CHAPTER 3
Functions

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

This chapter introduces the syntax of function calls and function definitions and discusses how to manage variables in functions and pass values to functions (including pass-by-value and pass-by-reference). It also covers variable functions and anonymous functions.

Calling a Function

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

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

The number of parameters a function requires differs from function to function (and, as we'll see later, may even vary for the same function). The parameters supplied to the function may be any valid expression and must be in the specific order expected by the function. If the parameters are given out of order, the function may still run by a fluke, but it's basically a case of garbage in = garbage out. A function's documentation will tell you what parameters the function expects and what values you can expect to be returned.

Here are some examples of functions:

```
// strlen() is a built-in function that returns the length of a string
$length = strlen("PHP"); // $length is now 3
```

```
// sin() and asin() are the sine and arcsine math functions
$result = sin(asin(1)); // $result is the sine of arcsin(1), or 1.0

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

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

The second example passes the result of `asin(1)` to the `sin()` function. Since the sine and arcsine functions are inverses, taking the sine of the arcsine of any value will always return that same value. Here we see that a function can be called within another function. The returned value of the inner call is subsequently sent to the outer function before the overall result is returned and stored in the `$result` variable.

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

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

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

PHP has a huge array of functions already defined for you to use in your programs. Everything from database access to creating graphics to reading and writing XML files to grabbing files from remote systems can be found in PHP's many extensions. PHP's built-in functions are described in detail in the Appendix.

Defining a Function

To define a function, use the following syntax:

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

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

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

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

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

Let's take a look at a simple function. Example 3-1 takes two strings, concatenates them, and then returns the result (in this case, we've created a slightly slower equivalent to the concatenation operator, but bear with us for the sake of example).

Example 3-1. String concatenation

```
function strcat($left, $right)
{
  $combinedString = $left . $right;

  return $combinedString;
}
```

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

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

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

If we put this function on a PHP page, we can call it from anywhere within the page. Take a look at Example 3-2.

Example 3-2. Using our concatenation function

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

```
$first = "This is a ";
$second = " complete sentence!";

echo strcat($first, $second);
```

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

In this example the function takes in an integer, doubles it via bit shifting the original value, and returns the result:

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

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

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

You can nest function declarations, but with limited effect. Nested declarations do not limit the visibility of the inner-defined function, which may be called from anywhere in your program. The inner function does not automatically get the outer function's arguments. And, finally, the inner function cannot be called until the outer function has been called, and also cannot be called from code parsed after the outer function:

```
function outer ($a)
{
  function inner ($b)
  {
    echo "there $b";
  }

  echo "$a, hello ";
}

// outputs "well, hello there reader"
outer("well");
inner("reader");
```

Variable Scope

If you don't use functions, any variable you create can be used anywhere in a page. With functions, this is not always true. Functions keep their own sets of variables that are distinct from those of the page and of other functions.

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

```
$a = 3;

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

foo();
echo $a;
```

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

As we discussed in Chapter 2, the extent to which a variable can be seen in a program is called the *scope* of the variable. Variables created within a function are inside the scope of the function (i.e., have *function-level scope*). Variables created outside of functions and objects have *global scope* and exist anywhere outside of those functions and objects. A few variables provided by PHP have both function-level and global scope (often referred to as *super-global variables*).

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

Global Variables

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

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

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

```
$a = 3;

function foo()
{
  global $a;

  $a += 2;
}

foo();
echo $a;
```

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

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

Using `global` is equivalent to creating a reference to the variable in the `$GLOBALS` variable. That is, both of the following declarations create a variable in the function's scope that is a reference to the same value as the variable `$var` in the global scope:

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

Static Variables

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

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

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

Example 3-3. Static variable counter

```
<?php
function counter()
{
  static $count = 0;

  return $count++;
}

for ($i = 1; $i <= 5; $i++) {
  print counter();
}
```

When the function is called for the first time, the static variable `$count` is assigned a value of `0`. The value is returned and `$count` is incremented. When the function ends, `$count` is not destroyed like a nonstatic variable, and its value remains the same until the next time `counter()` is called. The `for` loop displays the numbers from 0 to 4.

Function Parameters

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

Passing Parameters by Value

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

Passing Parameters by Reference

Passing by reference allows you to override the normal scoping rules and give a function direct access to a variable. To be passed by reference, the argument must be a variable; you indicate that a particular argument of a function will be passed by reference by preceding the variable name in the parameter list with an ampersand (`&`). Example 3-4 revisits our `doubler()` function with a slight change.

Example 3-4. Doubler redux

```
<?php
function doubler(&$value)
{
  $value = $value << 1;
}

$a = 3;
doubler($a);

echo $a;
```

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

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

Only variables—and not constants—can be supplied to parameters declared as passing by reference. Thus, if we included the statement `<?= doubler(7); ?>` in the previous example, it would issue an error. However, you may assign a default value to parameters passed by reference (in the same manner as you provide default values for parameters passed by value).

Even in cases where your function does not affect the given value, you may want a parameter to be passed by reference. When passing by value, PHP must copy the value. Particularly for large strings and objects, this can be an expensive operation. Passing by reference removes the need to copy the value.

Default Parameters

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

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

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

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

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

Variable Parameters

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

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

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

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

In Example 3-5, the `count_list()` function takes in any number of arguments. It loops over those arguments and returns the total of all the values. If no parameters are given, it returns `false`.

Example 3-5. Argument counter

```
<?php
function countList()
{
  if (func_num_args() == 0) {
    return false;
  }
  else {
    $count = 0;

    for ($i = 0; $i < func_num_args(); $i++) {
      $count += func_get_arg($i);
    }

    return $count;
  }
}

echo countList(1, 5, 9); // outputs "15"
```

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

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

Instead, use:

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

Missing Parameters

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

```
    function takesTwo($a, $b)
    {
      if (isset($a)) {
        echo " a is set\n";
      }

      if (isset($b)) {
        echo " b is set\n";
      }
    }

    echo "With two arguments:\n";
    takesTwo(1, 2);

    echo "With one argument:\n";
    takesTwo(1);
```

```
With two arguments:
 a is set
 b is set
With one argument:
Warning:  Missing argument 2 for takes_two()
 in /path/to/script.php on line 6
a is set
```

Type Hinting

When defining a function, you can require that a parameter be an instance of a particular class (including instances of classes that extend that class), an instance of a class that implements a particular interface, an array, or a callable. To add type hinting to a parameter, include the class name, `array`, or `callable` before the variable name in the function's parameter list. For example:

```
class Entertainment {}

class Clown extends Entertainment {}

class Job {}

function handleEntertainment(Entertainment $a, callable $callback = NULL)
{
  echo "Handling " . get_class($a) . " fun\n";

  if ($callback !== NULL) {
    $callback();
  }
}

$callback = function()
{
  // do something
};

handleEntertainment(new Clown); // works
handleEntertainment(new Job, $callback); // runtime error
```

A type-hinted parameter must either be `NULL`, or an instance of the given class or a subclass of class, an array, or a callable as specified parameter. Otherwise, a runtime error occurs.

Type hinting cannot be used to require a parameter be of a particular scalar type (such as integer or string) or to have a particular trait.

Return Values

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

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

To return multiple values, return an array:

```
function returnTwo()
{
    return array("Fred", 35);
}
```

If no return value is provided by a function, the function returns `NULL` instead.

By default, values are copied out of the function. To return a value by reference, both declare the function with an `&` before its name and when assigning the returned value to a variable:

```
$names = array("Fred", "Barney", "Wilma", "Betty");

function &findOne($n) {
    global $names;

    return $names[$n];
}

$person =& findOne(1);           // Barney
$person = "Barnetta";            // changes $names[1]
```

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

This technique is sometimes used to return large string or array values efficiently from a function. However, PHP implements copy-on-write for variable values, meaning that returning a reference from a function is typically unnecessary. Returning a reference to a value is slower than returning the value itself.

Variable Functions

As with variable variables where the expression refers to the value of the variable whose name is the value held by the apparent variable (the `$$` construct), you can add parentheses after a variable to call the function whose name is the value held by the apparent variable, e.g., `$variable()`. Consider this situation, where a variable is used to determine which of three functions to call:

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

  case 'second':
    second();
    break;
  case 'third':
    third();
    break;
}
```

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

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

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

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

For example:

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

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

```
    $which = "echo";
    $which("hello, world"); // does not work
```

Anonymous Functions

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

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

You can create an anonymous function using the normal function definition syntax, but assign it to a variable or pass it directly.

Example 3-6 shows an example using `usort()`.

Example 3-6. Anonymous functions

```
$array = array("really long string here, boy", "this", "middling length", "larger");

usort($array, function($a, $b) {
  return strlen($a) - strlen($b);
});

print_r($array);
```

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

Anonymous functions can use the variables defined in their enclosing scope using the `use` syntax. For example:

```
$array = array("really long string here, boy", "this", "middling length", "larger");
$sortOption = 'random';

usort($array, function($a, $b) use ($sortOption)
{
  if ($sortOption == 'random') {
    // sort randomly by returning (-1, 0, 1) at random
    return rand(0, 2) - 1;
  }
  else {
    return strlen($a) - strlen($b);
  }
});

print_r($array);
```

Note that incorporating variables from the enclosing scope is not the same as using global variables—global variables are always in the global scope, while incorporating variables allows a closure to use the variables defined in the enclosing scope. Also note that this is not necessarily the same as the scope in which the closure is called. For example:

```
$array = array("really long string here, boy", "this", "middling length", "larger");
$sortOption = "random";

function sortNonrandom($array)
{
  $sortOption = false;

  usort($array, function($a, $b) use ($sortOption)
  {
    if ($sortOption == "random") {
      // sort randomly by returning (-1, 0, 1) at random
      return rand(0, 2) - 1;
    }
    else {
      return strlen($a) - strlen($b);
    }
  });

  print_r($array);
}

print_r(sortNonrandom($array));
```

In this example, `$array` is sorted normally, rather than randomly—the value of `$sortOption` inside the closure is the value of `$sortOption` in the scope of `sortNonrandom()`, not the value of `$sortOption` in the global scope.