PHP 101 (part 5) – The Wonderland Factor

By Vikram Vaswani and Harish Kamath

Table of Contents

A Journey To Remember

If you've been following along, you already know how PHP can be used to extract data from a database...it's as easy as apple pie. But in the real world, data isn't always stored in neat rows and columns, and you're quite likely to come across situations where the data you need is actually stored in a bunch of ASCII text files.

You have three options here. You could con someone into the mind−numbing task of inserting the textual data into a database. You could throw up your hands in despair and look for another job. Or you could use PHP's built−in file functions to extract and format the data.

The first option requires guile and cunning. The second is unacceptable. And the third requires this issue of PHP 101.

Over the next few pages, we'll be discussing the basics of PHP's file functions, together with some interesting examples. And then we'll explore the murky tunnels of user−defined functions, and delve deep into the mysteries of local and global variables, return values and function parameters. Pack some warm clothes, bring enough food and get someone to feed the dogs − this is one journey you don't want to miss!

Windy Nights

From the days of C, programmers have had the ability to create files on the system, read them and perhaps even execute them via the shell. And, like all good programming languages, PHP comes with a bunch of functions which allow you to read and write files with minimum effort.

Let's get straight into the nitty−gritty of reading a file. Create a text file and populate it with some text − here's what our file, which we've named "random.txt", looks like:

```
It was a dark and stormy evening. Outside the pub, the wind
wailed and moaned, singing dirges to long−lost sailors and
making the timbers creak. Inside, all was silent except for a
brave nightlight that kept the ghosts awake.

Oh, what I wouldn't give for a slug of ale just about now!
```

Before you can use PHP to read the contents of this file, you need to ensure that you have permission to read it. On *NIX, this is accomplished via the "chmod" command.− try

```
$ chmod 744 random.txt
```

to make it world−readable.

And here's some simple PHP code that reads the file and returns the contents and file size:

```
<?php // read file $bytes = readfile("random.txt"); // display
size echo "<br>File is $bytes bytes in size."; ?>
```

And here's the output:

```
It was a dark and stormy night. Outside the pub, the wind
wailed and moaned, singing dirges to long−lost sailors and
making the timbers creak. Inside, all was silent except for a
brave nightlight that kept the ghosts awake. Oh, what I
wouldn't give for a slug of ale just about now! File is 286
bytes in size.
```

As you can see, readfile() doesn't do much − it simply reads the file and displays the contents. It also returns the size of the file in bytes − we've displayed this using an echo() above.

A far more useful function, and one which lets you format the extracted data, is the file() function, which allows you to read the file into a regular PHP array. Each element of the array represents one line of the file, and therefore, the size of the array is representative of the number of lines in the file. Take a look:

```
<?php // set filename $filename = "random.txt"; // read file
into array $contents = file($filename); // get array length
$length = sizeof($contents); echo "<b>File $filename contains
$length line(s)</b><p>"; // display each line with "for" loop
for($counter=0; $counter<$length; $counter++) { echo
"$contents[$counter]<br>"; } ?>
```

In this case, we've used the file() function to read the contents of the entire file into an array called $contents. Using the sizeof() function, we can obtain the length of the array, which is the same as the number of lines present in the text file. The "for" loop is used to read and display each element of the array.

And here's the output:

```
File random.txt contains 3 line(s) It was a dark and stormy
night. Outside the pub, the wind wailed and moaned, singing
dirges to long−lost sailors and making the timbers creak.
Inside, all was silent except for a brave nightlight that kept
the ghosts awake. Oh, what I wouldn't give for a slug of ale
just about now!
```

So Who Are You, Anyway?

In the examples you've just seen, we've made one basic assumption − the file that PHP is trying to read already exists. However, in the big bad world, this assumption can sometimes cost you hours of debug time if the file isn't actually present on the system.

To avoid situations like this, PHP offers the file_exists() function, which can be used to test for the presence of the file and generate an appropriate error message if it isn't found. Let's modify the example above to include this capability:

```
<?php // set filename $filename = "random.txt"; // check for
file if (file_exists($filename)) { // read file into array
$contents = file($filename); // get array length $length =
sizeof($contents); echo "<b>File $filename contains $length
line(s)</b><p>"; // display each line with "for" loop
for($counter=0; $counter<$length; $counter++) { echo
"$contents[$counter]<br>"; } } else { echo "<b>File not
found!</b>"; } ?>
```

Now try it out − rename or delete your file, and watch as PHP gracefully exits with your customized error message.

PHP also comes with a bunch of functions that can provide you with detailed information about the type of file − here's a list:

```
is_dir() − checks whether the specified file is a directory
is_executable() − checks whether the specified file is
executable is_link() − checks whether the specified file is a
link is_readable() − checks whether the specified file is
readable is_writeable() − checks whether the specified file is
writable filegroup() − returns the group fileowner() − returns
the owner of the file fileperms() − returns the current
permissions on the file filesize() − returns the size of the
file filetype() − returns the type of the file
```

And here's an example:

```
<?php // set filename $filename = "random.txt"; // check for
file if (file_exists($filename)) { echo
"<b>$filename</b><br>"; // check if it is a directory if
(is_dir($filename)) { echo "File is a directory
"; } // check if it is executable if
(is_executable($filename)) { echo "File is executable
"; } // check if it is readable if (is_readable($filename)) {
echo "File is readable
"; } // check if it is writeable if (is_writeable($filename))
```

```
{ echo "File is writeable
"; } echo "File size is " . filesize($filename) . "
bytes<br>"; echo "File group is " . filegroup($filename) .
"<br>"; echo "File owner is " . fileowner($filename) . "<br>";
echo "File permissions are " . fileperms($filename) . "<br>";
echo "File type is " . filetype($filename) . "<br>"; } else {
echo "<b>File not found!</br>"; } ?>
```

Here's the output:

```
File is readable File is writeable File size is 286 bytes File
group is 100 File owner is 538 File permissions are 33188 File
type is file
```

Graffiti For The Masses

So much for reading files − how about actually writing one? In PHP, this is somewhat more complex, involving as it does the use of file "pointers" to write data to a file.

The function that does most of the work here is a little munchkin called fopen(), which opens a pointer to the file. fopen() requires two parameters − the name of the file, and the mode in which the file is to be opened. This "mode" is one of the following:

Read−only mode, activated by the parameter "r". Here, the file pointer is initialized at the first line of the file.

Write−only mode, activated by the parameter "w". Here, the file is created anew and the pointer is initialized at the first line of the file. If the file already exists, all its contents are lost.

Read/write mode, activated by the parameter "r+". Here, the file is opened for reading and writing, with the pointer initialized at the first line of the file.

Append mode(write only), activated by the parameter "a". In this case, the pointer is placed at the *end* of the file, allowing you to append new content to the end of the file. If the file doesn't exist, a new one is created.

Append mode(read/write only), activated by the parameter "a+". Here too, the pointer is placed at the *end* of the file, allowing you to append new content to the end of the file. If the file doesn't exist, a new one is created. You can also read from the file.

Once you've got the file opened in the right mode, it's time to actually write something to it with the fputs() function. Our next example demonstrates this more clearly:

```
<?php // set the file name $filename = "/tmp/myfile.txt"; //
open the file $handle= fopen($filename,'a'); // create the
string $string = "This is a test of the Emergency Broadcast
System. If this had been an actual emergency, do you really
think we'd stick around to tell you?"; // write the string to
the file handle fputs($handle, $string); // close the file
fclose($handle); ?>
```

In this case, we've first opened a file pointer to the file; next, we've used the fputs() function to actually write a string to the file. The fputs() function needs two parameters − the file pointer to use, and the string to be written to the file. Once your file has been written, you can close the file with fclose(), and check the contents with readfile() or file().

Our next example combines everything you've just learnt to create the world's longest story − check it out!

```
<html> <head> <basefont face="Arial"> </head> <body> <?php //
set the file name $filename = "/tmp/graffiti.dat"; // open the
file $handle= fopen($filename,'a+'); // write the string to
the file handle fputs($handle, $graffiti); // close the file
fclose($handle); ?> <form action=graffiti.php4 method=get>
```

```
<input type=text size=30 name=graffiti> <input type=submit
name=submit value="Add your two bits!"> </form> <?php //
display current contents of file if available if
(file_exists($filename)) { echo "<b>Current graffiti reads:
</b>"; readfile($filename); } else { echo "File not found!"; }
?> </body> </html>
```

In this example, the user is presented with a simple form containing a text box. Once the user enters something into the text box and submits the form, the text string is inserted into the file "graffiti.dat". The page is then reloaded, the file is read and the complete contents displayed; the user now has the opportunity to enter something new into the box and have this text appended to the end of the existing text in the file.

Calling Godzilla

Next up, user−defined functions.

Most programming languages provide a number of functions that make coding easier − in PHP 101, we frequently use the echo() function, and in the last issue, we used a bunch of database−related functions. But sometimes, it's just more convenient to roll your own − and so, PHP also allows you to define your own custom functions, which you can program in accordance with your heart's darkest desires.

In PHP, a user−defined function is simply a set of program statements which perform a specific task, and which can be called, or executed, from anywhere in your PHP script.

There are two important reasons why separating your code into functions is a "good thing". First, this allows you to isolate your code into easily identifiable subsections, thereby making it easier to understand and debug. And second, a function makes your program modular by allowing you to write a piece of code once and then re−use it multiple times within the same program.

Here what a function definition looks like:

```
<? function godzilla() { statement 1; statement 2; statement
3; ... ... } ?>
```

And once it's defined, you can execute the statements within the function simply by calling it − like this:

```
<? function godzilla() { statement 1; statement 2; statement
3; ... ... } godzilla(); ?>
```

When the PHP parser encounter a function call such as the one above, control of the program shifts to the location where the function has been defined. Once the statements in the function block have been executed, control shifts back to the location where the function was invoked.

Qabout an example?

```
<html> <head> <basefont face=Arial>
</head> <body> <?php function
answer_yes() { echo "Yes!"; } function
answer_no() { echo "No!"; } ?> Would you
watch an Al Pacino movie without
thinking twice? <br> <? answer_yes(); ?>
<p> Does 2 + 2 equal 6? <br> <?
answer_no(); ?> <p> Is this the coolest
PHP tutorial on the planet? <br> <?
answer_yes(); ?> </body> </html>
```

And here's the output:

```
Would you watch an Al Pacino movie
without thinking twice? Yes! Does 2 + 2
equal 6? No! Is this the coolest PHP
tutorial on the planet? Yes!
```

You can also invoke a function before it's been defined.

```
<html> <head> <basefont face=Arial>
</head> <body> Would you watch an Al
Pacino movie without thinking twice?
<br> <? answer_yes(); ?> <p> Does 2 + 2
equal 6? <br> <? answer_no(); ?> <p> Is
this the coolest PHP tutorial on the
planet? <br> <? answer_yes(); ?> <?php
function answer_yes() { echo "Yes!"; }
function answer_no() { echo "No!"; } ?>
</body> </html>
```

Arguments And Responses

Now, all this is well and good, but it's not exactly setting the house on fire, is it? Well, there are a couple of things you need to add into the mix for your user−defined functions to actually start earning their keep: return values and function arguments.

Usually, when a function is invoked, it generates a "return value" − this may be value of the last expression evaluated, or a value explicitly returned to the main program via the "return" statement. Our next example demonstrates how a return value works:

```
<html> <head> <basefont face=Arial> </head> <body> <?php
function answer_yes() { $answer = "Yes!"; return $answer; }
function answer_no() { $answer = "No!"; return $answer; } ?>
Would you watch an Al Pacino movie without thinking twice?
<br> <? $response = answer_yes(); echo $response; ?> <p> Does
2 + 2 equal 6? <br> <? $response = answer_no(); echo
$response; ?> <p> Is this the coolest PHP tutorial on the
planet? <br> <? $response = answer_yes(); echo $response; ?>
</body> </html>
```

The output of the script above has not changed; however, there are significant differences in the manner of its execution. In the example above, the functions do not provide any output − they simply assign a string to the variable $answer. When the function is invoked, it passes the value of this variable to the main program, where it is assigned to the variable $response and displayed on the page.

And just as a PHP function can return values *to* the main program, it can also accept "function arguments" *from* the main program. Take a look:

```
<html> <head> <basefont face=Arial> </head> <body> <?php
function multiply($alpha, $beta) { $product = $alpha * $beta;
return $product; } $num1 = 9; $num2 = 16; ?> <? echo $num1; ?>
times <? echo $num2; ?> is <? $answer = multiply($num1,
$num2); echo $answer; ?> </body> </html>
```

The first thing you should notice here is that the function definition has changed − the parentheses, which seemed to be there only for decorative purposes, now enclose two PHP variables called $alpha and $beta. When arguments are passed to the function, they will be stored in these PHP variables for the duration of the function and can be further acted upon by the statements within the function.

In the example above, $num1 (9) and $num2 (16) are the two arguments passed to the function multiply() The function multiply() accepts the two arguments, performs a mathematical operation on them, stores the result in $product and returns $product to the main program as $answer.

Here's the output:

```
9 times 16 is 144
```

The multiply() function can be used with any pair of numbers − as the arguments passed to it change, so will the result. This is how functions allow you to re−use the same piece of code over and over again, without any need to re−write code each time.

Flavour Of The Month

An important point to be noted here is that it isn't possible to use variables outside the function within the function definition...unless you use the special "global" keyword. This keyword, when used within a function, allows you to use a variable from outside the function within the function as well. To illustrate this, consider the following two examples:

```
<?php // set variable $flavour = "strawberry"; // function to
change variable value function change_flavour($name) {
$flavour = $name; return $flavour; } echo "Before calling the
function, the flavour is $flavour.<p>"; // change variable
value change_flavour("blueberry"); echo "After calling the
function, the flavour is $flavour."; ?>
```

Here's the output:

```
Before calling the function, the flavour is strawberry. After
calling the function, the flavour is strawberry.
```

And here's what you need to do to make the variable a global variable:

```
<?php // set variable $flavour = "strawberry"; // function to
change variable value function change_flavour($name) { //make
the variable global global $flavour; $flavour = $name; return
$flavour; } echo "Before calling the function, the flavour is
$flavour.<p>"; // change variable value
change_flavour("blueberry"); echo "After calling the function,
the flavour is $flavour."; ?>
```

Here's the output:

```
Before calling the function, the flavour is strawberry. After
calling the function, the flavour is blueberry.
```

The Wonderland Factor

Our original intention was to make PHP's user−defined functions clearer via a simple calculator function, one which accepted two numbers and added, subtracted, multiplied and divided them.

How boring!

How about, instead, if we create a calculator, one which accepts two numbers and performs mathematical operations on them? How about if we add a twist by claiming this calculator to be the one Alice would carry on her journey through Wonderland (assuming she needed one)? And how about if we substantiate that claim by adding the Wonderland factor, a random number that skews the results of your calculations so that they're never correct?

```
<?php // generate the wonderland factor // the rand() function
generates a random number // in the range specified
$wonderland = rand(2, 100); function calculate($type, $num1,
$num2) { global $wonderland; // possible values for $type are:
// 1 for addition // 2 for subtraction // 3 for multiplication
// 4 for division switch($type) { case 1: $operation =
"addition"; $result = $num1 + $num2 + $wonderland; $answer =
"$num1 plus $num2 is $result"; return $answer; break; case 2:
$operation = "subtraction"; $result = $num1 − $num2 −
$wonderland; $answer = "$num1 minus $num2 is $result"; return
$answer; break; case 3: $operation = "multiplication"; $result
= $num1 * $num2 * $wonderland; $answer = "$num1 times num2 is
$result"; return $answer; break; case 4: $operation =
"division"; if($num2 != 0) { $result = ($num1 / $num2) /
$wonderland ; } $answer = "$num1 by $num2 is $result"; return
$answer; break; default: $result = 0; } return $result; }
$type = 1; $x = 9; $y = 17; $answer = calculate($type, $x,
$y); echo $answer; ?> </body> </html>
```

Each time you reload the page, you should get a different answer − that's the random number at work!

Coding For The Unexpected

So far, all the functions you've seen accept a fixed set of arguments. However, you might often find yourself in a situation where you're not quite sure of how many arguments you will be passing.

In the old days of PHP3, you had no choice but to grin and bear it. However, PHP4 allows you to create functions which will accept any number of arguments, regardless of whether or not they've been defined in the function definition.

There are two PHP functions that add this functionality: func_num_args(), which returns the number of variables passed to the function, and func_get_args(), which lets you obtain those variables in the form of an array. Our next example will make this clearer.

```
<html> <head> <basefont face=Arial> </head> <body> <?php
function cart() { $arguments = func_get_args(); return
$arguments; } $items = cart("toothpaste", "aspirin", "dog
food", "carving knife"); ?> You have selected <? echo
sizeof($items); ?> items. They are: <ol> <? for ($x=0;
$x<sizeof($items); $x++) { echo "<li>" . $items[$x]; } ?>
</ol> </body> </html>
```

And here's the output:

```
You have selected 4 items. They are: 1. toothpaste 2. aspirin
3. dog food 4. carving knife
```

And that's about it for this issue of PHP 101. You should now know enough to begin writing PHP code for your Web site. We'll continue to bring you tutorials on other aspects of the language − tell us what you'd like to read about! And until next time...stay healthy!