Beginning PHP and MySQL 5

From Novice to Professional, Second Edition

■ ■ ■

W. Jason Gilmore

Apress®

Beginning PHP and MySQL 5: From Novice to Professional, Second Edition

Copyright © 2006 by W. Jason Gilmore

ISBN-13 (pbk): 978-1-59059-552-7
ISBN-10 (pbk): 1-59059-552-1

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Matthew Moodie
Technical Reviewer: Matthew Wade
Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis, Jason Gilmore, Jonathan Hassell, Chris Mills, Dominic Shakeshaft, Jim Sumser
Project Managers: Laura Cheu, Beth Christmas
Copy Edit Manager: Nicole LeClerc
Copy Editor: Bill McManus
Assistant Production Director: Kari Brooks-Copony
Production Editor: Laura Cheu
Compositor: Susan Glinert Stevens
Proofreader: Nancy Sixsmith
Indexer: John Collin
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Source Code section.

CHAPTER 4

Functions

Even in trivial applications, repetitive processes are likely to exist. For nontrivial applications, such repetition is a given. For example, in an e-commerce application, you might need to query a customer's profile information numerous times: at login, at checkout, and when verifying a shipping address. However, repeating the profile querying process throughout the application would be not only error-prone, but also a nightmare to maintain. What happens if a new field has been added to the customer's profile? You might need to sift through each page of the application, modifying the query as necessary, likely introducing errors in the process.

Thankfully, the concept of embodying these repetitive processes within a named section of code, and then invoking this name as necessary, has long been a key component of any respectable computer language. These sections of code are known as *functions,* and they grant you the convenience of a singular point of modification if the embodied process requires changes in the future, which greatly reduces both the possibility of programming errors and maintenance overhead. In this chapter, you'll learn all about PHP functions, including how to create and invoke them, pass input, return both single and multiple values to the caller, and create and include function libraries. Additionally, you'll learn about both *recursive* and *variable* functions.

Invoking a Function

More than 1,000 standard functions are built into the standard PHP distribution, many of which you'll see throughout this book. You can invoke the function you want simply by specifying the function name, assuming that the function has been made available either through the library's compilation into the installed distribution or via the `include()` or `require()` statement. For example, suppose you want to raise 5 to the third power. You could invoke PHP's `pow()` function like this:

```
<?php
    $value = pow(5,3); // returns 125
    echo $value;
?>
```

If you simply want to output the function outcome, you can forego assigning the value to a variable, like this:

```
<?php
    echo pow(5,3);
?>
```

If you want to output function outcome within a larger string, you need to concatenate it like this:

```
echo "Five raised to the third power equals ".pow(5,3).".";
```

Creating a Function

Although PHP's vast assortment of function libraries is a tremendous benefit to any programmer who is seeking to avoid reinventing the programmatic wheel, sooner or later you'll need to go beyond what is offered in the standard distribution, which means you'll need to create custom functions or even entire function libraries. To do so, you'll need to define a function using a predefined syntactical pattern, like so:

```
function function_name (parameters) {
    function-body
}
```

For example, consider the following function, `generate_footer()`, which outputs a page footer:

```
function generate_footer() {
    echo "<p>Copyright &copy; 2006 W. Jason Gilmore</p>";
}
```

Once it is defined, you can then call this function as you would any other. For example:

```
<?php
    generate_footer();
?>
```

This yields the following result:

```
<p>Copyright &copy; 2005 W. Jason Gilmore</p>
```

Passing Arguments by Value

You'll often find it useful to pass data into a function. As an example, let's create a function that calculates an item's total cost by determining its sales tax and then adding that amount to the price:

```
function salestax($price,$tax) {
    $total = $price + ($price * $tax);
    echo "Total cost: $total";
}
```

This function accepts two parameters, aptly named `$price` and `$tax`, which are used in the calculation. Although these parameters are intended to be floats, because of PHP's loose typing, nothing prevents you from passing in variables of any data type, but the outcome might not be as one would expect. In addition, you're allowed to define as few or as many parameters as you deem necessary; there are no language-imposed constraints in this regard.

Once you define the function, you can then invoke it, as was demonstrated in the previous section. For example, the `salestax()` function would be called like so:

```
salestax(15.00,.075);
```

Of course, you're not bound to passing static values into the function. You can pass variables like this:

```
<?php
    $pricetag = 15.00;
    $salestax = .075;
    salestax($pricetag, $salestax);
?>
```

When you pass an argument in this manner, it's called *passing by value*. This means that any changes made to those values within the scope of the function are ignored outside of the function. If you want these changes to be reflected outside of the function's scope, you can pass the argument *by reference,* introduced next.

■ **Note** Note that you don't necessarily need to define the function before it's invoked, because PHP reads the entire script into the engine before execution. Therefore, you could actually call `salestax()` before it is defined, although such haphazard practice is not recommended.

Passing Arguments by Reference

On occasion, you may want any changes made to an argument within a function to be reflected outside of the function's scope. Passing the argument by reference accomplishes this need. Passing an argument by reference is done by appending an ampersand to the front of the argument. An example follows:

```
<?php
    $cost = 20.00;
    $tax = 0.05;
    function calculate_cost(&$cost, $tax)
    {
        // Modify the $cost variable
        $cost = $cost + ($cost * $tax);
        // Perform some random change to the $tax variable.
        $tax += 4;
    }
    calculate_cost($cost,$tax);
    echo "Tax is: ". $tax*100."<br />";
    echo "Cost is: $". $cost."<br />";
?>
```

Here's the result:

```
Tax is 5%
Cost is $21
```

Note that the value of `$tax` remains the same, although `$cost` has changed.

Default Argument Values

Default values can be assigned to input arguments, which will be automatically assigned to the argument if no other value is provided. To revise the sales tax example, suppose that the majority of your sales are to take place in Franklin County, located in the great state of Ohio. You could then assign `$tax` the default value of 5.75 percent, like this:

```
function salestax($price,$tax=.0575) {
   $total = $price + ($price * $tax);
   echo "Total cost: $total";
}
```

Keep in mind that you can still pass `$tax` another taxation rate; 5.75 percent will be used only if `salestax()` is invoked like this:

```
$price = 15.47;
salestax($price);
```

Note that default argument values must be constant expressions; you cannot assign nonconstant values such as function calls or variables.

Optional Arguments

You can designate certain arguments as *optional* by placing them at the end of the list and assigning them a default value of nothing, like so:

```
function salestax($price,$tax="") {
   $total = $price + ($price * $tax);
   echo "Total cost: $total";
}
```

This allows you to call `salestax()` without the second parameter if there is no sales tax:

```
salestax(42.00);
```

This returns the following:

```
Total cost: $42.00
```

If multiple optional arguments are specified, you can selectively choose which ones are passed along. Consider this example:

```
function calculate($price,$price2="",$price3="") {
   echo $price + $price2 + $price3;
}
```

You can then call `calculate()`, passing along just `$price` and `$price3`, like so:

```
calculate(10,"",3);
```

This returns the following value:

```
13
```

Returning Values from a Function

Often, simply relying on a function to do something is insufficient; a script's outcome might depend on a function's outcome, or on changes in data resulting from its execution. Yet variable scoping prevents information from easily being passed from a function body back to its caller, so how can we accomplish this? You can pass data back to the caller by way of the `return` keyword.

return()

The `return()` statement returns any ensuing value back to the function caller, returning program control back to the caller's scope in the process. If `return()` is called from within the global scope, the script execution is terminated. Revising the `salestax()` function again, suppose you don't want to immediately echo the sales total back to the user upon calculation, but rather want to return the value to the calling block:

```
function salestax($price,$tax=.0575) {
   $total = $price + ($price * $tax);
   return $total;
}
```

Alternatively, you could return the calculation directly without even assigning it to `$total`, like this:

```
function salestax($price,$tax=.0575) {
   return $price + ($price * $tax);
}
```

Here's an example of how you would call this function:

```
<?php
     $price = 6.50;
     $total = salestax($price);
?>
```

Returning Multiple Values

It's often quite convenient to return multiple values from a function. For example, suppose that you'd like to create a function that retrieves user data from a database, say the user's name, e-mail address, and phone number, and returns it to the caller. Accomplishing this is much easier than you might think, with the help of a very useful language construct, `list()`. The `list()` construct offers a convenient means for retrieving values from an array, like so:

```
<?php
    $colors = array("red","blue","green");
    list($red,$blue,$green) = $colors; // $red="red", $blue="blue", $green="green"
?>
```

Building on this example, you can imagine how the three prerequisite values might be returned from a function using `list()`:

```
<?php
   function retrieve_user_profile() {
      $user[] = "Jason";
      $user[] = "jason@example.com";
      $user[] = "English";
      return $user;
   }
   list($name,$email,$language) = retrieve_user_profile();
   echo "Name: $name, email: $email, preferred language: $language";
?>
```

Executing this script returns:

```
Name: Jason, email: jason@example.com, preferred language: English
```

This concept is useful and will be used repeatedly throughout this book.

Nesting Functions

PHP supports the practice of *nesting functions,* or defining and invoking functions within functions. For example, a dollar-to-pound conversion function, `convert_pound()`, could be both defined and invoked entirely within the `salestax()` function, like this:

```
function salestax($price,$tax) {
   function convert_pound($dollars, $conversion=1.6) {
      return $dollars * $conversion;
   }
   $total = $price + ($price * $tax);
   echo "Total cost in dollars: $total. Cost in British pounds: "
        .convert_pound($total);
}
```

Note that PHP does not restrict the scope of a nested function. For example, you could still call `convert_pound()` outside of `salestax()`, like this:

```
salestax(15.00,.075);
echo convert_pound(15);
```

Recursive Functions

Recursive functions, or functions that call themselves, offer considerable practical value to the programmer and are used to divide an otherwise complex problem into a simple case, reiterating that case until the problem is resolved.

Practically every introductory recursion example involves factorial computation. Yawn. Let's do something a tad more practical and create a loan payment calculator. Specifically, the following example uses recursion to create a payment schedule, telling you the principal and interest amounts required of each payment installment to repay the loan. The recursive function, `amortizationTable()`, is introduced in Listing 4-1. It takes as input four arguments: `$paymentNum`, which identifies the payment number, `$periodicPayment`, which carries the total monthly payment, `$balance`, which indicates the remaining loan balance, and `$monthlyInterest`, which determines the monthly interest percentage rate. These items are designated or determined in the script listed in Listing 4-2, titled `mortgage.php`.

Listing 4-1. *The Payment Calculator Function, amortizationTable()*

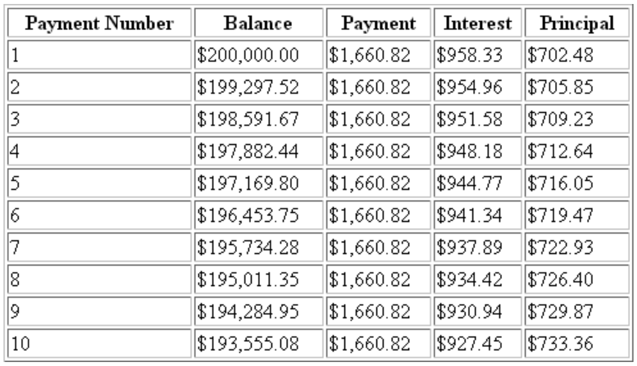
```
function amortizationTable($paymentNum, $periodicPayment, $balance,
                           $monthlyInterest) {
    $paymentInterest = round($balance * $monthlyInterest,2);
    $paymentPrincipal = round($periodicPayment - $paymentInterest,2);
    $newBalance = round($balance - $paymentPrincipal,2);
    print "<tr>
           <td>$paymentNum</td>
           <td>\$".number_format($balance,2)."</td>
           <td>\$".number_format($periodicPayment,2)."</td>
           <td>\$".number_format($paymentInterest,2)."</td>
           <td>\$".number_format($paymentPrincipal,2)."</td>
           </tr>";
     # If balance not yet zero, recursively call amortizationTable()
     if ($newBalance > 0) {
        $paymentNum++;
        amortizationTable($paymentNum, $periodicPayment, $newBalance,
                          $monthlyInterest);
     } else {
        exit;
     }
} #end amortizationTable()
```

After setting pertinent variables and performing a few preliminary calculations, Listing 4-2 invokes the `amortizationTable()` function. Because this function calls itself recursively, all amortization table calculations will be performed internal to this function; once complete, control is returned to the caller.

Listing 4-2. *A Payment Schedule Calculator Using Recursion (mortgage.php)*

```
<?php
   # Loan balance
   $balance = 200000.00;

   # Loan interest rate
   $interestRate = .0575;

   # Monthly interest rate
   $monthlyInterest = .0575 / 12;

   # Term length of the loan, in years.
   $termLength = 30;

   # Number of payments per year.
   $paymentsPerYear = 12;

   # Payment iteration
   $paymentNumber = 1;

   # Perform preliminary calculations
   $totalPayments = $termLength * $paymentsPerYear;
   $intCalc = 1 + $interestRate / $paymentsPerYear;
   $periodicPayment = $balance * pow($intCalc,$totalPayments) * ($intCalc - 1) /
                                 (pow($intCalc,$totalPayments) - 1);
   $periodicPayment = round($periodicPayment,2);

   # Create table
   echo "<table width='50%' align='center' border='1'>";
   print "<tr>
          <th>Payment Number</th><th>Balance</th>
          <th>Payment</th><th>Interest</th><th>Principal</th>
          </tr>";

   # Call recursive function
   amortizationTable($paymentNumber, $periodicPayment, $balance, $monthlyInterest);

   # Close table
   print "</table>";
?>
```

Figure 4-1 shows sample output, based on monthly payments made on a 30-year fixed loan of $200,000.00 at 6.25 percent interest. For reasons of space conservation, just the first 10 payment iterations are listed.



Payment Number	Balance	Payment	Interest	Principal
1	$200,000.00	$1,660.82	$958.33	$702.48
2	$199,297.52	$1,660.82	$954.96	$705.85
3	$198,591.67	$1,660.82	$951.58	$709.23
4	$197,882.44	$1,660.82	$948.18	$712.64
5	$197,169.80	$1,660.82	$944.77	$716.05
6	$196,453.75	$1,660.82	$941.34	$719.47
7	$195,734.28	$1,660.82	$937.89	$722.93
8	$195,011.35	$1,660.82	$934.42	$726.40
9	$194,284.95	$1,660.82	$930.94	$729.87
10	$193,555.08	$1,660.82	$927.45	$733.36

Figure 4-1. *Sample output from mortgage.php*

Employing a recursive strategy often results in significant code savings and promotes reusability. Although recursive functions are not always the optimal solution, they are often a welcome addition to any language's repertoire.

Variable Functions

One of PHP's most attractive traits is its syntactical clarity. On occasion, however, taking a somewhat more abstract programmatic route can eliminate a great deal of coding overhead. For example, consider a scenario in which several data-retrieval functions have been created: `retrieveUser()`, `retrieveNews()`, and `retrieveWeather()`, where the name of each function implies its purpose. In order to trigger a given function, you could use a URL parameter and an `if` conditional statement, like this:

```
<?php
    if ($trigger == "retrieveUser") {
        retrieveUser($rowid);
    } else if ($trigger == "retrieveNews") {
        retrieveNews($rowid);
    } else if ($trigger == "retrieveWeather") {
        retrieveWeather($rowid);
    }
?>
```

This code allows you to pass along URLs like this:

```
http://www.example.com/content/index.php?trigger=retrieveUser&rowid=5
```

The `index.php` file will then use `$trigger` to determine which function should be executed. Although this works just fine, it is tedious, particularly if a large number of retrieval functions are required. An alternative, much shorter means for accomplishing the same goal is through variable functions. A *variable function* is a function whose name is also evaluated before execution, meaning that its exact name is not known until execution time. Variable functions are prefaced with a dollar sign, just like regular variables, like this:

```
$function();
```

Using variable functions, let's revisit the previous example:

```
<?php
    $trigger($rowid);
?>
```

Although variable functions are at times convenient, keep in mind that they do present certain security risks. Most notably, an attacker could execute any function in PHP's repertoire simply by modifying the variable used to declare the function name. For example, consider the ramifications of modifying the `$trigger` variable in the previous example to contain the value `exec` and modifying the `$rowid` variable to contain `rm -rf /`. PHP's `exec()` command will happily attempt to execute its argument on the system level. The command `rm -rf /` will attempt to recursively delete all files, starting at the root-level directory. The results could be catastrophic. Therefore, as always, be sure to sanitize all user information; you never know what will be attempted next.

Function Libraries

Great programmers are lazy, and lazy programmers think in terms of reusability. Functions form the crux of such efforts, and are often collectively assembled into *libraries* and subsequently repeatedly reused within similar applications. PHP libraries are created via the simple aggregation of function definitions in a single file, like this:

```
<?php
   function local_tax($grossIncome, $taxRate) {
      // function body here
   }
   function state_tax($grossIncome, $taxRate) {
      // function body here
   }
   function medicare($grossIncome, $medicareRate) {
      // function body here
   }
?>
```

Save this library, preferably using a naming convention that will clearly denote its purpose, like `taxes.library.php`. You can then insert this function into scripts using `include()`, `include_once()`, `require()`, or `require_once()`, each of which was introduced in Chapter 3. (Alternatively, you could use PHP's `auto_prepend` configuration directive to automate the task of file insertion for you.) For example, assuming that you titled this library `taxation.library.php`, you could include it into a script like this:

```
<?php
    require_once("taxation.library.php");
    ...
?>
```

Once included, any of the three functions found in this library can be invoked as needed.

Summary

This chapter concentrated on one of the basic building blocks of modern-day programming languages: reusability through functional programming. You learned how to create and invoke functions, pass information to and from the function block, nest functions, and create both recursive and variable functions. Finally, you learned how to aggregate functions together as libraries and include them into the script as needed.

The next chapter introduces PHP's array functionality, covering the language's vast array of management capabilities and introducing PHP 5's new array-handling features.