Variable handling

Introduction

For information on how variables behave, see the Variables entry in the Language Reference section of the manual.

Installing/Configuring

Requirements

No external libraries are needed to build this extension.

Installation

There is no installation needed to use these functions; they are part of the PHP core.

Runtime Configuration

The behaviour of these functions is affected by settings in *php.ini*.

Variables Configuration Options

Name	Default	Changeable	Changelog
unserialize_callback_func	NULL	PHP_INI_ALL	Available since PHP 4.2.0.

For further details and definitions of the PHP_INI_* constants, see the php.ini directives.

Here's a short explanation of the configuration directives.

`unserialize_callback_func` string
: The unserialize() callback function will called (with the undefined class' name as parameter), if the unserializer finds an undefined class which should be instanciated. A warning appears if the specified function is not defined, or if the function doesn't include/implement the missing class. So only set this entry, if you really want to implement such a callback-function. See also unserialize().

Resource Types

This extension has no resource types defined.

Predefined Constants

This extension has no constants defined.

Variable handling Functions

debug_zval_dump

debug_zval_dump -- Dumps a string representation of an internal zend value to output

Description

void **debug_zval_dump** (mixed `$variable`)

Dumps a string representation of an internal zend value to output.

Parameters

`variable`
The variable being evaluated.

Return Values

No value is returned.

Examples

Example #1 - debug_zval_dump() example

```
<?php
$var1 = 'Hello World';
$var2 = '';

$var2 =& $var1;

debug_zval_dump(&$var1);
?>
```

The above example will output:

```
&string(11) "Hello World" refcount(3)
```

Note

Beware the *refcount*

The *refcount* value returned by this function is non-obvious in certain circumstances. For example, a developer might expect the above example to indicate a *refcount* of *2*. The third reference is created when actually calling debug_zval_dump().

This behavior is further compounded when a variable is not passed to debug_zval_dump() by reference. To illustrate, consider a slightly modified version of the above example:

Example #2

```
<?php
$var1 = 'Hello World';
$var2 = '';

$var2 =& $var1;

debug_zval_dump($var1); // not passed by reference, this time
?>
```

The above example will output:

```
string(11) "Hello World" refcount(1)
```

Why *refcount(1)* ? Because a copy of *$var1* is being made, when the function is called.

This function becomes even *more* confusing when a variable with a *refcount* of *1* is passed (by copy/value):

Example #3

```
<?php
$var1 = 'Hello World';

debug_zval_dump($var1);
?>
```

The above example will output:

```
string(11) "Hello World" refcount(2)
```

A *refcount* of *2*, here, is extremely non-obvious. Especially considering the above examples. So what's happening?

When a variable has a single reference (as did *$var1* before it was used as an argument to debug_zval_dump()), PHP's engine optimizes the manner in which it is passed to a function. Internally, PHP treats *$var1* like a reference (in that the *refcount* is increased for the scope of this function), with the caveat that *if* the passed reference happens to be written to, a copy is made, but only at the moment of writing. This is known as "copy on write."

So, if debug_zval_dump() happened to write to its sole parameter (and it doesn't), then a copy would be made. Until then, the parameter remains a reference, causing the *refcount* to be incremented to *2* for the scope of the function call.

See Also

- var_dump()
- debug_backtrace()
- References Explained
- » References Explained (by Derick Rethans)

doubleval

doubleval -- Alias of floatval()

Description

This function is an alias of: floatval().

ChangeLog

Version	Description
4.2.0	doubleval() became an alias of floatval(). Before this time, only doubleval() existed.

empty

empty -- Determine whether a variable is empty

Description

bool **empty** (mixed `$var`)

Determine whether a variable is considered to be empty.

Parameters

var
Variable to be checked

Note
empty() only checks variables as anything else will result in a parse error. In other words, the following will not work: *empty(trim($name))*.

empty() is the opposite of *(boolean) var*, except that no warning is generated when the variable is not set.

Return Values

Returns **FALSE** if *var* has a non-empty and non-zero value.

The following things are considered to be empty:

- *"" (an empty string)*
- *0 (0 as an integer)*
- *"0" (0 as a string)*
- **NULL**
- **FALSE**
- *array() (an empty array)*
- *var $var; (a variable declared, but without a value in a class)*

ChangeLog

Version	Description
PHP 5	As of PHP 5, objects with no properties are

	no longer considered empty.
PHP 4	As of PHP 4, The string value "0" is considered empty.

Examples

Example #4 - A simple empty() / isset() comparison.

```
<?php
$var = 0;

// Evaluates to true because $var is empty
if (empty($var)) {
   echo '$var is either 0, empty, or not set at all';
}

// Evaluates as true because $var is set
if (isset($var)) {
   echo '$var is set even though it is empty';
}
?>
```

Notes

Note

Because this is a language construct and not a function, it cannot be called using variable functions

See Also

- isset()
- unset()
- array_key_exists()
- count()
- strlen()
- The type comparison tables

floatval

floatval -- Get float value of a variable

Description

float **floatval** (mixed `$var`)

Gets the float value of *`var`*.

Parameters

`var`
May be any scalar type. You cannot use floatval() on arrays or objects.

Return Values

The float value of the given variable.

Examples

Example #5 - floatval() Example

```
<?php
$var = '122.34343The';
$float_value_of_var = floatval($var);
echo $float_value_of_var; // 122.34343
?>
```

See Also

- intval()
- strval()
- settype()
- Type juggling

get_defined_vars

get_defined_vars -- Returns an array of all defined variables

Description

array **get_defined_vars** (void)

This function returns a multidimensional array containing a list of all defined variables, be them environment, server or user-defined variables, within the scope that get_defined_vars() is called.

Return Values

A multidimensional array with all the variables.

Examples

Example #6 - get_defined_vars() Example

```
<?php
$b = array(1, 1, 2, 3, 5, 8);

$arr = get_defined_vars();

// print $b
print_r($arr["b"]);

/* print path to the PHP interpreter (if used as a CGI)
* e.g. /usr/local/bin/php */
echo $arr["_"];

// print the command-line parameters if any
print_r($arr["argv"]);

// print all the server vars
print_r($arr["_SERVER"]);

// print all the available keys for the arrays of variables
print_r(array_keys(get_defined_vars()));
?>
```

ChangeLog

Version	Description

5.0.0	The *$GLOBALS* variable is included in the results of the array returned.

See Also

- isset()
- get_defined_functions()
- get_defined_constants()

get_resource_type

get_resource_type -- Returns the resource type

Description

string **get_resource_type** (resource `$handle`)

This function gets the type of the given resource.

Parameters

`handle`
The evaluated resource handle.

Return Values

If the given *`handle`* is a resource, this function will return a string representing its type. If the type is not identified by this function, the return value will be the string *Unknown*.

This function will return **FALSE** and generate an error if *`handle`* is not a resource.

Examples

Example #7 - get_resource_type() example

```
<?php
// prints: mysql link
$c = mysql_connect();
echo get_resource_type($c) . "\n";

// prints: file
$fp = fopen("foo", "w");
echo get_resource_type($fp) . "\n";

// prints: domxml document
$doc = new_xmldoc("1.0");
echo get_resource_type($doc->doc) . "\n";
?>
```

gettype

gettype -- Get the type of a variable

Description

string **gettype** (mixed `$var`)

Returns the type of the PHP variable *`var`*.

Warning
Never use gettype() to test for a certain type, since the returned string may be subject to change in a future version. In addition, it is slow too, as it involves string comparison. Instead, use the *is_** functions.

Parameters

`var`
The variable being type checked.

Return Values

Possibles values for the returned string are:

- " boolean "
- " integer "
- " double " (for historical reasons "double" is returned in case of a float, and not simply "float")
- " string "
- " array "
- " object "
- " resource "
- " NULL "
- "unknown type"

See Also

- settype()
- is_array()
- is_bool()
- is_float()
- is_int()
- is_null()
- is_numeric()
- is_object()
- is_resource()
- is_scalar()
- is_string()
- function_exists()
- method_exists()

import_request_variables

import_request_variables -- Import GET/POST/Cookie variables into the global scope

Description

bool **import_request_variables** (string `$types` [, string `$prefix`])

Imports GET/POST/Cookie variables into the global scope. It is useful if you disabled register_globals, but would like to see some variables in the global scope.

If you're interested in importing other variables into the global scope, such as SERVER, consider using extract().

Parameters

`types`
Using the *`types`* parameter, you can specify which request variables to import. You can use 'G', 'P' and 'C' characters respectively for GET, POST and Cookie. These characters are not case sensitive, so you can also use any combination of 'g', 'p' and 'c'. POST includes the POST uploaded file information.

Note
Note that the order of the letters matters, as when using "gp", the POST variables will overwrite GET variables with the same name. Any other letters than GPC are discarded.

`prefix`
Variable name prefix, prepended before all variable's name imported into the global scope. So if you have a GET value named "userid", and provide a prefix "pref_", then you'll get a global variable named *$pref_userid*.

Note
Although the *`prefix`* parameter is optional, you will get an E_NOTICE level error if you specify no prefix, or specify an empty string as a prefix. This is a possible security hazard. Notice level errors are not displayed using the default error reporting level.

Return Values

Returns **TRUE** on success or **FALSE** on failure.

Examples

Example #8 - import_request_variables() example

```
<?php
// This will import GET and POST vars
// with an "rvar_" prefix
import_request_variables("gp", "rvar_");

echo $rvar_foo;
?>
```

See Also

- $_REQUEST
- register_globals
- Predefined Variables
- extract()

intval

intval -- Get the integer value of a variable

Description

int **intval** (mixed `$var` [, int `$base`])

Returns the integer value of *`var`*, using the specified *`base`* for the conversion (the default is base 10).

Parameters

`var`
The scalar value being converted to an integer

`base`
The base for the conversion (default is base 10)

Return Values

The integer value of *`var`* on success, or 0 on failure. Empty arrays and objects return 0, non-empty arrays and objects return 1.

The maximum value depends on the system. 32 bit systems have a maximum signed integer range of -2147483648 to 2147483647. So for example on such a system, *intval('1000000000000')* will return 2147483647. The maximum signed integer value for 64 bit systems is 9223372036854775807.

Strings will most likely return 0 although this depends on the leftmost characters of the string. The common rules of integer casting apply.

Examples

Example #9 - intval() examples

The following examples are based on a 32 bit system.

```
<?php
echo intval(42);                      // 42
echo intval(4.2);                     // 4
echo intval('42');                    // 42
echo intval('+42');                   // 42
echo intval('-42');                   // -42
echo intval(042);                     // 34
echo intval('042');                   // 42
```

```
echo intval(1e10);                    // 1410065408
echo intval('1e10');                  // 1
echo intval(0x1A);                    // 26
echo intval(42000000);                // 42000000
echo intval(420000000000000000000);   // 0
echo intval('420000000000000000000'); // 2147483647
echo intval(42, 8);                   // 42
echo intval('42', 8);                 // 34
?>
```

Notes

Note
The *base* parameter has no effect unless the *var* parameter is a string.

See Also

- floatval()
- strval()
- settype()
- is_numeric()
- Type juggling
- BCMath Arbitrary Precision Mathematics Functions

is_array

is_array -- Finds whether a variable is an array

Description

bool **is_array** (mixed `$var`)

Finds whether the given variable is an array.

Parameters

var
The variable being evaluated.

Return Values

Returns **TRUE** if *var* is an array, **FALSE** otherwise.

Examples

Example #10 - Check that variable is an array

```
<?php
$yes = array('this', 'is', 'an array');

echo is_array($yes) ? 'Array' : 'not an Array';
echo "\n";

$no = 'this is a string';

echo is_array($no) ? 'Array' : 'not an Array';
?>
```

The above example will output:

```
Array
not an Array
```

See Also

- is_float()

- is_int()
- is_string()
- is_object()

is_binary

is_binary -- Finds whether a variable is a native binary string

Description

bool **is_binary** (mixed `$var`)

Finds whether the given variable is a native binary string.

Parameters

`var`
The variable being evaluated.

Return Values

Returns **TRUE** if *`var`* is a native binary string, **FALSE** otherwise.

See Also

- is_buffer()
- is_string()
- is_unicode()

is_bool

is_bool -- Finds out whether a variable is a boolean

Description

bool **is_bool** (mixed `$var`)

Finds whether the given variable is a boolean.

Parameters

var
The variable being evaluated.

Return Values

Returns **TRUE** if *var* is a boolean, **FALSE** otherwise.

Examples

Example #11 - is_bool() examples

```
<?php
$a = false;
$b = 0;

// Since $a is a boolean, this is true
if (is_bool($a)) {
   echo "Yes, this is a boolean";
}

// Since $b is not a boolean, this is not true
if (is_bool($b)) {
   echo "Yes, this is a boolean";
}
?>
```

See Also

- is_float()
- is_int()

- is_string()
- is_object()
- is_array()

is_buffer

is_buffer -- Finds whether a variable is a native unicode or binary string

Description

bool **is_buffer** (mixed `$var`)

Finds whether the given variable is a native unicode or binary string.

Parameters

`var`
The variable being evaluated.

Return Values

Returns **TRUE** if *`var`* is a native unicode or binary string, **FALSE** otherwise.

See Also

- is_binary()
- is_string()
- is_unicode()

is_callable

is_callable -- Verify that the contents of a variable can be called as a function

Description

bool **is_callable** (mixed `$var` [, bool `$syntax_only` [, string `&$callable_name`]])

Verify that the contents of a variable can be called as a function. This can check that a simple variable contains the name of a valid function, or that an array contains a properly encoded object and function name.

Parameters

`var`
Can be either the name of a function stored in a string variable, or an object and the name of a method within the object, like this:
`array($SomeObject, 'MethodName')`

`syntax_only`
If set to **TRUE** the function only verifies that *`var`* might be a function or method. It will only reject simple variables that are not strings, or an array that does not have a valid structure to be used as a callback. The valid ones are supposed to have only 2 entries, the first of which is an object or a string, and the second a string.

`callable_name`
Receives the "callable name". In the example below it is "someClass::someMethod". Note, however, that despite the implication that someClass::SomeMethod() is a callable static method, this is not the case.

Return Values

Returns **TRUE** if *`var`* is callable, **FALSE** otherwise.

Examples

Example #12 - is_callable() example

```
<?php
//  How to check a variable to see if it can be called
//  as a function.

//
//  Simple variable containing a function
//
```

```
function someFunction()
{
}

$functionVariable = 'someFunction';

var_dump(is_callable($functionVariable, false, $callable_name));  //
bool(true)

echo $callable_name, "\n";  // someFunction

//
//  Array containing a method
//

class someClass {

 function someMethod()
 {
 }

}

$anObject = new someClass();

$methodVariable = array($anObject, 'someMethod');

var_dump(is_callable($methodVariable, true, $callable_name));  //
bool(true)

echo $callable_name, "\n";  //  someClass::someMethod

?>
```

See Also

- function_exists()
- method_exists()

is_double

is_double -- Alias of is_float()

Description

This function is an alias of: is_float().

is_float

is_float -- Finds whether the type of a variable is float

Description

bool **is_float** (mixed `$var`)

Finds whether the type of the given variable is float.

Note
To test if a variable is a number or a numeric string (such as form input, which is always a string), you must use is_numeric().

Parameters

var
The variable being evaluated.

Return Values

Returns **TRUE** if *var* is a float, **FALSE** otherwise.

Examples

Example #13 - is_float() example

```
<?php
if(is_float(27.25)) {
echo "is float\n";
}else {
echo "is not float\n";
}
var_dump(is_float('abc'));
var_dump(is_float(23));
var_dump(is_float(23.5));
var_dump(is_float(1e7));  //Scientific Notation
var_dump(is_float(true));
?>
```

The above example will output:

```
is float
bool(false)
```

```
bool(false)
bool(true)
bool(true)
bool(false)
```

See Also

- is_bool()
- is_int()
- is_numeric()
- is_string()
- is_array()
- is_object()

is_int

is_int -- Find whether the type of a variable is integer

Description

bool **is_int** (mixed `$var`)

Finds whether the type of the given variable is integer.

Note
To test if a variable is a number or a numeric string (such as form input, which is always a string), you must use is_numeric().

Parameters

var
The variable being evaluated.

Return Values

Returns **TRUE** if *var* is an integer, **FALSE** otherwise.

Examples

Example #14 - is_int() example

```
<?php
if (is_int(23)) {
echo "is integer\n";
} else {
echo "is not an integer\n";
}
var_dump(is_int(23));
var_dump(is_int("23"));
var_dump(is_int(23.5));
var_dump(is_int(true));
?>
```

The above example will output:

```
is integer
bool(true)
bool(false)
```

```
bool(false)
bool(false)
```

See Also

- is_bool()
- is_float()
- is_numeric()
- is_string()
- is_array()
- is_object()

is_integer

is_integer -- Alias of is_int()

Description

This function is an alias of: is_int().

is_long

is_long -- Alias of is_int()

Description

This function is an alias of: is_int().

is_null

is_null -- Finds whether a variable is **NULL**

Description

bool **is_null** (mixed `$var`)

Finds whether the given variable is **NULL**.

Parameters

`var`
The variable being evaluated.

Return Values

Returns **TRUE** if *`var`* is null, **FALSE** otherwise.

See Also

- The **NULL** type
- isset()
- is_bool()
- is_numeric()
- is_float()
- is_int()
- is_string()
- is_object()
- is_array()

is_numeric

is_numeric -- Finds whether a variable is a number or a numeric string

Description

bool **is_numeric** (mixed `$var`)

Finds whether the given variable is numeric. Numeric strings consist of optional sign, any number of digits, optional decimal part and optional exponential part. Thus *+0123.45e6* is a valid numeric value. Hexadecimal notation (*0xFF*) is allowed too but only without sign, decimal and exponential part.

Parameters

`var`
The variable being evaluated.

Return Values

Returns **TRUE** if *`var`* is a number or a numeric string, **FALSE** otherwise.

See Also

- ctype_digit()
- is_bool()
- is_null()
- is_float()
- is_int()
- is_string()
- is_object()
- is_array()

is_object

is_object -- Finds whether a variable is an object

Description

bool **is_object** (mixed `$var`)

Finds whether the given variable is an object.

Parameters

var
The variable being evaluated.

Return Values

Returns **TRUE** if *var* is an object, **FALSE** otherwise.

Notes

Note
This function will return **FALSE** if used on an unserialized object where the class definition is not present (even though gettype() returns object).

See Also

- is_bool()
- is_int()
- is_float()
- is_string()
- is_array()

is_real

is_real -- Alias of is_float()

Description

This function is an alias of: is_float().

is_resource

is_resource -- Finds whether a variable is a resource

Description

bool **is_resource** (mixed `$var`)

Finds whether the given variable is a resource.

Parameters

var
The variable being evaluated.

Return Values

Returns **TRUE** if *var* is a resource, **FALSE** otherwise.

Examples

Example #15 - is_resource() example

```
<?php

$db_link = @mysql_connect('localhost', 'mysql_user', 'mysql_pass');
if (!is_resource($db_link)) {
   die('Can\'t connect : ' . mysql_error());
}

?>
```

See Also

- The resource-type documentation
- get_resource_type()

is_scalar

is_scalar -- Finds whether a variable is a scalar

Description

bool **is_scalar** (mixed `$var`)

Finds whether the given variable is a scalar.

Scalar variables are those containing an integer, float, string or boolean. Types array, object and resource are not scalar.

Note
is_scalar() does not consider resource type values to be scalar as resources are abstract datatypes which are currently based on integers. This implementation detail should not be relied upon, as it may change.

Parameters

var
The variable being evaluated.

Return Values

Returns **TRUE** if *var* is a scalar **FALSE** otherwise.

Examples

Example #16 - is_scalar() example

```
<?php
function show_var($var)
{
   if (is_scalar($var)) {
       echo $var;
   } else {
       var_dump($var);
   }
}
$pi = 3.1416;
$proteins = array("hemoglobin", "cytochrome c oxidase", "ferredoxin");
```

```
show_var($pi);
show_var($proteins)

?>
```

The above example will output:

```
3.1416
array(3) {
 [0]=>
 string(10) "hemoglobin"
 [1]=>
 string(20) "cytochrome c oxidase"
 [2]=>
 string(10) "ferredoxin"
}
```

See Also

- is_float()
- is_int()
- is_numeric()
- is_real()
- is_string()
- is_bool()
- is_object()
- is_array()

is_string

is_string -- Find whether the type of a variable is string

Description

bool **is_string** (mixed `$var`)

Finds whether the type given variable is string.

Parameters

var
The variable being evaluated.

Return Values

Returns **TRUE** if *var* is of type string, **FALSE** otherwise.

Examples

Example #17 - is_string() example

```
<?php
if (is_string("23")) {
echo "is string\n";
} else {
echo "is not an string\n";
}
var_dump(is_string('abc'));
var_dump(is_string("23"));
var_dump(is_string(23.5));
var_dump(is_string(true));
?>
```

The above example will output:

```
is string
bool(true)
bool(true)
bool(false)
bool(false)
```

See Also

- is_float()
- is_int()
- is_bool()
- is_object()
- is_array()

is_unicode

is_unicode -- Finds whether a variable is a unicode string

Description

bool **is_unicode** (mixed `$var`)

Finds whether the given variable is a unicode string.

Parameters

`var`
The variable being evaluated.

Return Values

Returns **TRUE** if *`var`* is a unicode string, **FALSE** otherwise.

See Also

- is_binary()
- is_buffer()
- is_string()
- unicode_encode()

isset

isset -- Determine whether a variable is set

Description

bool **isset** (mixed `$var` [, mixed `$var` [, `$...`]])

Determine whether a variable is set.

If a variable has been unset with unset(), it will no longer be set. isset() will return **FALSE** if testing a variable that has been set to **NULL**. Also note that a **NULL** byte (*"\0"*) is not equivalent to the PHP **NULL** constant.

If multiple parameters are supplied then isset() will return **TRUE** only if all of the parameters are set. Evaluation goes from left to right and stops as soon as an unset variable is encountered.

Parameters

var
The variable to be checked.

var
Another variable ..

...

Return Values

Returns **TRUE** if *var* exists; **FALSE** otherwise.

Examples

Example #18 - isset() Examples

```
<?php

$var = '';

// This will evaluate to TRUE so the text will be printed.
if (isset($var)) {
   echo "This var is set so I will print.";
}
```

```
// In the next examples we'll use var_dump to output
// the return value of isset().

$a = "test";
$b = "anothertest";

var_dump(isset($a));      // TRUE
var_dump(isset($a, $b)); // TRUE

unset ($a);

var_dump(isset($a));     // FALSE
var_dump(isset($a, $b)); // FALSE

$foo = NULL;
var_dump(isset($foo));   // FALSE

?>
```

This also work for elements in arrays:

```
<?php

$a = array ('test' => 1, 'hello' => NULL);

var_dump(isset($a['test']));            // TRUE
var_dump(isset($a['foo']));             // FALSE
var_dump(isset($a['hello']));           // FALSE

// The key 'hello' equals NULL so is considered unset
// If you want to check for NULL key values then try:
var_dump(array_key_exists('hello', $a)); // TRUE

?>
```

Notes

Warning

isset() only works with variables as passing anything else will result in a parse error. For checking if constants are set use the defined() function.

Note

Because this is a language construct and not a function, it cannot be called using variable functions

See Also

- empty()
- unset()
- defined()
- the type comparison tables
- array_key_exists()
- is_null()
- the error control @ operator

print_r

print_r -- Prints human-readable information about a variable

Description

mixed **print_r** (mixed `$expression` [, bool `$return`])

print_r() displays information about a variable in a way that's readable by humans.

print_r(), var_dump() and var_export() will also show protected and private properties of objects with PHP 5. Static class members will not be shown.

Remember that print_r() will move the array pointer to the end. Use reset() to bring it back to beginning.

Parameters

`expression`
The expression to be printed.

`return`
If you would like to capture the output of print_r(), use the *`return`* parameter. If this parameter is set to **TRUE**, print_r() will return its output, instead of printing it (which it does by default).

Return Values

If given a string, integer or float, the value itself will be printed. If given an array, values will be presented in a format that shows keys and elements. Similar notation is used for object s.

Notes

Note
This function uses internal output buffering with this parameter so it can not be used inside an ob_start() callback function.

ChangeLog

Version	**Description**

4.3.0	The *return* parameter was added. If you need to capture the output of print_r() with an older version of PHP prior, use the output-control functions.
4.0.4	Prior to PHP 4.0.4, print_r() will continue forever if given an array or object that contains a direct or indirect reference to itself. An example is *print_r($GLOBALS)* because *$GLOBALS* is itself a global variable that contains a reference to itself.

Examples

Example #19 - print_r() example

```
<pre>
<?php
$a = array ('a' => 'apple', 'b' => 'banana', 'c' => array ('x', 'y', 'z'));
print_r ($a);
?>
</pre>
```

The above example will output:

```
<pre>
Array
(
   [a] => apple
   [b] => banana
   [c] => Array
       (
           [0] => x
           [1] => y
           [2] => z
       )
)
</pre>
```

Example #20 - *return* parameter example

```
<?php
$b = array ('m' => 'monkey', 'foo' => 'bar', 'x' => array ('x', 'y', 'z'));
$results = print_r($b, true); // $results now contains output from print_r
?>
```

See Also

- ob_start()
- var_dump()
- var_export()

serialize

serialize -- Generates a storable representation of a value

Description

string **serialize** (mixed `$value`)

Generates a storable representation of a value

This is useful for storing or passing PHP values around without losing their type and structure.

To make the serialized string into a PHP value again, use unserialize().

Parameters

value
: The value to be serialized. serialize() handles all types, except the resource -type. You can even serialize() arrays that contain references to itself. Circular references inside the array/object you are serialize() ing will also be stored. Any other reference will be lost. When serializing objects, PHP will attempt to call the member function __sleep prior to serialization. This is to allow the object to do any last minute clean-up, etc. prior to being serialized. Likewise, when the object is restored using unserialize() the __wakeup member function is called.

Return Values

Returns a string containing a byte-stream representation of *value* that can be stored anywhere.

Examples

Example #21 - serialize() example

```
<?php
// $session_data contains a multi-dimensional array with session
// information for the current user.  We use serialize() to store
// it in a database at the end of the request.

$conn = odbc_connect("webdb", "php", "chicken");
$stmt = odbc_prepare($conn,
     "UPDATE sessions SET data = ? WHERE id = ?");
$sqldata = array (serialize($session_data), $_SERVER['PHP_AUTH_USER']);
if (!odbc_execute($stmt, &$sqldata)) {
   $stmt = odbc_prepare($conn,
```

```
    "INSERT INTO sessions (id, data) VALUES(?, ?)");
   if (!odbc_execute($stmt, &$sqldata)) {
       /* Something went wrong.. */
   }
}
?>
```

ChangeLog

Version	Description
4.0.7	The object serialization process was fixed.
4.0.0	When serializing an object, methods are not lost anymore. Please see the Serializing Objects for more information.

Notes

Note
It is not possible to serialize PHP built-in objects.

See Also

- unserialize()
- Serializing Objects

settype

settype -- Set the type of a variable

Description

bool **settype** (mixed `&$var`, string `$type`)

Set the type of variable *`var`* to *`type`*.

Parameters

`var`
The variable being converted.

`type`
Possibles values of *`type`* are:

- "boolean" (or, since PHP 4.2.0, "bool")
- "integer" (or, since PHP 4.2.0, "int")
- "float" (only possible since PHP 4.2.0, for older versions use the deprecated variant "double")
- "string"
- "array"
- "object"
- "null" (since PHP 4.2.0)

Return Values

Returns **TRUE** on success or **FALSE** on failure.

Examples

Example #22 - settype() example

```
<?php
$foo = "5bar"; // string
$bar = true;   // boolean

settype($foo, "integer"); // $foo is now 5   (integer)
settype($bar, "string");  // $bar is now "1" (string)
```

```
?>
```

Notes

Note
Maximum value for "int" is **PHP_INT_MAX**.

See Also

- gettype()
- type-casting
- type-juggling

strval

strval -- Get string value of a variable

Description

string **strval** (mixed `$var`)

Get the string value of a variable. See the documentation on string for more information on converting to string.

Parameters

`var`
: The variable that is being converted to a string. *`var`* may be any scalar type. You cannot use strval() on arrays or objects.

Return Values

The string value of *`var`*.

See Also

- floatval()
- intval()
- settype()
- Type juggling

unserialize

unserialize -- Creates a PHP value from a stored representation

Description

mixed **unserialize** (string `$str`)

unserialize() takes a single serialized variable and converts it back into a PHP value.

Parameters

`str`
The serialized string. If the variable being unserialized is an object, after successfully reconstructing the object PHP will automatically attempt to call the **__wakeup()** member function (if it exists).

Note
unserialize_callback_func directive It's possible to set a callback-function which will be called, if an undefined class should be instantiated during unserializing. (to prevent getting an incomplete object "__PHP_Incomplete_Class".) Use your *php.ini*, ini_set() or.*htaccess* to define 'unserialize_callback_func'. Everytime an undefined class should be instantiated, it'll be called. To disable this feature just empty this setting.

Return Values

The converted value is returned, and can be a boolean, integer, float, string, array or object.

In case the passed string is not unserializeable, **FALSE** is returned and *E_NOTICE* is issued.

ChangeLog

Version	Description
4.2.0	The directive unserialize_callback_func directive became available.

4.0.0	When serializing an object, methods are not lost anymore. Please see the Serializing Objects for more information.

Examples

Example #23 - unserialize() example

```
<?php
// Here, we use unserialize() to load session data to the
// $session_data array from the string selected from a database.
// This example complements the one described with serialize().

$conn = odbc_connect("webdb", "php", "chicken");
$stmt = odbc_prepare($conn, "SELECT data FROM sessions WHERE id = ?");
$sqldata = array($_SERVER['PHP_AUTH_USER']);
if (!odbc_execute($stmt, &$sqldata) || !odbc_fetch_into($stmt, &$tmp)) {
   // if the execute or fetch fails, initialize to empty array
   $session_data = array();
} else {
   // we should now have the serialized data in $tmp[0].
   $session_data = unserialize($tmp[0]);
   if (!is_array($session_data)) {
       // something went wrong, initialize to empty array
       $session_data = array();
   }
}
?>
```

Example #24 - unserialize_callback_func example

```
<?php
$serialized_object='O:1:"a":1:{s:5:"value";s:3:"100";}';

// unserialize_callback_func directive available as of PHP 4.2.0
ini_set('unserialize_callback_func', 'mycallback'); // set your
callback_function

function mycallback($classname)
{
   // just include a file containing your classdefinition
   // you get $classname to figure out which classdefinition is required
}
?>
```

Notes

Warning
FALSE is returned both in the case of an error and if unserializing the serialized **FALSE** value. It is possible to catch this special case by comparing `str` with *serialize(false)* or by catching the issued *E_NOTICE*.

See Also

- serialize()

unset

unset -- Unset a given variable

Description

void **unset** (mixed `$var` [, mixed `$var` [, mixed `$...`]])

unset() destroys the specified variables.

The behavior of unset() inside of a function can vary depending on what type of variable you are attempting to destroy.

If a globalized variable is unset() inside of a function, only the local variable is destroyed. The variable in the calling environment will retain the same value as before unset() was called.

```
<?php
function destroy_foo()
{
   global $foo;
   unset($foo);
}

$foo = 'bar';
destroy_foo();
echo $foo;
?>
```

The above example will output:

```
bar
```

If you would like to unset() a global variable inside of a function, you can use the *$GLOBALS* array to do so:

```
<?php
function foo()
{
   unset($GLOBALS['bar']);
}

$bar = "something";
foo();
?>
```

If a variable that is PASSED BY REFERENCE is unset() inside of a function, only the local variable is destroyed. The variable in the calling environment will retain the same value as before unset() was called.

```
<?php
function foo(&$bar)
{
   unset($bar);
   $bar = "blah";
}

$bar = 'something';
echo "$bar\n";

foo($bar);
echo "$bar\n";
?>
```

The above example will output:

```
something
something
```

If a static variable is unset() inside of a function, unset() destroys the variable only in the context of the rest of a function. Following calls will restore the previous value of a variable.

```
<?php
function foo()
{
   static $bar;
   $bar++;
   echo "Before unset: $bar, ";
   unset($bar);
   $bar = 23;
   echo "after unset: $bar\n";
}

foo();
foo();
foo();
?>
```

The above example will output:

```
Before unset: 1, after unset: 23
Before unset: 2, after unset: 23
```

```
Before unset: 3, after unset: 23
```

Parameters

var
The variable to be unset.

var
Another variable ..

...

Return Values

No value is returned.

ChangeLog

Version	Description
4.0.0	unset() became an expression. (In PHP 3, unset() would always return 1).

Examples

Example #25 - unset() example

```
<?php
// destroy a single variable
unset($foo);

// destroy a single element of an array
unset($bar['quux']);

// destroy more than one variable
unset($foo1, $foo2, $foo3);
?>
```

Notes

Note
Because this is a language construct and not a function, it cannot be called using variable functions

Note
It is possible to unset even object properties visible in current context.

Note
It is not possible to unset *$this* inside an object method since PHP 5.

See Also

- isset()
- empty()
- array_splice()

var_dump

var_dump -- Dumps information about a variable

Description

void **var_dump** (mixed `$expression` [, mixed `$expression` [, `$...`]])

This function displays structured information about one or more expressions that includes its type and value. Arrays and objects are explored recursively with values indented to show structure.

In PHP 5 all public, private and protected properties of objects will be returned in the output.

Tip
As with anything that outputs its result directly to the browser, the output-control functions can be used to capture the output of this function, and save it in a string (for example).

Parameters

`expression`
The variable you want to export.

Return Values

No value is returned.

Examples

Example #26 - var_dump() example

```
<?php
$a = array(1, 2, array("a", "b", "c"));
var_dump($a);
?>
```

The above example will output:

```
array(3) {
 [0]=>
 int(1)
```

```
 [1]=>
 int(2)
 [2]=>
 array(3) {
   [0]=>
   string(1) "a"
   [1]=>
   string(1) "b"
   [2]=>
   string(1) "c"
 }
}

<?php

$b = 3.1;
$c = true;
var_dump($b, $c);

?>
```

The above example will output:

```
float(3.1)
bool(true)
```

See Also

- var_export()
- print_r()

var_export

var_export -- Outputs or returns a parsable string representation of a variable

Description

mixed **var_export** (mixed `$expression` [, bool `$return`])

var_export() gets structured information about the given variable. It is similar to var_dump() with one exception: the returned representation is valid PHP code.

Parameters

`expression`
The variable you want to export.

`return`
If used and set to **TRUE**, var_export() will return the variable representation instead of outputing it.

Note
This function uses internal output buffering with this parameter so it can not be used inside an ob_start() callback function.

Return Values

Returns the variable representation when the *`return`* parameter is used and evaluates to **TRUE**. Otherwise, this function will return **NULL**.

ChangeLog

Version	Description
5.1.0	Possibility to export classes and arrays containing classes using the __set_state magic method.

Examples

Example #27 - var_export() Examples

```
<?php
$a = array (1, 2, array ("a", "b", "c"));
var_export($a);
?>
```

The above example will output:

```
array (
 0 => 1,
 1 => 2,
 2 =>
 array (
   0 => 'a',
   1 => 'b',
   2 => 'c',
 ),
)
```

```
<?php

$b = 3.1;
$v = var_export($b, true);
echo $v;

?>
```

The above example will output:

```
3.1
```

Example #28 - Exporting classes since PHP 5.1.0

```
<?php
class A { public $var; }
$a = new A;
$a->var = 5;
var_export($a);
?>
```

The above example will output:

```
A::__set_state(array(
  'var' => 5,
))
```

Example #29 - Using __set_state (since PHP 5.1.0)

```
<?php
class A
{
   public $var1;
   public $var2;

   public static function __set_state($an_array)
   {
       $obj = new A;
       $obj->var1 = $an_array['var1'];
       $obj->var2 = $an_array['var2'];
       return $obj;
   }
}

$a = new A;
$a->var1 = 5;
$a->var2 = 'foo';

eval('$b = ' . var_export($a, true) . ';'); // $b = A::__set_state(array(
                                            //    'var1' => 5,
                                            //    'var2' => 'foo',
                                            // ));
var_dump($b);
?>
```

The above example will output:

```
object(A)#2 (2) {
 ["var1"]=>
 int(5)
 ["var2"]=>
 string(3) "foo"
}
```

Notes

Note

Variables of type resource couldn't be exported by this function.

Note

var_export() does not handle circular references as it would be close to impossible to generate parsable PHP code for that. If you want to do something with the full representation of an array or object, use serialize().

See Also

- print_r()
- serialize()
- var_dump()