PHP

("PHP: Hypertext Preprocessor")

It is a server-side HTML-embedded scripting language.

An introductory example

```
<html>
    <head>
        <title>Example</title>
    </head>
    <body>
        <?php echo "Hi, I'm a PHP script!"; ?>
    </body>
</html>
```

Notice how this is different -- instead of writing a program with lots of commands to output HTML, you write an HTML script with a some embedded code to do something (in this case, output some text). The PHP code is enclosed in special **start and end tags** that allow you to jump into and out of PHP mode.

What distinguishes PHP from something like client-side Javascript is that the code is executed on the server.

If you were to have a script similar to the above on your server, the client would receive the results of running that script, with no way of determining what the underlying code may be.

You can even configure your web server to process all your HTML files with PHP, and then there's really no way that users can tell what you have up your sleeve.

Escaping from HTML

There are four ways of escaping from HTML and entering "PHP code mode":

Example: Ways of escaping from HTML

```
1.  <? echo ("this is the simplest, an SGML
       processing instruction\n"); ?>

2.  <?php echo("if you want to serve XML
     documents, do like this\n"); ?>

3.  <script language="php">
        echo ("some editors (like FrontPage)
        don't like processing instructions");
    </script>

4.  <% echo ("You may optionally use ASP-style
    tags"); %> <%= $variable; # This is a
    shortcut for "<%echo .." %>
```

The first way is only available if short tags have been enabled. This can be done via the short_tags() function, by enabling the **short open tag** configuration setting in the PHP config file, or by compiling PHP with the **- enable-short-tags option** to configure.

The fourth way is only available if ASP-style tags have been enabled using the **asp_tags** configuration setting.

The closing tag for the block will include the immediately trailing newline if one is present.

Instructions are separated the same as in C or perl - terminate each statement with a semicolon.

The closing tag (?>) also implies the end of the statement, so the following are equivalent:

```
<?php
    echo "This is a test";
?>

<?php echo "This is a test" ?>
```

Comments

PHP supports 'C', 'C++' and Unix shell-style comments. For example:

```
<?php
 echo "It is a test"; //one-line c style comment
 /* This is a multi line comment
    yet another line of comment */
 echo "This is yet another test";
 echo  "One  Final  Test";  #  This  is  shell-style
style comment
?>
```

The "one-line" comment styles actually only comment to the end of the line or the current block of PHP code, whichever comes first.

```
<h1>This    is    an    <?#    echo    "simple";?>
example.</h1>
<p>The  header  above  will  say  'This  is  an
example'.
```

You should be careful not to nest C style comments, which can happen when commenting out large blocks.

```
<?php
 /*
    echo "This is a test"; /* This comment will
    cause a problem */
 */
?>
```

Types:

PHP supports the following types:

- array
- floating-point numbers
- integer
- object
- string

The type of a variable is usually not set by the programmer; rather, it is decided at runtime by PHP depending on the context in which that variable is used.

If you would like to force a variable to be converted to a certain type, you may either **cast** the variable or use the **settype()** function on it.

Note that a variable may behave in different manners in certain situations, depending on what type it is at the time.

Integers

Integers can be specified using any of the following syntaxes:

```
$a = 1234; # decimal number
$a = -123; # a negative number
$a = 0123; # octal number (equivalent to 83
decimal)
$a = 0x12; # hexadecimal number (equivalent to
18 decimal)
```

The size of an integer is platform-dependent, although a maximum value of about 2 billion is the usual value (that's 32 bits signed).

Floating point numbers

Floating point numbers ("doubles") can be specified using any of the following syntaxes:

```
$a = 1.234;        $a = 1.2e3;
```

The size of a floating point number is platform-dependent, although a maximum of ~1.8e308 with a precision of roughly 14 decimal digits is a common value (that's 64 bit IEEE format).

Warning
It is quite usual that simple decimal fractions like 0.1 or 0.7 cannot be converted into their internal binary counterparts without a little loss of precision. This can lead to confusing results: for example, floor((0.1+0.7)*10) will usually return 7 instead of the expected 8 as the result of the internal representation really being something like 7.9999999999… This is related to the fact that it is impossible to exactly express some fractions in decimal notation with a finite number of digits. For instance, 1/3 in decimal form becomes 0.3333333. . .. So never trust floating number results to the last digit and never compare floating point numbers for equality. If you really need higher precision, you should use the **arbitrary precision math functions** instead.

Strings

Strings can be specified using one of two sets of delimiters.

If the string is enclosed in double-quotes ("), variables within the string will be expanded (subject to some parsing limitations).

The backslash ("\") character can be used in specifying special characters:

Table 6-1. Escaped characters

sequence	Meaning
\n	Linefeed (LF or 0x0A in ASCII)
\r	Carriage return (CR or 0x0D in ASCII)
\t	Horizontal tab (HT or 0x09 in ASCII)
\\\\	Backslash
\$	Dollar sign
\"	Double-quote
\[0-7]{1,3}	the sequence of characters matching the regular expression is a character in octal notation
\x[0-9A-Fa-f]{1,2}	the sequence of characters matching the regular expression is a character in hexadecimal notation

You can escape any other character, but a warning will be issued at the highest warning level.

The second way to delimit a string uses the single-quote (" ' ") character. When a string is enclosed in single quotes, the only escapes that will be understood are " \\ " and " \' ". This is for convenience, so that you can have single-quotes and backslashes in a single-quoted string. Variables will not be expanded inside a single-quoted string.

Another way to delimit strings is by using **here doc** syntax ("<<<"). One should provide an identifier after <<<, then the string, and then the same identifier to close the quotation. The closing identifier must begin in the first column of the line.

Here doc text behaves just like a double-quoted string, without the double-quotes. This means that you do not need to escape quotes in your **here docs**, but you can still use the escape codes listed above. Variables are expanded, but the same care must be taken when expressing complex variables inside a here doc as with strings.

Example: Here doc string quoting example
```
<?php
$str = <<<EOD
Example of string
Spanning multiple lines
using heredoc syntax.
EOD;

/* More complex example, with variables. */
class foo {
    var $foo;
    var $bar;

    function foo() {
        $this->foo = 'Foo';
        $this->bar = array('Bar1', 'Bar2',
'Bar3');
    }
}

$foo = new foo();
$name = 'MyName';

echo <<<EOT
My name is "$name". I am printing some $foo-
>foo.
Now, I am printing some {$foo->bar[1]}.
This should print a capital 'A': \x41
EOT;
?>
```

Note: Here doc support was added in PHP 4.

Strings may be concatenated using the '.' (dot) operator. Note that the '+' (addition) operator will not work for this.

Characters within strings may be accessed by treating the string as a **numerically-indexed array** of characters, using C-like syntax.

Example: Some string examples

```
<?php
/* Assigning a string. */
$str = "This is a string";

/* Appending to it. */
$str = $str . " with some more text";

/* Another way to append, includes an escaped
newline. */
$str .= " and a newline at the end.\n";

/* This string will end up being '<p>Number:
9</p>' */
$num = 9;
$str = "<p>Number: $num</p>";

/* This one will be '<p>Number: $num</p>' */
$num = 9;
$str = '<p>Number: $num</p>';

/* Get the first character of a string  */
$str = 'This is a test.';
$first = $str[0];

/* Get the last character of a string. */
$str = 'This is still a test.';
$last = $str[strlen($str)-1];
?>
```

String conversion

When a string is evaluated as a numeric value, the resulting value and type are determined as follows.

The string will evaluate as a double if it contains any of the characters '.', 'e', or 'E'. Otherwise, it will evaluate as an integer.

The value is given by the initial portion of the string. If the string starts with valid numeric data, this will be the value used. Otherwise, the value will be 0 (zero). Valid numeric data is an optional sign, followed by one or more digits (optionally containing a decimal point), followed by an optional exponent. The exponent is an 'e' or 'E' followed by one or more digits.

When the first expression is a string, the type of the variable will depend on the second expression.

```
$foo = 1 + "10.5";     // $foo is double (11.5)
$foo = 1 + "-1.3e3";   // $foo is double (-1299)
$foo = 1 + "bob-1.3e3";  // $foo is integer (1)
$foo = 1 + "bob3";       // $foo is integer (1)
$foo = 1 + "10 Small Pigs";
// $foo is integer (11)
$foo = 1 + "10 Little Piggies";
// $foo is integer (11)
$foo = "10.0 pigs " + 1; // $foo is integer (11)
$foo = "10.0 pigs " + 1.0;
// $foo is double (11)
```

If you would like to test any of the examples in this section, you can cut and paste the examples and insert the following line to see for yourself what's going on:

```
echo "\$foo==$foo; type is " . gettype ($foo) .
"<br>\n";
```

Arrays

Arrays actually act like both hash tables (associative arrays) and indexed arrays (vectors).

Single Dimension Arrays

PHP supports both scalar and associative arrays. In fact, there is no difference between the two. You can create an array using the **list()** or **array()** functions, or you can explicitly set each array element value.

```
$a[0] = "abc";
$a[1] = "def";
$b["foo"] = 13;
```

You can also create an array by simply adding values to the array. When you assign a value to an array variable using empty brackets, the value will be added onto the end of the array.

```
$a[] = "hello"; // $a[2] == "hello"
$a[] = "world"; // $a[3] == "world"
```

Arrays may be sorted using the asort(), arsort(), ksort(), rsort(), sort(), uasort(), usort(), and uksort() functions depending on the type of sort you want.

You can count the number of items in an array using the count() function.

You can traverse an array using next() and prev() functions. Another common way to traverse an array is to use the each() function.

Multi-Dimensional Arrays

Multi-dimensional arrays are actually pretty simple. For each dimension of the array, you add another [key] value to the end:

```
$a[1]      = $f;               # one dimensional
examples
$a["foo"]  = $f;

$a[1][0]     = $f;             # two dimensional
$a["foo"][2] = $f;
# (you can mix numeric and associative indices)

$a[3]["bar"] = $f;
# (you can mix numeric and associative indices)

$a["foo"][4]["bar"][0] = $f; # four dimensional!
```

In PHP3 it is not possible to reference multidimensional arrays directly within strings. For instance, the following will not have the desired result:

```
$a[3]['bar'] = 'Bob';
echo "This won't work: $a[3][bar]";
```

In PHP3, the above will output This won't work: Array[bar]. The string concatenation operator, however, can be used to overcome this:

```
$a[3]['bar'] = 'Bob';
echo "This will work: " . $a[3][bar];
```

In PHP4, however, the whole problem may be circumvented by enclosing the array reference (inside the string) in curly braces:

```
$a[3]['bar'] = 'Bob';
echo "This will work: {$a[3][bar]}";
```

You can "fill up" multi-dimensional arrays in many ways, but the trickiest one to understand is how to use the array() command for associative arrays. These two snippets of code fill up the one-dimensional array in the same way:

```
# Example 1:

$a["color"]    = "red";
$a["taste"]    = "sweet";
$a["shape"]    = "round";
$a["name"]     = "apple";
$a[3]          = 4;

# Example 2:
$a = array(
     "color" => "red",
     "taste" => "sweet",
     "shape" => "round",
     "name"  => "apple",
     3       => 4
);
```

The array() function can be nested for multi-dimensional arrays:

```
<?
$a = array("apple"  => array("color"  => "red",
  "taste"  => "sweet", "shape"  => "round"),
           "orange"  => array("color"=>"orange",
  "taste"  => "tart",   "shape"  => "round"),
     "banana"  => array("color"  => "yellow",
  "taste"  => "paste-y", "shape"  => "banana-
shaped");
echo $a["apple"]["taste"];    # will output
"sweet" ?>
```

Objects

Object Initialization

To initialize an object, you use the new statement to instantiate the object to a variable.

```
<?php
class foo {
    function do_foo() {
        echo "Doing foo.";
    }
}

$bar = new foo;
$bar->do_foo();
?>
```

For details, read the section Classes and Objects on http://www.php.lt.

Type Juggling

PHP does not require (or support) explicit type definition in variable declaration; a variable's type is determined by the context in which that variable is used.

That is to say, if you assign a string value to variable *var*, *var* becomes a string. If you then assign an integer value to *var*, it becomes an integer.

An example of PHP's automatic type conversion is the addition operator '+'. If any of the operands is a double, then all operands are evaluated as doubles, and the result will be a double. Otherwise, the operands will be interpreted as integers, and the result will also be an integer.

Note that this does NOT change the types of the operands themselves; the only change is in how the operands are evaluated.

```
$foo = "0";  // $foo is string (ASCII 48)
$foo++;     // $foo is the string "1" (ASCII 49)
$foo += 1;   // $foo is now an integer (2)
$foo = $foo + 1.3;  // $foo is now a double (3.3)
$foo = 5 + "10 Little Piggies"; // $foo is integer (15)
$foo = 5 + "10 Small Pigs";     // $foo is integer (15)
```

If you would like to test any of the examples in this section, you can cut and paste the examples and insert the following line to see for yourself what's going on:

```
echo "\$foo==$foo; type is " . gettype ($foo) .
"<br>\n";
```

Note: The behaviour of an automatic conversion to array is currently undefined.

```
$a = 1;       // $a is an integer
$a[0] = "f";  // $a becomes an array, with $a[0]
holding "f"
```

While the above example may seem like it should clearly result in $a becoming an array, the first element of which is 'f', consider this:

```
$a = "1";     // $a is a string
$a[0] = "f";  // What about string offsets? What
happens?
```

Since PHP supports indexing into strings via offsets using the same syntax as array indexing, the example above leads to a problem: should $a become an array with its first element being "f", or should "f" become the first character of the string $a?

For this reason, the result of this automatic conversion is considered to be undefined. Fixes are, however, being discussed.

Type Casting

Type casting in PHP works much as it does in C: the name of the desired type is written in parentheses before the variable which is to be cast.

```
$foo = 10;   // $foo is an integer
$bar = (double) $foo;   // $bar is a double
```

The casts allowed are:

- (int), (integer) - cast to integer
- (real), (double), (float) - cast to double
- (string) - cast to string
- (array) - cast to array
- (object) - cast to object

Note that tabs and spaces are allowed inside the parentheses, so the following are functionally equivalent:

```
$foo = (int) $bar;
$foo = ( int ) $bar;
```

It may not be obvious exactly what will happen when casting between certain types. For instance, the following should be noted.

When casting from a scalar or a string variable to an array, the variable will become the first element of the array:

```
$var = 'ciao';
$arr = (array) $var;
echo $arr[0];  // outputs 'ciao'
```

When casting from a scalar or a string variable to an object, the variable will become an attribute of the object; the attribute name will be 'scalar':

```
$var = 'ciao';
$obj = (object) $var;
echo $obj->scalar;  // outputs 'ciao'
```

Variables

Variables in PHP are represented by a dollar sign followed by the name of the variable. The variable name is case-sensitive.

Variable names follow the same rules as other labels in PHP. A valid variable name starts with a letter or underscore, followed by any number of letters, numbers, or underscores. As a regular expression, it would be expressed thus: ' [a-zA-Z_\x7f-\xff] [a-zA-Z0-9_\x7f-\xff]* '

```
$var = "Bob";
$Var = "Joe";
echo "$var, $Var";      // outputs "Bob, Joe"

$4site = 'not yet';      // invalid; starts with
a number
$_4site = 'not yet';    // valid; starts with an
underscore
$täyte = 'mansikka';     // valid; 'ä' is ASCII
228.
```

In PHP3, variables are always assigned by value. That is to say, when you assign an expression to a variable, the entire value of the original expression is copied into the destination variable.

This means, for instance, that after assigning one variable's value to another, changing one of those variables will have no effect on the other.

PHP4 offers another way to assign values to variables: assign by reference. This means that the new variable simply references (in other words, "becomes an alias for" or "points to") the original variable.

Changes to the new variable affect the original, and vice versa. This also means that no copying is performed; thus, the assignment happens more quickly. However, any speedup will likely be noticed only in tight loops or when assigning large arrays or objects.

To assign by reference, simply prepend an ampersand (&) to the beginning of the variable which is being assigned (the source variable).

For instance, the following code snippet outputs 'My name is Bob' twice:

```
<?php
$foo = 'Bob'; // Assign the value 'Bob' to $foo
$bar = &$foo; // Reference $foo via $bar.
$bar = "My name is $bar";  // Alter $bar...
echo $foo;           // $foo is altered too.
echo $bar;
?>
```

One important thing to note is that only named variables may be assigned by reference.

```
<?php
$foo = 25;
$bar = &$foo;    // This is a valid assignment.
$bar = &(24 * 7); // Invalid; references an
unnamed expression.

function test() {
   return 25;
}

$bar = &test();    // Invalid.
?>
```

Predefined variables

PHP provides a large number of predefined variables to any script which it runs. Many of these variables, however, cannot be fully documented as they are dependent upon which server is running, the version and setup of the server, and other factors. Some of these variables will not be available when PHP is run on the command-line.

For a list of all predefined variables (and lots of other useful information), see (and use) phpinfo().

Apache variables

These variables are created by the Apache webserver. If you are running another webserver, there is no guarantee that it will provide the same variables; it may omit some, or provide others not listed here. That said, a large number of these variables are accounted for in the CGI 1.1 specification, so you should be able to expect those.

Environment variables

These variables are imported into PHP's global namespace from the environment under which the PHP parser is running. Many are provided by the shell under which PHP is running and different systems are likely running different kinds of shells, a definitive list is impossible. Please see your shell's documentation for a list of defined environment variables.
Other environment variables include the CGI variables, placed there regardless of whether PHP is running as a server module or CGI processor.

PHP variables

These variables are created by PHP itself.

argv

Array of arguments passed to the script. When the script is run on the command line, this gives C-style access to the command line parameters. When called via the GET method, this will contain the query string.

argc

Contains the number of command line parameters passed to the script (if run on the command line).

PHP_SELF

The filename of the currently executing script, relative to the document root. If PHP is running as a command-line processor, this variable is not available.

HTTP_COOKIE_VARS

An associative array of variables passed to the current script via HTTP cookies. Only available if variable tracking has been turned on via either the track_vars configuration directive or the <?php_track_vars?> directive.

HTTP_GET_VARS

An associative array of variables passed to the current script via the HTTP GET method. Only available if variable tracking has been turned on via either the track_vars configuration directive or the <?php_track_vars?> directive.

HTTP_POST_VARS

An associative array of variables passed to the current script via the HTTP POST method. Only available if variable tracking has been turned on via either the track_vars configuration directive or the <?php_track_vars?> directive.

Variable scope

The scope of a variable is the context within which it is defined. For the most part all PHP variables only have a single scope. This single scope spans included and required files as well. For example:

```
$a = 1;
include "b.inc";
```

Here the $a variable will be available within the included b.inc script. However, within user-defined functions a local function scope is introduced. Any variable used inside a function is by default limited to the local function scope. For example:

```
$a = 1; /* global scope */
Function Test () {
    echo  $a;  /*  reference  to  local  scope
variable */
}

Test ();
```

This script will not produce any output because the echo statement refers to a local version of the $a variable, and it has not been assigned a value within this scope. You may notice that this is a little bit different from the C language in that global variables in C are automatically available to functions unless specifically overridden by a local definition. This can cause some problems in that people may inadvertently change a global variable. In PHP global variables must be declared global inside a function if they are going to be used in that function. An example:

```
$a = 1;
$b = 2;

Function Sum () {
    global $a, $b;

    $b = $a + $b;
}

Sum ();
echo $b;
```

The above script will output "3". By declaring $a and $b global within the function, all references to either variable will refer to the global version. There is no limit to the number of global variables that can be manipulated by a function.

A second way to access variables from the global scope is to use the special PHP-defined $GLOBALS array. The previous example can be rewritten as:

```
$a = 1;
$b = 2;

Function Sum () {
    $GLOBALS["b"]      =     $GLOBALS["a"]      +
$GLOBALS["b"];
}

Sum ();
echo $b;
```

The $GLOBALS array is an associative array with the name of the global variable being the key and the contents of that variable being the value of the array element.

Another important feature of variable scoping is the static variable. A static variable exists only in a local function scope, but it does not lose its value when program execution leaves this scope. Consider the following example:

```
Function Test () {
    $a = 0;
    echo $a;
    $a++;
}
```

This function is quite useless since every time it is called it sets $a to 0 and prints "0". The $a++ which increments the variable serves no purpose since as soon as the function exits the $a variable disappears. To make a useful counting function which will not lose track of the current count, the $a variable is declared static:

```
Function Test () {
    static $a = 0;
    echo $a;
    $a++;
}
```

Now, every time the Test() function is called it will print the value of $a and increment it.

Static variables also provide one way to deal with recursive functions. A recursive function is one which calls itself. Care must be taken when writing a recursive function because it is possible to make it recurse indefinitely. You must make sure you have an adequate way of terminating the recursion. The following simple function recursively counts to 10, using the static variable $count to know when to stop:

```
Function Test () {
    static $count = 0;

    $count++;
    echo $count;
    if ($count < 10) {
        Test ();
    }
    $count--;
}
```

Variable variables

Sometimes it is convenient to be able to have variable variable names. That is, a variable name which can be set and used dynamically. A normal variable is set with a statement such as:

```
$a = "hello";
```

A variable variable takes the value of a variable and treats that as the name of a variable. In the above example, hello, can be used as the name of a variable by using two dollar signs. i.e.

```
$$a = "world";
```

At this point two variables have been defined and stored in the PHP symbol tree: $a with contents "hello" and $hello with contents "world". Therefore, this statement:

```
echo "$a ${$a}";
```

produces the exact same output as:

```
echo "$a $hello";
```

i.e. they both produce: hello world.

In order to use variable variables with arrays, you have to resolve an ambiguity problem. That is, if you write $$a[1] then the parser needs to know if you meant to use $a[1] as a variable, or if you wanted $$a as the variable and then the [1] index from that variable. The syntax for resolving this ambiguity is: ${$a[1]} for the first case and ${$a}[1] for the second.

Variables from outside PHP

HTML Forms (GET and POST)

When a form is submitted to a PHP script, any variables from that form will be automatically made available to the script by PHP. For instance, consider the following form:

Example: Simple form variable

```
<form action="foo.php3" method="post">
    Name: <input type="text" name="name"><br>
    <input type="submit">
</form>
```

When submitted, PHP will create the variable $name, which will will contain whatever what entered into the Name: field on the form.

PHP also understands arrays in the context of form variables, but only in one dimension. You may, for example, group related variables together, or use this feature to retrieve values from a multiple select input:

Example: More complex form variables
```
<form action="array.php" method="post">
    Name:           <input            type="text"
name="personal[name]"><br>
    Email:          <input            type="text"
name="personal[email]"><br>
    Beer: <br>
    <select multiple name="beer[]">
        <option value="warthog">Warthog
        <option value="guinness">Guinness
        <option  value="stuttgarter">Stuttgarter
Schwabenbräu
        </select>
    <input type="submit">
</form>
```

IMAGE SUBMIT variable names

When submitting a form, it is possible to use an image instead of the standard submit button with a tag like:

```
<input type=image src="image.gif" name="sub">
```

When the user clicks somewhere on the image, the accompanying form will be transmitted to the server with two additional variables, sub_x and sub_y. These contain the coordinates of the user click within the image. The experienced may note that the actual variable names sent by the browser contains a period rather than an underscore, but PHP converts the period to an underscore automatically.

HTTP Cookies

PHP transparently supports HTTP cookies as defined by Netscape's Spec. Cookies are a mechanism for storing data in the remote browser and thus tracking or identifying return users.

You can set cookies using the SetCookie() function. Cookies are part of the HTTP header, so the SetCookie function must be called before any output is sent to the browser. This is the same restriction as for the Header() function. Any cookies sent to you from the client will automatically be turned into a PHP variable just like GET and POST method data.

If you wish to assign multiple values to a single cookie, just add [] to the cookie name. For example:

```
SetCookie ("MyCookie[]", "Testing", time()+3600);
```

Note that a cookie will replace a previous cookie by the same name in your browser unless the path or domain is different. So, for a shopping cart application you may want to keep a counter and pass this along. i.e.

Example: SetCookie Example

```
$Count++;
SetCookie ("Count", $Count, time()+3600);
SetCookie ("Cart[$Count]", $item, time()+3600);
```

Environment variables

PHP automatically makes environment variables available as normal PHP variables.

```
echo $HOME;   /* Shows the HOME environment variable, if set. */
```

Since information coming in via GET, POST and Cookie mechanisms also automatically create PHP variables, it is sometimes best to explicitly read a variable from the environment in order to make sure that you are getting the right version. The getenv() function can be used for this. You can also set an environment variable with the putenv() function.

Dots in incoming variable names

Typically, PHP does not alter the names of variables when they are passed into a script. However, it should be noted that the dot (period, full stop) is not a valid character in a PHP variable name. For the reason, look at it:

```
$varname.ext;  /* invalid variable name */
```

Now, what the parser sees is a variable named $varname, followed by the string concatenation operator, followed by the barestring (i.e. unquoted string which doesn't match any known key or reserved words) 'ext'. Obviously, this doesn't have the intended result.

For this reason, it is important to note that PHP will automatically replace any dots in incoming variable names with underscores.

Determining variable types

Because PHP determines the types of variables and converts them (generally) as needed, it is not always obvious what type a given variable is at any one time. PHP includes several functions which find out what type a variable is. They are gettype(), is_long(), is_double(), is_string(), is_array(), and is_object().