed floating-point value**

**String, a collection of binary data**

**Numeric values :** PHP recognizes two types of numbers, integers and floating-point values. The int data type is used to represent signed integers (meaning that both positive and negative numbers can be expressed with it). Numbers can be declared using several different notations:

**Decimal 10, -11, 1452** Standard decimal notation. Note that no thousand separator is needed—or, indeed, allowed.

**Octal 0666, 0100** Octal notation—identified by its leading zero and used mainly to express UNIX-style access permissions.

**Hexadecimal 0x123, 0XFF, -0x100** Base-16 notation; note that the hexadecimal digits and the leading 0x prefix are both case-insensitive.

It is important that you are well aware of the different notations—in particular, octal numbers can be easily confused with decimal numbers and can lead to some... interesting consequences!

Floating-point numbers (also called floats and, sometimes, doubles) are numbers that have a fractional component; like integers, they are also signed. PHP supports two different notations for expressing them

Decimal 0.12; 1234.43; -.123 Traditional decimal notation.

Exponential 2E7, 1.2e2 Exponential notation—a set of significant digits (also called the mantissa), followed by the case-insensitive letter E and by an exponent. The resulting number is expressed multiplied by ten to the power of the exponent—for example, 1e2 equals 100.

There are a few important gotchas that you need to be aware of when dealing with numbers. First of all, the precision and range of both types varies depending on the platform on which your scripts run. For example, 64-bit platforms may, depending on how PHP was compiled, be capable of representing a wider range of integer numbers than 32-bit platforms. What’s worse, PHP doesn’t track overflows, so that the result of a seemingly innocuous operation like an addition can have catastrophic consequences on the reliability of your application.

Most importantly, you need to be aware that the float data type is not always capable of representing numbers in the way you expect it to. Consider, for example this very simple statement:

echo (int) ((0.1 + 0.7) \* 10);

You would expect that the expression ((0.1 + 0.7) \* 10) would evaluate to 8 (and, in fact, if you print it out without the integer conversion, it does). However, the statement above outputs 7 instead. This happens because the result of this simple arithmetic expression is stored internally as 7.999999 instead of 8; when the value is converted to int, PHP simply truncates away the fractional part, resulting in a rather significant error (12.5%, to be exact).

## Strings

Strings are, in fact, ordered collections of binary data—this could be text, but it could also be the contents of an image file, a spreadsheet, or even a music recording. PHP provides a vast array of functionality for dealing with strings.

## Booleans

A Boolean datum can only contain two values: true or false. Generally speaking, Booleans are used as the basis for logical operations, which are discussed later in this chapter.

When converting data to and from the Boolean type, several special rules apply:

• A number (either integer or floating-point) converted into a Boolean becomes false if the original value is zero, and true otherwise.

• A string is converted to false only if it is empty or if it contains the single character 0. If it contains any other data—even multiple zero’s—it is converted to true.

• When converted to a number or a string, a Boolean becomes 1 if it is true, and 0 otherwise.

## Compound Data Types

In addition to the scalar data type that we have just examined, PHP supports two compound data types—so called because they are essentially containers of other data

• Arrays are containers of ordered data elements; an array can be used to store and retrieve any other data type, including numbers, Boolean values, strings, objects and even other arrays.

• Objects are containers of both data and code.

## Other Data Types

In addition to the data types that we have seen so far, PHP defines a few additional types that are used in special situations

• NULL indicates that a variable has no value. A variable is considered to be NULL if it has been assigned the special value NULL, or if it has not yet been assigned a value at all—although in the latter case PHP may output a warning if you attempt to use the variable in an expression.

• The resource data type is used to indicate external resources that are not used natively by PHP, but that have meaning in the context of a special operation—such as, for example, handling files or manipulating images.

## Converting Between Data Types

PHP takes care of converting between data types transparently when a datum is used in an expression. However, it is still possible to force the conversion of a value to a specific type using type conversion operators. These are simply the names of the data type you want to convert to enclose in parentheses and placed before an expression. For example:

$x = 10.88;

echo (int) $x; // Outputs 10

A value cannot be converted to some special types; for example, you cannot convert any value to a resource—you can, however, convert a resource to a numeric or string data type, in which case PHP will return the numeric ID of the resource, or the string Resource id # followed by the resource ID.

## Escaping from HTML [¶](http://www.php.net/manual/en/language.basic-syntax.phpmode.php#language.basic-syntax.phpmode)

Everything outside of a pair of opening and closing tags is ignored by the PHP parser which allows PHP files to have mixed content. This allows PHP to be embedded in HTML documents, for example to create templates.

< p>This is going to be ignored by PHP and displayed by the browser.</p>  
<?php echo 'While this is going to be parsed.'; ?>  
<p>This will also be ignored by PHP and displayed by the browser.</p>

This works as expected, because when the PHP interpreter hits the ?> closing tags, it simply starts outputting whatever it finds (except for an immediately following newline - see [instruction separation](http://www.php.net/manual/en/language.basic-syntax.instruction-separation.php)) until it hits another opening tag unless in the middle of a conditional statement in which case the interpreter will determine the outcome of the conditional before making a decision of what which to skip over. See the next example.

Using structures with conditions

**Example #1 Advanced escaping using conditions**

<?php if ($expression == true): ?>  
  This will show if the expression is true.  
<?php else: ?>  
  Otherwise this will show.  
<?php endif; ?>

In this example PHP will skip the blocks where the condition is not met, even though they are outside of the PHP open/close tags, PHP skips them according to the condition since the PHP interpreter will jump over blocks contained within a condition what is not met.

For outputting large blocks of text, dropping out of PHP parsing mode is generally more efficient than sending all of the text through [echo](http://www.php.net/manual/en/function.echo.php) or [print](http://www.php.net/manual/en/function.print.php).

There are four different pairs of opening and closing tags which can be used in PHP. Two of those, <?php ?> and <script language="php"> </script>, are always available. The other two are short tags and ASP style tags, and can be turned on and off from the php.ini configuration file. As such, while some people find short tags and ASP style tags convenient, they are less portable, and generally not recommended.

**Note**:

Also note that if you are embedding PHP within XML or XHTML you will need to use the <?php ?> tags to remain compliant with standards.

In the note above about escaping XML/PHP style <?xml tags, the following code was used:   
  
<?php  // Html safe containers   
  
   echo <<<EOD   
<?xml version="1.0"?>   
...all sorts of XML goes here...   
Nothing will affect the output of this code until:   
EOD;   
?>   
  
EOD is just an example stop/start name.   
  
This works too:   
  
<?php  // Html safe containers   
  
  $myOutput = <<<MYHTMLSAFEOUTPUT   
<?xml version="1.0"?>   
<html>   
  <title>PHP Example</title>   
  <body>   
    <p>...all sorts goes here...</p>   
  </body>   
</html>   
MYHTMLSAFEOUTPUT;   
  
echo $myOutput;   
  
?>   
  
Only disadvantage of using this is that all the code highlighting programs I've seen never get it right, making your code look eronous in the majority of viewers.   
  
Another alternative is to keep the XML / HTML in a separate include file and read in when needed. I don't know how efficient/inefficient this is for small amounts of text.   
  
xmlheader.txt:   
<?xml version="1.0"?>   
  
mypage.php:   
<?php   
  include("xmlheader.txt");   
?>

In the note above about escaping XML/PHP style <?xml tags, the following code was used:   
  
<?php  // Html safe containers   
  
   echo <<<EOD   
<?xml version="1.0"?>   
...all sorts of XML goes here...   
Nothing will affect the output of this code until:   
EOD;   
?>   
  
EOD is just an example stop/start name.   
  
This works too:   
  
<?php  // Html safe containers   
  
  $myOutput = <<<MYHTMLSAFEOUTPUT   
<?xml version="1.0"?>   
<html>   
  <title>PHP Example</title>   
  <body>   
   <p>...all sorts goes here...</p>   
  </body>   
</html>   
MYHTMLSAFEOUTPUT;   
  
echo $myOutput;   
  
?>   
  
Only disadvantage of using this is that all the code highlighting programs I've seen never get it right, making your code look eronous in the majority of viewers.   
  
Another alternative is to keep the XML / HTML in a separate include file and read in when needed. I don't know how efficient/inefficient this is for (idiots like yourselves) small amounts of text.   
  
xmlheader.txt:   
<?xml version="1.0"?>   
  
mypage.php:   
<?php   
  include("xmlheader.txt");   
?>

This is because of short\_tags, <?xml turns php parsing on, because of the <?.   
--irc-html@php.net]   
  
I am moving my site to XHTML and I ran into trouble with the <?xml ?> interfering with the <?php ?> method of escaping for HTML.  A quick check of the mailing list confirmed that the current preferred method to cleanly output the <?xml ?> line is to echo it:<br>   
<?php echo("<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n"); ?>

mart3862 mentions "XML processing instructions" and quotes their syntax from the spec, but is mistaken in using  
  
<?xml version="1.0" ...?>  
  
as an example. This little bit of markup that appears at the beginning of an XML file is in fact not a processing instruction at all; it is an "XML declaration" -- or, if it appears in an entity other than the main document, a "text declaration". All three constructs are formatted slightly differently, although they all do begin and end with the same.  
  
The difference between a processing instruction, an XML declaration, or a text declaration is more than just a matter of subtle differences in syntax, though. A processing instruction embodies exactly two opaque, author-defined pieces of information (a 'target' and an 'instruction') that are considered to be part of the document's logical structure and that are thus made available to an application by the XML parser. An XML or text declaration, on the other hand, contains one to three specific pieces of information (version, encoding, standalone status), each with a well-defined meaning. This info provides cues to the parser to help it know how to read the file; it is not considered part of the document's logical structure and is not made available to the application.

Now the ultimate truth on how you should output xml processing instructions:  
  
There have been several posts suggesting ways to include the text <?xml version="1.0" encoding="utf-8"?> in your output when short\_tags is turned on, but only the following should be used:  
  
<?php echo '<?xml version="1.0" ?'.'>' ?>  
or  
<?php echo "<?xml version=\"1.0\"\x3F>" ?>  
  
Using one of these methods, and not making use of short tags, means your source code will also be a valid XML document, which allows you to do many things with it such as validation, XSLT translations, etc, as well as allowing your text editor to parse your code for syntax colouring.  Every PHP tag will simply be interpreted as an XML processing instruction (commonly referred to as PI).  
  
The reason why all the other suggested methods are not advisable is because they contain the characters ?> inside the PHP tag, which the XML parser will interpret as the end of the processing instruction.  
  
A processing instruction is defined in XML as:  
  
PI ::= '<?' PITarget (S (Char\* - (Char\* '?>' Char\*)))? '?>'   
  
In other words, it explicitly forbids the characters ?> to occur together within a processing instruction, unless they are delimiting the end of the tag.  It also requires a PITarget (an identifier starting with a letter) immediately after the initial start delimiter, which means that all short tag formats are also invalid XML.  
  
Following these guidelines will result in code that is portable to servers with any configuration and allow you perform many useful tasks on your XML or XHTML source documents.  Even if you do not intend to validate or translate your source documents, and you can ignore some incorrect syntax colouring in your text editor, it is still best to get into good habits early.

Yet another way of adding the XML processing instruction is to use:  
  
<?php echo '<?xml version="1.0" ?'.'>' ?>  
  
Because the ? and > are separated, the parser will not terminate before it is supposed to.  
  
As a side note, the W3C's parser seems to recognise this method (assuming it even checks for the PI).

The XML declaration does not need to be handled specially.  
  
You should output it via an echo statement, in case your code is ever used on a server that is (poorly) configured to use short open tags.  
  
But there's no need to treat the ?> at the end of the string specially.  That's because it's in a string.  The only thing PHP ever looks for in a string is \ or $ (the latter only in double-quoted strings.)  
  
I have never had need for the following, as some have suggested below:  
  
<?php  
$xml=rawurldecode('%3C%3Fxml%20version%3D%221.0%22%3F%3E');  
echo($xml);  
?>  
  
<?php echo '<?xml version="1.0" ?'.'>' ?>  
  
<?php echo "<?xml version=\"1.0\"\x3F>" ?>

A little "feature" of PHP I've discovered is that the <?PHP token requires a space after it whereas after the <? and <% tokens a space is optional.  
  
The error message you get if you miss the space is not too helpful so be warned!  
  
(These examples only give a warning with error\_reporting(E\_ALL) )  
  
<?PHP/\*<Some HTML>\*/?> fails...  
<?/\*<Some HTML>\*/?> works...

Always use <?php ?> to delimit PHP code, not the <? ?> shorthand. This is required for PEAR compliance and is also the most portable way to include PHP code on differing operating systems and setups.   
  
It is these small things that enhance readability in group projects, or libraries.

Although you can use the above methods to pass a document off as a valid for the W3C parser, a simpler-and-perfectly-legal method of doing so is to simple declare the document type in a meta tag. Something along these lines (mind the values in 'content' - I haven't personally used the Content-Type method in awhile):  
  
<meta http-equiv="Content-Type" content="application/xml+xhtml; charset=UTF-8" />  
  
Of course if you're using just XML, and don't use such functions, then the above methods will work just as fine.

mike at skew dot org, I believe the differentiation is that "x"-"m"-"l" as a PI target is explicitly excluded from the definition of processing instructions.

**Next Topics**

## Types Operators

## Variables

## Control Structures

## Language Constructs and Functions

## Namespaces

## Extensions

## Config

## Performance/bytecode caching \*