Apress™
Books for Professionals by Professionals

Chapter Four: "Functions"

A Programmer's Introduction to PHP 4.0

by William Jason Gilmore
ISBN # 1-893115-85-2

Gilmore_04 12/4/00 1:04 PM Page 81

CHAPTER 4 Functions

This chapter introduces the general concepts of functional programming, one of the most influential advances in application development. Functions enable you to develop reusable and easily modifiable components, which are particularly useful when you need to develop Web applications similar in concept and utility. Functional programming results in shorter, easier to read programs.

In particular, this chapter is concerned with the creation, implementation, and manipulation of PHP functions. Although the general focus is on defining and executing user-defined functions, it is also important to know that PHP offers hundreds of predefined functions. Predefined functions are used exactly as user-defined functions are and save considerable time for developing new applications. For the most up-to-date listing of these functions, check out http://www.php.net.

What Is a Function?

A *function* is a section of code with a specific purpose that is assigned a unique name. The function name can be called at various points in a program, allowing the section of code represented by this name to be repeatedly executed as needed. This is convenient because the same section of code is written only once, but can be easily modified as necessary.

Function Definition and Invocation

Creating a PHP function is a rather straightforward process. You can create a function at any point in a PHP program. However, for organizational purposes you may find it convenient to place all functions intended for use in a script at the very top of the script file. An alternative method for function organization that can greatly reduce redundancy and promote code reuse is the placement of the functions in a separate file (also known as a *library*). This is convenient because you can use the functions repeatedly in various applications without having to make redundant copies and thus risk errors due to rewriting. I explain this process in detail toward the conclusion of this chapter, in “Building Function Libraries.”

A function definition generally consists of three distinct parts:

- The name of the function
- Parentheses enclosing an optional set of comma-delimited input parameters
- The body of the function, enclosed in curly brackets

The general form of a PHP function is as follows:

```
function function_name (optional $arg1, $arg2, ..., $argn) {
       code section
}
```

The function name must follow the lexical structure conditions as specified in Chapter 2, "Variables and Data Types." The function name is then followed by a mandatory set of parentheses, enclosing an optional set of input parameters (`$arg1, $arg2`, ..., `$argn`). Due to PHP's relatively relaxed perspective on variable definitions, there is no need to specify the data type of the input parameters. While this has its advantages, realize that the PHP engine does not verify that the data passed into the function is intended to be handled by the function. This could result in unexpected results if the input parameter is used in an unintended fashion. (To ensure that the input parameter is being used as intended, you can test it using the predefined gettype() function.) A set of curly brackets ({}) follows the closing parentheses, enclosing the section of code to be associated with the function name.

Let's consider a simple example of practical usage of a function. Suppose you wanted to create a function that outputs a general copyright notice to a Web page:

```
function display_copyright() {
print "Copyright &copy; 2000 PHP-Powered Recipes. All Rights Reserved.";
}
```

Assuming that your Web site contains many pages, you could simply call this function at the bottom of each page, eliminating the need to continually rewrite the same information. Conveniently, the arrival of the year 2001 will bring about one simple modification of the text contained in the function that will result in an updated copyright statement. If functional programming weren't possible, you would have to modify every page in which the copyright statement was included!

Consider a variation of the display_copyright() function in which we pass a parameter. Suppose that you were in charge of the administration of several Web sites, each with a different name. Further imagine that each site had its own administration script, consisting of various variables relative to the specific site, one of the variables being $site_name. With this in mind, the function display_copyright() could be rewritten as follows:

```
function display_copyright($site_name) {
print "Copyright &copy 2000 $site_name. All Rights Reserved.";
}
```

The variable `$site_name`, assigned a value from somewhere outside of the function, is passed into `display_copyright()` as an input parameter. It can then be used and modified anywhere in the function. However, modifications to the variable will not be recognized anywhere outside of the function, although it is possible to force this recognition through the use of special keywords. These keywords, along with a general overview of variable scoping as it relates to functions, were introduced in Chapter 2, "Variables and Data Types."

Nested Functions

It is also possible to nest functions within functions, much as you can insert one control structure (if, while, for, and so on) within another. This is useful for programs large and small, as it adds another level of modularization to the application, resulting in increasingly manageable code.

Revisiting the copyright example described earlier, you can eliminate the need to modify the date altogether by nesting PHP's predefined `date()` function in the `display_copyright()` function:

```
function display_copyright($site_name) {
print "Copyright &copy". date("Y"). " $site_name. All Rights Reserved.";
}
```

The Y input parameter of the `date()` function specifies that the return value should be the current year, formatted using four digits. Assuming that the system date configuration is correct, PHP will output the correct year on each invocation of the script. PHP's `date()` function is extremely flexible, offering 25 different date- and time-formatting flags.

You can also nest function declarations inside one another. However, nesting a function declaration does not imply that it is protected in the sense of it being limited to use only in the function in which it is declared. Furthermore, a nested function does not inherit the input parameters of its parent; they must be passed to the nested function just as they are passed to any other function. Regardless, you may find it useful to do nest function declarations for reasons of code management and clarity. Listing 4-1 gives an example of nesting function declarations.

Listing 4-1: Making efficient use of nested functions

```
function display_footer($site_name) {

         function display_copyright($site_name) {
              print "Copyright &copy ". date("Y"). " $site_name. All Rights
Reserved.";
         }

          print "<center>
          <a href = \"\">home</a> | <a href = \"\">recipes</a> | <a href =
\"\">events</a><br>
          <a href = \"\">tutorials</a> | <a href = \"\">about</a> | <a href =
\"\">contact us</a><br>";

         display_copyright($site_name);

         print "</center>";

}

$site_name = "PHP Recipes";

display_footer($site_name);
```

Executing this script produces the following output:

```
                    home | recipes | events
                 tutorials | about | contact us
          Copyright © 2000 PHP Recipes. All Rights Reserved.
```

NOTE *It is important to note that we could also call* `display_copyright()` *from outside the* `display_footer()` *function, just as* `display_footer()` *was called in the preceding example. PHP does not support the concept of protected functions.*

Although nested functions are not protected from being called from any other location of the script, they cannot be called until *after* their parent function has been called. An attempt to call a nested function before calling its parent function results in an error message.

Returning Values from a Function

It is often useful to return a value from a function. This is accomplished by assigning the function call to a variable. Any type may be returned from a function, including lists and arrays. Consider Listing 4-2, in which the sales tax for a given price is calculated and the total cost subsequently returned. Before checking out the code, take a minute to review the pseudocode summary:

- Assume that a few values have been set, in this case some product price, $price, and sales tax, $tax.
- Declare function calculate_cost(). It accepts two parameters, the sales tax and the product price.
- Calculate the total cost and use return to send the calculated cost back to the caller.
- Call calculate_cost(), setting $total_cost to whatever value is returned from the function.
- Output a relevant message.

Listing 4-2: Building a function that calculates sales tax

```
$price = 24.99;
$tax = .06;

function calculate_cost($tax, $price) {
     $sales_tax = $tax;
     return $price + ($price * $sales_tax);
}

// Notice how calculate_cost() returns a value.
$total_cost = calculate_cost ($tax, $price);
// round the cost to two decimal places.
$total_cost = round($total_cost, 2);

print "Total cost: ".$total_cost;
// $total_cost = 26.49
```

NOTE *A function that does not return a value is also known as a procedure.*

Another way in which to use returned values is to incorporate the function call directly into a conditional/iterative statement. Listing 4-3 checks a user's total bill against a credit limit. The pseudocode is found here:

- Declare function `check_limit()`, which takes as input two parameters. The first parameter, `$total_cost`, is the total bill accumulated by the user thus far. The second, `$credit_limit`, is the maximum cash amount the user is allowed to charge.

- If the total accumulated bill is greater than the credit limit, return a false (0) value.

- If the if statement evaluates to false, then the function has not yet terminated. Therefore, the total cost has not exceeded the credit limit, and true should be returned.

- Use the function check_limit() in an if conditional statement. Check_limit() will return either a true or a false value. This returned value will determine the action that the if statement takes.

If `check_limit()` evaluates to true, tell the user to keep shopping. Otherwise, inform that user that the credit limit has been exceeded.

Listing 4-3: Comparing a user's current bill against a credit limit

```
$cost = 1456.22;
$limit = 1000.00;

function check_limit($total_cost, $credit_limit) {

     if ($total_cost > $credit_limit) :
          return 0;
     endif;

     return 1;

}

if (check_limit($cost, $limit)) :
// let the user keep shopping
print "Keep shopping!";
else :
print "Please lower your total bill to less than $".$limit."!";
endif;
```

Execution of Listing 4-3 results in the error message being displayed, since `$cost` has exceeded `$limit`.

It is also possible to simultaneously return multiple values from a function by using a list. Continuing with the culinary theme, consider a function that returns the three recommended years of a particular wine. This function is illustrated in Listing 4-4. Read through the pseudocode first:

- Declare function `best_years()`, which takes as input one parameter. The parameter `$label` is the type of wine in which the user would like to view the three recommended years.

- Declare two arrays, `$merlot`, and `$zinfandel`. Each array holds the three recommended years for that wine type.

- Implement the return statement to make wise use of the variable functionality. The statement `$$label` will first interpret the variable `$label` and then interpret whatever the value of `$label` is as another variable. In this case, the merlot array will be returned as a list, each year taking its respective position in the calling list.

- Print out a relevant message, informing the user of these recommended years.

Listing 4-4: Returning multiple values from a function

```
// wine for which best years will be displayed
$label = "merlot";

// This function merely makes use of various arrays and a variable variable to
return multiple values.
function best_years($label) {

     $merlot = array(1987, 1983, 1977);
     $zinfandel = array(1992, 1990, 1989);

     return $$label;

}
// a list() Is used to display the wine's best years.
list ($yr_one, $yr_two, $yr_three) = best_years($label);

print "$label had three particularly remarkable years: $yr_one, $yr_two, and
$yr_three.";
```

Execution of Listing 4-3 results in the following output:

```
merlot had three particularly remarkable years: 1987, 1983, and 1977.
```

Recursive Functions

The act of a function calling on itself again and again to satisfy some operation is indeed a powerful one. Used properly, *recursive* function calls can save undue space and redundancy in a script and are especially useful for performing repetitive procedures. Examples of these repetitive applications include file/array searches and graphic renderings (fractals, for instance). An example commonly illustrated in computer science courses is the summation of integers 1 to N. Listing 4-5 recursively sums all integers between 1 and 10.

Listing 4-5: Using a recursive function to sum an integer set

```
function summation ($count) {
     if ($count != 0) :
          return $count + summation($count-1);
     endif;
}
$sum = summation(10);
print "Summation = $sum";
```

Execution of the Listing 4-5 produces the following results:

```
Summation = 55
```

Using functional iteration (recursion) can result in speed improvements in a program if the function is called often enough. However, be careful when writing recursive procedures, as improper coding can result in an infinite loop.

Variable Functions

An interesting capability of PHP is the possibility to execute *variable functions*. A variable function is a dynamic call to a function whose name is determined at the time of execution. Although not necessary in most Web applications, variable functions can significantly reduce code size and complexity, often eliminating unnecessary if conditional statements.

A call to a variable function is nothing more than a variable name followed by a set of parentheses. In the parentheses an optional set of input parameters can be included. The general form of a variable function is as follows:

```
$function_name();
```

Listing 4-6 illustrates this odd but useful feature. Suppose that users are given the possibility to view certain information in their choice of language. Our example will keep things simple, offering a welcome message tailored to English- and Italian-speaking users. Here is the pseudocode:

- An Italian interface is created in a function entitled "italian".
- An English interface is created in a function entitled "english".
- The choice of language is passed into the script, set in the variable `$language`.

The variable `$language` is used to execute a variable function, in this case `italian()`.

Listing 4-6: Using a variable function determined by some input variable

```
// italian welcome message.
function italian() {
     print "Benvenuti al PHP Recipes.";
}

// english welcome message
function english() {
     print "Welcome to PHP Recipes.";
}

// set the user language to italian
$language = "italian";

// execute the variable function
$language();
```

Listing 4-6 illustrates the interesting concept of a variable function and how it can be used to greatly limit code volume. Without the capability of using a variable function, you would be forced to use a switch or if statement to determine which function should be executed. This would take up considerably more space and introduce the possibility of errors due to added coding.

Building Function Libraries

Function libraries are one of the most efficient ways to save time when building applications. For example, you may have written a series of function for sorting arrays. You could probably use these functions repeatedly in various applications. Rather than continually rewrite or copy and paste these functions into new scripts, it is much more convenient to place all relevant sorting functions into a separate file altogether. This file would then be given an easily recognizable title, for example, array_sorting.inc, as shown in Listing 4-7.

Listing 4-7: A sample function library (array_sorting.inc)

```
<?
// file: array_sorting.inc
// purpose: library containing functions used for sorting arrays.
// date: July 17, 2000

function merge_sort($array, $tmparray, $right, $left) {
. . .
}

function bubble_sort($array, $n) {
. . .
}

function quick_sort($array, $right, $left) {
. . .
}

?>
```

This function library, array_sorting.inc, acts as a receptacle for all of my array-sorting functions. This is convenient because I can effectively organize my functions according to purpose, allowing for easy lookup when necessary. As you can see in Listing 4-7, I like to add a few lines of commented header at the top of each library so I have a quick synopsis of the library contents once I open the file. Once you have built your own custom function library, you can use PHP's include() and require() statements to include the entire library file to a script, thus making all of the functions available. The general syntax of both statements is as follows:

```
include(path/filename);
require(path/filename);
```

An alternate syntax is also available:

```
include "path/filename";
require "path/filename";
```

where "path" refers to either the relative or absolute path location of the filename. The include() and require() constructs are introduced in detail in Chapter 9, "PHP and the Web." For the moment, however, you should just understand that these constructs can be used to include a file directly into a script for use.

Suppose you wanted to use the library array_sorting.inc in a script. You could easily include the library, as shown in Listing 4-8.

Listing 4-8: Including a function library (array_sorting.inc) in a script

```
// this assumes that the array_sorting.inc library resides in the same folder as
this script.
include ("array_sorting.inc");

// you are now free to use any function in array_sorting.inc.
$some_array = (50, 42, 35, 46);

// make use of the bubble_sort() function
$sorted_array = bubble_sort($some_array, 1);
```

What's Next?

This chapter introduced functions and their range of uses as applied to PHP. In particular, the following topics were discussed:

- Function definition and invocation
- Nested functions
- Returning values from a function
- Returning multiple values
- Recursive functions
- Variable functions
- Building function libraries

Understanding this chapter will be integral to understanding the concepts discussed throughout the remaining chapters, as functions are used whenever possible. As is the case with every other chapter, I suggest experimenting with the examples in order to strengthen your comprehension of the provided material.

Chapter 5 introduces what will surely become a welcome addition to your PHP knowledge: arrays. Chapter 5 will provide you with your first real taste of data storage, paving the way to more content-oriented and ultimately interesting applications.