Other Services
Search Engine Extensions
Server Specific Extensions
Session Extensions
Text Processing
Variable and Type Related Extensions
Web Services
Windows Only Extensions
XML Manipulation
GUI Extensions

Keyboard Shortcuts
?
This help
j
Next menu item
k
Previous menu item
g p
Previous man page
g n
Next man page
G
Scroll to bottom
g g
Scroll to top
g h
Goto homepage
g s
Goto search
(current page)
/
Focus search box

Returning values »
« User-defined functions

- PHP Manual
- Language Reference
- Functions

Change language: English

Edit Report a Bug

Function arguments ¶

Information may be passed to functions via the argument list, which is a comma-delimited list of expressions. The arguments are evaluated from left to right.

PHP supports passing arguments by value (the default), passing by reference, and default argument values. Variable-length argument lists are also supported.

Example #1 Passing arrays to functions

```
<?php
function takes_array($input)
{
    echo "$input[0] + $input[1] = ", $input[0]+$input[1];
}
?>
```

Passing arguments by reference ¶

By default, function arguments are passed by value (so that if the value of the argument within the function is changed, it does not get changed outside of the function). To allow a function to modify its arguments, they must be passed by reference.

To have an argument to a function always passed by reference, prepend an ampersand (&) to the argument name in the function definition:

Example #2 Passing function parameters by reference

```
<?php
function add_some_extra(&$string)
{
    $string .= 'and something extra.';
}
$str = 'This is a string, ';
add_some_extra($str);
echo $str;    // outputs 'This is a string, and something extra.'
?>
```

Default argument values ¶

A function may define C++-style default values for scalar arguments as follows:

Example #3 Use of default parameters in functions

```
<?php
function makecoffee($type = "cappuccino")
{
    return "Making a cup of $type.\n";
}
echo makecoffee();
echo makecoffee(null);
echo makecoffee("espresso");
?>
```

The above example will output:

```
Making a cup of cappuccino.
Making a cup of .
Making a cup of espresso.
```

PHP also allows the use of arrays and the special type `NULL` as default values, for example:

Example #4 Using non-scalar types as default values

```
<?php
function makecoffee($types = array("cappuccino"), $coffeeMaker = NULL)
{
    $device = is_null($coffeeMaker) ? "hands" : $coffeeMaker;
    return "Making a cup of ".join(", ", $types)." with $device.\n";
}
echo makecoffee();
echo makecoffee(array("cappuccino", "lavazza"), "teapot");
?>
```

The default value must be a constant expression, not (for example) a variable, a class member or a function call.

Note that when using default arguments, any defaults should be on the right side of any non-default arguments; otherwise, things will not work as expected. Consider the following code snippet:

Example #5 Incorrect usage of default function arguments

```
<?php
function makeyogurt($type = "acidophilus", $flavour)
{
    return "Making a bowl of $type $flavour.\n";
}

echo makeyogurt("raspberry");   // won't work as expected
?>
```

The above example will output:

```
Warning: Missing argument 2 in call to makeyogurt() in
/usr/local/etc/httpd/htdocs/phptest/functest.html on line 41
Making a bowl of raspberry .
```

Now, compare the above with this:

Example #6 Correct usage of default function arguments

```
<?php
function makeyogurt($flavour, $type = "acidophilus")
{
    return "Making a bowl of $type $flavour.\n";
}

echo makeyogurt("raspberry");   // works as expected
?>
```

The above example will output:

```
Making a bowl of acidophilus raspberry.
```

> **Note**: As of PHP 5, arguments that are passed by reference may have a default value.

Type declarations ¶

> **Note**:
>
> Type declarations were also known as type hints in PHP 5.

Type declarations allow functions to require that parameters are of a certain type at call time. If the given value is of the incorrect type, then an error is generated: in PHP 5, this will be a recoverable fatal error, while PHP 7 will throw a TypeError exception.

To specify a type declaration, the type name should be added before the parameter name. The declaration can be made to accept **NULL** values if the default value of the parameter is set to **NULL**.

Valid types ¶

Type	Description	Minimum PHP version
Class/interface name	The parameter must be an *instanceof* the given class or interface name.	PHP 5.0.0
self	The parameter must be an *instanceof* the same class as the one the method is defined on. This can only be used on class and instance methods.	PHP 5.0.0
array	The parameter must be an array.	PHP 5.1.0
callable	The parameter must be a valid callable.	PHP 5.4.0
bool	The parameter must be a boolean value.	PHP 7.0.0
float	The parameter must be a floating point number.	PHP 7.0.0
int	The parameter must be an integer.	PHP 7.0.0
string	The parameter must be a string.	PHP 7.0.0
iterable	The parameter must be either an array or an *instanceof* Traversable.	PHP 7.1.0
object	The parameter must be an object.	PHP 7.2.0

Warning

Aliases for the above scalar types are not supported. Instead, they are treated as class or interface names. For example, using *boolean* as a parameter or return type will require an argument or return value that is an *instanceof* the class or interface *boolean*, rather than of type bool:

```
<?php
 function test(boolean $param) {}
 test(true);
 ?>
```

The above example will output:

```
 Fatal error: Uncaught TypeError: Argument 1 passed to test() must be an instance of boolean, boolean given, called in - on line 1 and defined in -:1
```

Examples ¶

Example #7 Basic class type declaration

```
<?php
class C {}
class D extends C {}

// This doesn't extend C.
class E {}

function f(C $c) {
    echo get_class($c)."\n";
}

f(new C);
f(new D);
f(new E);
?>
```

The above example will output:

```
C
D

Fatal error: Uncaught TypeError: Argument 1 passed to f() must be an instance of C, instance of E given, called in - on line 14 and defined in -:8
Stack trace:
#0 -(14): f(Object(E))
#1 {main}
  thrown in - on line 8
```

Example #8 Basic interface type declaration

```
<?php
interface I { public function f(); }
class C implements I { public function f() {} }

// This doesn't implement I.
class E {}

function f(I $i) {
    echo get_class($i)."\n";
}

f(new C);
f(new E);
?>
```

The above example will output:

```
C

Fatal error: Uncaught TypeError: Argument 1 passed to f() must implement interface I, instance of E given, called in - on line 13 and defined in -:8
Stack trace:
#0 -(13): f(Object(E))
#1 {main}
  thrown in - on line 8
```

Example #9 Typed pass-by-reference Parameters

Declared types of reference parameters are checked on function entry, but not when the function returns, so after the function had returned, the argument's type may have changed.

```
<?php
function array_baz(array &$param)
{
    $param = 1;
}
$var = [];
array_baz($var);
var_dump($var);
array_baz($var);
?>
```

The above example will output something similar to:

```
int(1)

Fatal error: Uncaught TypeError: Argument 1 passed to array_baz() must be of the type array, int given, called in %s on line %d
```

Example #10 Nullable type declaration

```
<?php
class C {}

function f(C $c = null) {
    var_dump($c);
}

f(new C);
f(null);
?>
```

The above example will output:

```
object(C)#1 (0) {
}
NULL
```

Strict typing ¶

By default, PHP will coerce values of the wrong type into the expected scalar type if possible. For example, a function that is given an integer for a parameter that expects a string will get a variable of type string.

It is possible to enable strict mode on a per-file basis. In strict mode, only a variable of exact type of the type declaration will be accepted, or a TypeError will be thrown. The only exception to this rule is that an integer may be given to a function expecting a float. Function calls from within internal functions will not be affected by the *strict_types* declaration.

To enable strict mode, the *declare* statement is used with the *strict_types* declaration:

Caution

Enabling strict mode will also affect return type declarations.

> **Note**:
>
> Strict typing applies to function calls made from *within* the file with strict typing enabled, not to the functions declared within that file. If a file without strict typing enabled makes a call to a function that was defined in a file with strict typing, the caller's preference (weak typing) will be respected, and the value will be coerced.
>
> **Note**:
>
> Strict typing is only defined for scalar type declarations, and as such, requires PHP 7.0.0 or later, as scalar type declarations were added in that version.

Example #11 Strict typing

```
<?php
declare(strict_types=1);

function sum(int $a, int $b) {
    return $a + $b;
}
```

```
var_dump(sum(1, 2));
var_dump(sum(1.5, 2.5));
?>
```

The above example will output:

```
int(3)

Fatal error: Uncaught TypeError: Argument 1 passed to sum() must be of the type integer, float given, called in - on line 9 and defined in -:4
Stack trace:
#0 -(9): sum(1.5, 2.5)
#1 {main}
  thrown in - on line 4
```

Example #12 Weak typing

```
<?php
function sum(int $a, int $b) {
    return $a + $b;
}

var_dump(sum(1, 2));

// These will be coerced to integers: note the output below!
var_dump(sum(1.5, 2.5));
?>
```

The above example will output:

```
int(3)
int(3)
```

Example #13 Catching TypeError

```
<?php
declare(strict_types=1);

function sum(int $a, int $b) {
    return $a + $b;
}

try {
    var_dump(sum(1, 2));
    var_dump(sum(1.5, 2.5));
} catch (TypeError $e) {
    echo 'Error: '.$e->getMessage();
}
?>
```

The above example will output:

```
int(3)
Error: Argument 1 passed to sum() must be of the type integer, float given, called in - on line 10
```

Variable-length argument lists ¶

PHP has support for variable-length argument lists in user-defined functions. This is implemented using the ... token in PHP 5.6 and later, and using the func_num_args(), func_get_arg(), and func_get_args() functions in PHP 5.5 and earlier.

... in PHP 5.6+ ¶

In PHP 5.6 and later, argument lists may include the ... token to denote that the function accepts a variable number of arguments. The arguments will be passed into the given variable as an array; for example:

Example #14 Using ... to access variable arguments

```
<?php
function sum(...$numbers) {
    $acc = 0;
    foreach ($numbers as $n) {
        $acc += $n;
    }
    return $acc;
}

echo sum(1, 2, 3, 4);
?>
```

The above example will output:

```
10
```

You can also use ... when calling functions to unpack an array or Traversable variable or literal into the argument list:

Example #15 Using ... to provide arguments

```
<?php
function add($a, $b) {
    return $a + $b;
}

echo add(...[1, 2])."\n";

$a = [1, 2];
echo add(...$a);
?>
```

The above example will output:

```
3
3
```

You may specify normal positional arguments before the ... token. In this case, only the trailing arguments that don't match a positional argument will be added to the array generated by ....

It is also possible to add a type hint before the ... token. If this is present, then all arguments captured by ... must be objects of the hinted class.

Example #16 Type hinted variable arguments

```
<?php
function total_intervals($unit, DateInterval ...$intervals) {
    $time = 0;
    foreach ($intervals as $interval) {
        $time += $interval->$unit;
    }
    return $time;
}

$a = new DateInterval('P1D');
$b = new DateInterval('P2D');
echo total_intervals('d', $a, $b).' days';

// This will fail, since null isn't a DateInterval object.
echo total_intervals('d', null);
?>
```

The above example will output:

```
3 days
Catchable fatal error: Argument 2 passed to total_intervals() must be an instance of DateInterval, null given, called in - on line 14 and defined in - on l
```

Finally, you may also pass variable arguments by reference by prefixing the ... with an ampersand (&).

Older versions of PHP ¶

No special syntax is required to note that a function is variadic; however access to the function's arguments must use func_num_args(), func_get_arg() and func_get_args().

The first example above would be implemented as follows in PHP 5.5 and earlier:

Example #17 Accessing variable arguments in PHP 5.5 and earlier

```
<?php
function sum() {
    $acc = 0;
    foreach (func_get_args() as $n) {
        $acc += $n;
    }
    return $acc;
}

echo sum(1, 2, 3, 4);
?>
```

The above example will output:

```
10
```

add a note

User Contributed Notes 31 notes

up
down
86
php at richardneill dot org ¶
4 years ago

To experiment on performance of pass-by-reference and pass-by-value, I used this script. Conclusions are below.

```
#!/usr/bin/php
<?php
function sum($array,$max){   //For Reference, use:  "&$array"
    $sum=0;
    for ($i=0; $i<2; $i++){
        #$array[$i]++;        //Uncomment this line to modify the array within the function.
        $sum += $array[$i];
    }
    return ($sum);
}

$max = 1E7                  //10 M data points.
$data = range(0,$max,1);

$start = microtime(true);
for ($x = 0 ; $x < 100; $x++){
    $sum = sum($data, $max);
}
$end =  microtime(true);
echo "Time: ".($end - $start)." s\n";

/* Run times:
#    PASS BY    MODIFIED?   Time
-    -------    ---------   ----
1    value      no          56 us
2    reference  no          58 us

3    valuue     yes         129 s
4    reference  yes         66 us

Conclusions:

1. PHP is already smart about zero-copy / copy-on-write. A function call does NOT copy the data unless it needs to; the data is
   only copied on write. That's why  #1 and #2 take similar times, whereas #3 takes 2 million times longer than #4.
   [You never need to use &$array to ask the compiler to do a zero-copy optimisation; it can work that out for itself.]

2. You do use &$array  to tell the compiler "it is OK for the function to over-write my argument in place, I don't need the original
   any more." This can make a huge difference to performance when we have large amounts of memory to copy.
   (This is the only way it is done in C, arrays are always passed as pointers)

3. The other use of & is as a way to specify where data should be *returned*. (e.g. as used by exec() ).
   (This is a C-like way of passing pointers for outputs, whereas PHP functions normally return complex types, or multiple answers
   in an array)

4. It's  unhelpful that only the function definition has &. The caller should have it, at least as syntactic sugar. Otherwise
   it leads to unreadable code: because the person reading the function call doesn't expect it to pass by reference. At the moment,
   it's necessary to write a by-reference function call with a comment, thus:
    $sum = sum($data,$max);  //warning, $data passed by reference, and may be modified.

5. Sometimes, pass by reference could be at the choice of the caller, NOT the function definitition. PHP doesn't allow it, but it
   would be meaningful for the caller to decide to pass data in as a reference. i.e. "I'm done with the variable, it's OK to stomp
   on it in memory".
*/
?>
```

up
down
26
gabriel at figdice dot org ¶
3 years ago

A function's argument that is an object, will have its properties modified by the function although you don't need to pass it by reference.

```
<?php
$x = new stdClass();
$x->prop = 1;

function f ( $o ) // Notice the absence of &
{
  $o->prop ++;
}

f($x);

echo $x->prop; // shows: 2
?>
```

This is different for arrays:

```
<?php
$y = [ 'prop' => 1 ];
```

```
function g( $a )
{
  $a['prop'] ++;
  echo $a['prop'];  // shows: 2
}

g($y);

echo $y['prop'];  // shows: 1
?>
```

up
down
8

Hayley Watson ¶

2 years ago

There are fewer restrictions on using ... to supply multiple arguments to a function call than there are on using it to declare a variadic parameter in the function declaration. In particular, it can be used more than once to unpack arguments, provided that all such uses come after any positional arguments.

```
<?php

$array1 = [[1],[2],[3]];
$array2 = [4];
$array3 = [[5],[6],[7]];

$result = array_merge(...$array1); // Legal, of course: $result == [1,2,3];
$result = array_merge($array2, ...$array1); // $result == [4,1,2,3]
$result = array_merge(...$array1, $array2); // Fatal error: Cannot use positional argument after argument unpacking.
$result = array_merge(...$array1, ...$array3); // Legal! $result == [1,2,3,5,6,7]
?>
```

The Right Thing for the error case above would be for $result==[1,2,3,4], but this isn't yet (v7.1.8) supported.

up
down
15

jcaplan at bogus dot amazon dot com ¶

13 years ago

In function calls, PHP clearly distinguishes between missing arguments and present but empty arguments. Thus:

```
<?php
function f( $x = 4 ) { echo $x . "\\n"; }
f(); // prints 4
f( null ); // prints blank line
f( $y ); // $y undefined, prints blank line
?>
```

The utility of the optional argument feature is thus somewhat diminished. Suppose you want to call the function f many times from function g, allowing the caller of g to specify if f should be called with a specific value or with its default value:

```
<?php
function f( $x = 4 ) {echo $x . "\\n"; }

// option 1: cut and paste the default value from f's interface into g's
function g( $x = 4 ) { f( $x ); f( $x ); }

// option 2: branch based on input to g
function g( $x = null ) { if ( !isset( $x ) ) { f(); f() } else { f( $x ); f( $x ); } }
?>
```

Both options suck.

The best approach, it seems to me, is to always use a sentinel like null as the default value of an optional argument. This way, callers like g and g's clients have many options, and furthermore, callers always know how to omit arguments so they can omit one in the middle of the parameter list.

```
<?php
function f( $x = null ) { if ( !isset( $x ) ) $x = 4; echo $x . "\\n"; }

function g( $x = null ) { f( $x ); f( $x ); }

f(); // prints 4
f( null ); // prints 4
f( $y ); // $y undefined, prints 4
g(); // prints 4 twice
g( null ); // prints 4 twice
g( 5 ); // prints 5 twice

?>
```

up
down
18

Sergio Santana: ssantana at tlaloc dot imta dot mx ¶
14 years ago

```
PASSING A "VARIABLE-LENGTH ARGUMENT LIST OF REFERENCES" TO A FUNCTION
As of PHP 5, Call-time pass-by-reference has been deprecated, this represents no problem in most cases, since instead of calling a function like this:
   myfunction($arg1, &$arg2, &$arg3);

you can call it
   myfunction($arg1, $arg2, $arg3);

provided you have defined your function as
   function myfuncion($a1, &$a2, &$a3) { // so &$a2 and &$a3 are
                                                                  // declared to be refs.
    ... <function-code>
   }

However, what happens if you wanted to pass an undefined number of references, i.e., something like:
   myfunction(&$arg1, &$arg2, ..., &$arg-n);?
This doesn't work in PHP 5 anymore.

In the following code I tried to amend this by using the
array() language-construct as the actual argument in the
call to the function.

<?php

  function aa ($A) {
    // This function increments each
    // "pseudo-argument" by 2s
    foreach ($A as &$x) {
      $x += 2;
    }
  }

  $x = 1; $y = 2; $z = 3;

  aa(array(&$x, &$y, &$z));
  echo "--$x--$y--$z--\n";
  // This will output:
  // --3--4--5--
?>

I hope this is useful.

Sergio.
```

up
down
3

dmitry dot balabka at gmail dot com ¶
1 year ago

```
There is a possibility to use parent keyword as type hint which is not mentioned in the documentation.

Following code snippet will be executed w/o errors on PHP version 7. In this example, parent keyword is referencing on ParentClass instead of ClassTrait.
<?php
namespace TestTypeHints;

class ParentClass
{
    public function someMethod()
    {
        echo 'Some method called' . \PHP_EOL;
    }
}

trait ClassTrait
{
    private $original;

    public function __construct(parent $original)
    {
        $this->original = $original;
    }

    protected function getOriginal(): parent
    {
        return $this->original;
    }
}

class Implementation extends ParentClass
{
    use ClassTrait;
```

```
    public function callSomeMethod()
    {
        $this->getOriginal()->someMethod();
    }
}

$obj = new Implementation(new ParentClass());
$obj->callSomeMethod();
?>

Outputs:
Some method called
```

up
down
20
carlos at wfmh dot org dot pl dot REMOVE dot COM ¶
9 years ago

You can use (very) limited signatures for your functions, specifing type of arguments allowed.

For example:

```
public function Right( My_Class $a, array $b )
```

tells first argument have to by object of My_Class, second an array. My_Class means that you can pass also object of class that either extends My_Class or implements (if My_Class is abstract class) My_Class. If you need exactly My_Class you need to either make it final, or add some code to check what $a really.

Also note, that (unfortunately) "array" is the only built-in type you can use in signature. Any other types i.e.:

```
public function Wrong( string $a, boolean $b )
```

will cause an error, because PHP will complain that $a is not an *object* of class string (and $b is not an object of class boolean).

So if you need to know if $a is a string or $b bool, you need to write some code in your function body and i.e. throw exception if you detect type mismatch (or you can try to cast if it's doable).

up
down
4
ohcc at 163 dot com ¶
4 years ago

As of PHP 5.6, you can use an array as arguments when calling a function with the ... $args syntax.

```
<?php
    $args = array('wu','WU','wuxiancheng.cn');
    $string = str_replace(...$args);
    echo $string;
?>
```

Ha ha, is that interesting and powerful?

Also you can use it like this

```
<?php
    $args = array('WU','wuxiancheng.cn');
    $string = str_replace('wu', ...$args);
    echo $string;
?>
```

It also can be used to define a user function.

```
<?php
    function wxc ($arg1, $arg2, ...$otherArgs){
        echo '<pre>';
        print_r($otherArgs);
        print_r(func_get_args());
        echo '</pre>';
    }
    wxc (1, 2, ...array(3,4,5));
?>
```

REMEMBER this: ... $args is not supported in PHP 5.5 and older versions.

up
down
6
info at keraweb dot nl ¶
2 years ago

You can use a class constant as a default parameter.

```
<?php
```

```
class A {
    const FOO = 'default';
    function bar( $val = self::FOO ) {
        echo $val;
    }
}

$a = new A();
$a->bar(); // Will echo "default"
```

up
down
8
Horst Schirmeier ¶
6 years ago
Editor's note: what is expected here by the parser is a non-evaluated expression. An operand and two constants requires evaluation, which is not done by the parser. However, this feature is included as of PHP 5.6.0. See this page for more information: http://php.net/migration56.new-features#migration56.new-features.const-scalar-exprs
---

"The default value must be a constant expression" is misleading (or even wrong). PHP 5.4.4 fails to parse this function definition:

```
function htmlspecialchars_latin1($s, $flags = ENT_COMPAT | ENT_HTML401) {}
```

This yields a " PHP Parse error: syntax error, unexpected '|', expecting ')' " although ENT_COMPAT|ENT_HTML401 is certainly what a compiler-affine person would call a "constant expression".

The obvious workaround is to use a single special value ($flags = NULL) as the default, and to set it to the desired value in the function's body (if ($flags === NULL) { $flags = ENT_COMPAT | ENT_HTML401; }).

up
down
10
mracky at pacbell dot net ¶
9 years ago
Nothing was written here about argument types as part of the function definition.

When working with classes, the class name can be used as argument type. This acts as a reminder to the user of the class, as well as a prototype for php control. (At least in php 5 -- did not check 4).

```
<?php
class foo {
  public $data;
  public function __construct($dd)
  {
    $this->data = $dd;
  }
};

class test {
  public $bar;

  public function __construct(foo $arg) // Strict typing for argument
  {
    $this->bar = $arg;
  }
  public function dump()
  {
    echo $this->bar->data . "\n";
  }

};

$A = new foo(25);
$Test1 = new test($A);
$Test1->dump();
$Test2 = new test(10); // wrong argument for testing

?>
```

outputs:

```
25
PHP Fatal error:  Argument 1 passed to test::__construct() must be an object of class foo, called in testArgType.php on line 27 and defined in testArgType.php on line 13
```

up
down
4
catman at esteticas dot se ¶
4 years ago
I wondered if variable length argument lists and references works together, and what the syntax might be. It is not mentioned explicitly yet in the php manual as far as I can find. But other sources mention the following syntax "&...$variable" that works in php 5.6.16.

```
<?php
function foo(&...$args)
```

```
{
    $i = 0;
    foreach ($args as &$arg) {
        $arg = ++$i;
    }
}
foo($a, $b, $c);
echo 'a = ', $a, ', b = ', $b, ', c = ', $c;
?>
Gives
a = 1, b = 2, c = 3
```

up
down
11
herenvardoREMOVEatSTUFFgmailINdotCAPScom ¶
11 years ago

There is a nice trick to emulate variables/function calls/etc as default values:

```
<?php
$myVar = "Using a variable as a default value!";

function myFunction($myArgument=null) {
    if($myArgument===null)
        $myArgument = $GLOBALS["myVar"];
    echo $myArgument;
}

// Outputs "Hello World!":
myFunction("Hello World!");
// Outputs "Using a variable as a default value!":
myFunction();
// Outputs the same again:
myFunction(null);
// Outputs "Changing the variable affects the function!":
$myVar = "Changing the variable affects the function!";
myFunction();
?>
```

In general, you define the default value as null (or whatever constant you like), and then check for that value at the start of the function, computing the actual default if needed, before using the argument for actual work.
Building upon this, it's also easy to provide fallback behaviors when the argument given is not valid: simply put a default that is known to be invalid in the prototype, and then check for general validity instead of a specific value: if the argument is not valid (either not given, so the default is used, or an invalid value was given), the function computes a (valid) default to use.

up
down
4
boan dot web at outlook dot com ¶
2 years ago

Quote:

"The declaration can be made to accept NULL values if the default value of the parameter is set to NULL."

But you can do this (PHP 7.1+):

```
<?php
function foo(?string $bar) {
    //...
}

foo(); // Fatal error
foo(null); // Okay
foo('Hello world'); // Okay
?>
```

up
down
3
igorsantos07 at gmail dot com ¶
2 years ago

PHP 7+ does type coercion if strict typing is not enabled, but there's a small gotcha: it won't convert null values into anything.

You must explicitly set your default argument value to be null (as explained in this page) so your function can also receive nulls.

For instance, if you type an argument as "string", but pass a null variable into it, you might expect to receive an empty string, but actually, PHP will yell at you a TypeError.

```
<?php
function null_string_wrong(string $str) {
  var_dump($str);
}
function null_string_correct(string $str = null) {
  var_dump($str);
}
```

```
$null = null;
null_string_wrong('a');     //string(1) "a"
null_string_correct('a');   //string(1) "a"
null_string_correct();      //NULL
null_string_correct($null); //NULL
null_string_wrong($null);   //TypeError thrown
?>
```

up
down
7

conciseusa at yahoo[nospammmm] dot com ¶
12 years ago

With regards to:

It is also possible to force a parameter type using this syntax. I couldn't see it in the documentation.
function foo(myclass par) { }

I think you are referring to Type Hinting. It is documented here: http://ch2.php.net/language.oop5.typehinting

up
down
2

Hayley Watson ¶
2 years ago

If you use ... in a function's parameter list, you can use it only once for obvious reasons. Less obvious is that it has to be on the LAST parameter; as the manual puts it: "You may specify normal positional arguments BEFORE the ... token. (emphasis mine).

```
<?php
function variadic($first, ...$most, $last)
{/*etc.*/}

variadic(1, 2, 3, 4, 5);
?>
```

results in a fatal error, even though it looks like the Thing To Do™ would be to set $first to 1, $most to [2, 3, 4], and $last to 5.

up
down
2

iqravi dot bhushan at gmail dot com ¶
1 year ago

You can pass a function as an argument too.

```
<?php

    function sum($numbers){
        $acc = 0;
        foreach ($numbers as $key => $value) {
            $acc += $value;
        }
        return $acc;
    }

    function generateString(){
        $x = array(1,2,3,4,5,6,7);
        return $x;
    }

    echo sum(generateString());
?>
```

up
down
2

g dot sokol99 at g-sokol dot info ¶
3 years ago

Nullable arguments:

```
<?php

class C {}

function foo(?C $a)
{
    var_dump($a);
}

foo(null);

foo();
?>
```

Output:

```
NULL
```

```
PHP Warning:  Uncaught ArgumentCountError: Too few arguments to function foo(), 0 passed in php shell code on line 1 and exactly 1 expected in php shell
code:1
Stack trace:
#0 php shell code(1): foo()
#1 {main}
  thrown in php shell code on line 1
```

Usage of "?" Is also possible with "string", "int", "array" and so on primitive types (which is strange). Also unliKe "= null" "?" can be passed not only for tail of arguments, e.g.:

```
<?php
function foo(?string $a, string $b) {}
?>
```

up
down
4
John ¶
13 years ago

This might be documented somewhere OR obvious to most, but when passing an argument by reference (as of PHP 5.04) you can assign a value to an argument variable in the function call. For example:

```
function my_function($arg1, &$arg2) {
  if ($arg1 == true) {
    $arg2 = true;
  }
}
my_function(true, $arg2 = false);
echo $arg2;

outputs 1 (true)

my_function(false, $arg2 = false);
echo $arg2;

outputs 0 (false)
```

up
down
2
ksamvel at gmail dot com ¶
14 years ago

by default Classes constructor does not have any arguments. Using small trick with func_get_args() and other relative functions constructor becomes a function w/ args (tested in php 5.1.2). Check it out:

```
class A {
    public function __construct() {
        echo func_num_args() . "<br>";
        var_dump( func_get_args());
        echo "<br>";
    }
}

$oA = new A();
$oA = new A( 1, 2, 3, "txt");

Output:

0
array(0) { }
4
array(4) { [0]=> int(1) [1]=> int(2) [2]=> int(3) [3]=> string(3) "txt" }
```

up
down
2
allankelly at gmail dot com ¶
10 years ago

I like to pass an associative array as an argument. This is reminiscent of a Perl technique and can be tested with is_array. For example:

```
<?php
function div( $opt )
{
    $class = '';
    $text  = '';
    if( is_array( $opt ) )
    {
        foreach( $opt as $k => $v )
        {
            switch( $k )
            {
                case 'class': $class = "class = '$v'";
                              break;
                case 'text': $text = $v;
                              break;
            }
```

```
        }
    }
    else
    {
        $text = $opt;
    }
    return "<div $class>$text</div>";
}
?>
```

up
down
2
info at ensostudio dot ru ¶
4 months ago

If class have __toString method then his methods may set return type 'string' and return self:

```
<?php
class Str
{
    public function getAsString(): string
    {
        return $this;
    }

    public function getAsObject(): self
    {
        return $this;
    }

    public function __toString()
    {
        return get_class($this);
    }
}
?>
```

up
down
0
shaman_master at list dot ru ¶
3 months ago

You can use the class/interface as a type even if the class/interface is not defined yet or the class/interface implements current class/interface.

```
<?php
interface RouteInterface
{
    public function getGroup(): ?RouteGroupInterface;
}
interface RouteGroupInterface extends RouteInterface
{
    public function set(RouteInterface $item);
}
?>
```

'self' type - alias to current class/interface, it's not changed in implementations. This code looks right but throw error:

```
<?php
class Route
{
    protected $name;
     // method must return Route object
    public function setName(string $name): self
    {
         $this->name = $name;
         return $this;
    }
}
class RouteGroup extends Route
{
    // method STILL must return only Route object
    public function setName(string $name): self
    {
         $name .= ' group';
         return parent::setName($name);
    }
}
?>
```

up
down
0
Anonymous ¶
1 year ago

Notice that only order matters. Example:

```
function f($x='a', $y='b'){
   $output = $x.$y;
```

```
   echo $output;
}
f($y='c',$x='d');

This will print 'cd'.

If we call function like this
f($y='c')

actually $x in function gets value of 'c' and $y takes default value, as it just fills parameters in order specified so output will be 'cb'. Names have no meaning in function call. This may be confusing if you are coming from python for example.
```

up
down
1
thesibster at hotmail dot com ¶
16 years ago

```
Call-time pass-by-ref arguments are deprecated and may not be supported later, so doing this:

----
function foo($str) {
    $str = "bar";
}

$mystr = "hello world";
foo(&$mystr);
----

will produce a warning when using the recommended php.ini file.  The way I ended up using for optional pass-by-ref args is to just pass an unused variable when you don't want to use the resulting parameter value:

----
function foo(&$str) {
    $str = "bar";
}

foo($_unused_);
----

Note that trying to pass a value of NULL will produce an error.
```

up
down
0
rburnap at intelligent dash imaging dot com ¶
9 years ago

```
This may be helpful when you need to call an arbitrary function known only at runtime:

You can call a function as a variable name.
<?php

function foo(){
    echo"\nfoo()";
}

function callfunc($x, $y = '')
{
    if( $y=='' )
    {
        if( $x=='' )
             echo "\nempty";
        else $x();
    }
    else
         $y->$x();
}

class cbar {
    public function fcatch(){ echo "\nfcatch"; }
}

$x = '';
callfunc($x);
$x = 'foo';
callfunc($x);
$o = new cbar();
$x = 'fcatch';
callfunc($x, $o);
echo "\n\n";
?>

The code will output
```

```
empty
foo()
fcatch
```

up
down
-1
pigiman at gmail dot com ¶
9 years ago

```
Hey,

I started to learn for the Zend Certificate exam a few days ago and I got stuck with one unanswered-well question.
This is the question:
“Absent any actual need for choosing one method over the other, does passing arrays by value to a read-only function reduce performance compared to
passing them by reference?’

This question answered by Zend support team at Zend.com:

"A copy of the original $array is created within the function scope. Once the function terminates, the scope is removed and the copy of $array with it."
(By massimilianoc)

Have a nice day!

Shaked KO
```

up
down
-1
guillaume dot goutaudier at eurecom dot fr ¶
17 years ago

```
Concerning default values for arguments passed by reference:
I often use that trick:
func($ref=$defaultValue) {
    $ref = "new value";
}
func(&$var);
print($var) // echo "new value"

Setting $defaultValue to null enables you to write functions with optional arguments which, if given, are to be modified.
```

up
down
-1
wls at wwco dot com ¶
18 years ago

```
Follow up to resource passing:

It appears that if you have defined the resource in the same file
as the function that uses it, you can get away with the global trick.

Here's the failure case:

  include "functions_doing_globals.php"
  $conn = openDatabaseConnection();
  invoke_function_doing_global_conn();

...that it fails.

Perhaps it's some strange scoping problem with include/require, or
globals trying to resolve before the variable is defined, rather
than at function execution.
```

up
down
-1
nate at natemurray dot com ¶
14 years ago

```
Of course you can fake a global variable for a default argument by something like this:
<?php
function self_url($text, $page, $per_page = NULL) {
  $per_page = ($per_page == NULL) ? $GLOBALS['gPER_PAGE'] : $per_page; # setup a default value of per page
  return sprintf("<a href=%s?page=%s&perpage=%s>%s</a>", $_SERVER["PHP_SELF"], $page, $per_page, $text);
}
?>
```

add a note