

## PHP Back to Basics – Functions.

Often when your coding you'll find that blocks of code in your scripts are fairly similar, with only a few minor differences. For example your PHP code could output the same HTML layout each time, but the actual data within the layout would be different. This leads to a lot of code repetition, which creates longer code, more complex debugging and of course it takes longer to create. If you later want to change the way the data is displayed, you would have to make the same type of changes over and over again, which is time consuming, and also gives a greater chance of introducing errors into your code.

### What are Functions?

Functions remedy this situation by allowing you to create "building blocks" of code. These contain the blocks of code that are exactly the same, but allow you to pass parameters to the function which are then inserted into the relevant places in the code, so different data can be used. A very similar concept is used with extensions in Dreamweaver MX. Extensions generate the same basic code, and this is "hard coded" into the extension, however different data can be inserted into this code to produce different results, which are passed through the extension dialog box.

Functions don't necessarily have to output HTML, for example, instead they can just pass back a value to the code that called the function. As an example, if you had a complex calculation, you would place the calculation in a function, pass the data that you want to work with to the function, and then the function will return the result to the code that called it, saving you having to type the calculation over and over again in your code.

Most of the PHP commands are themselves functions. As an example, if we use the `count()` command which returns the number of elements in an array that we looked at in [part 3 of this series](#), you're actually using a function. You pass an array to the `count()` command, internal code is run and then the result is returned back to the code that called the command.

### Creating a function

The basic structure of a function is shown below:

```
function name($optional_parameters) {  
    // PHP code goes here  
}
```

First, we have the keyword **function**. We then have the function name, and a set of parentheses `()` which contain any arguments that the function will use, although if no arguments are expected they can be left empty. We then have an opening brace `{`, the PHP code that makes up the function, and then a closing brace `}` to end the function.

As a simple example, try the following code shown below:

```
<?php  
function helloWorld() {  
    echo "Hello World!";  
}
```

```
?>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
<title>Example Function</title>
</head>

<body>
<?php helloWorld(); ?>
</body>
</html>
```

As you can see from the above code, any PHP functions are normally placed at the top of the page above the HTML code. This helps to keep the code neat and easily readable. We then have a simple function named `helloWorld()`, which doesn't accept any arguments, with some simple PHP code to output the words "Hello World!" to the screen.

In the body of the page, we call the function by simply using the name of the function. When the code above is run, you'll see "Hello World!" in the browser window.

## Adding arguments to a function

Now we've created a simple function, we're next going to look at making the function accept arguments to make it more useful. This is very simple to do, and simply involves adding variables to the function parentheses to hold the data that is passed to the function.

As a simple example, we'll create a function that will accept an argument, and performs a simple calculation. Add the following function to the top of a new PHP page.

```
<?php
function calculate($value){
    $newValue = $value * 10;
    echo $value . " * 10 is " . $newValue;
}
?>
```

Here, we have a simple function called `calculate()`, that accepts an argument which is held in a variable called `$value`. Inside the function we multiply the number held in `$value` by 10 and then output the new value to the browser. To call the function, add the following PHP code to the body of the page.

```
<?php
$test = 15;
calculate($test);
?>
```

First, we define a new variable called `$test` and set its value to 15. We then call the `calculate()` function, passing it the variable `$test` as the argument.

When the code is run, you will see "15 \* 10 is 150" output to the browser.

To make the function accept more arguments, you simply add more variables to the parentheses as shown in the example function below.

```
<?php
function calculate($value1,$value2){
    $newValue = $value1 * $value2;
    echo $value1 . " * " . $value2 . " = " . $newValue;
}
?>
```

Here we have a function that accepts two arguments, held in **\$value1** and **\$value2**. Inside the function we multiply **\$value1** by **\$value2** and place the result in **\$newValue**. We then output the answer to the browser.

To call the function, add the following code to the body of the page.

```
<?php
$test1 = 15;
$test2 = 5;
calculate($test1,$test2);
?>
```

First we setup two variables, **\$test1** and **\$test2**, and place the values 15 and 5 into them. We then call the `calculate()` function, passing it the two variables.

When the code is run, you will see the output "15 \* 5 = 75" in the browser. Although this is a very simple example, much more complex calculations can be performed. Functions can accept any type of variable as arguments, for example numbers, strings or arrays, and we'll be using this ability as we progress through this series.

It's important to note that when you define a function as having arguments, you pass a variable for each argument when you call the function, or your code will fail with an error. In the next section, we'll look at how to specify optional arguments for a function.

## Setting default arguments for a function

As mentioned in the previous section, if you specify arguments for a function, you must pass a variable for each argument when you call the function, or the code will fail. However, there is a way round this by using default arguments for the function. When you define default arguments, you set a value for the argument in the function definition, which will be used if no argument is specified when the function is called. If an argument is passed when the function is called, then this will override the default argument, and its value will be used instead.

As an example, we'll adapt the `calculate()` function that we created earlier, so that it has a default argument. The code to do this is shown below:

```
<?php
function calculate($value1,$value2 = 20){
    $newValue = $value1 * $value2;
    echo $value1 . " * " . $value2 . " = " . $newValue;
}
?>
```

In the above code, you can see that the change is underlined. Here, we've specified a value for the second argument, `$value2`, so if nothing is passed to the function for this argument, the value 20 will be used instead of the code failing. If a variable is passed to the function for the second argument then this will be used instead of 20, as shown below.

First, we'll call the `calculate()` function and passing it variables for both of the arguments using the code shown below.

```
<?
$value1 = 10;
$value2 = 15;
calculate($value1,$value2);
?>
```

When this code is run, the function will output "`10 * 15 = 150`" to the browser, and you can see that the default value for `$value2` has been ignored.

Next, we'll call the function, but only pass a value for `$value1`, as shown in the code below.

```
<?
$value1 = 10;
calculate($value1);
?>
```

This time, when you run the code the function will output "`10 * 20 = 200`" to the browser. Because we only passed a variable for `$value1`, the default value for `$value2` (20) is used instead.

## Returning a value from a function

So far, the functions that we've looked at have produced output using the code inside the function. This doesn't always have to be the case however, and functions don't have to necessarily produce any output. Instead they can just return a value to the code that called the function, so you can then work further with that value before you output it to the browser.

To return a value from a function, you use the **return** keyword as shown in the example below.

```
<?php
function calculate($value1,$value2 = 20){
    $newValue = $value1 * $value2;
    return $newValue;
}
?>
```

In the code above, instead of outputting the value from the calculation, we return the result stored in **`$newValue`** to the code that called the function.

To call the function, we can use the code shown below:

```
<?
$value1 = 10;
$value2 = 15;
$answer = calculate($value1,$value2);
```

```
echo "Answer is " . $answer;  
?>
```

When this code is run, the following text is output to the browser "Answer is 150", where 150 is the result of the calculation and is returned by the function and stored in **\$answer**.

## Functions and Variable Scope

When you define a variable inside a function, you cannot access that variable from outside the function, as shown in the example below.

```
<?php  
function example(){  
    $test = "This is an example variable";  
}  
example();  
echo "Variable test contains " . $test;  
?>
```

When this code is run, the text "Variable test contains " is output to the browser, and the contents of variable `$test` is not printed, as the variable `$test` only exists within the function itself. This means that you can quite safely define the same variable names within different functions knowing that they will not conflict in any way.

This also works the other way round, in that the PHP code in functions cannot normally access variables which are defined in the code outside of the function. There is however an exception to this rule, which we're going to look at next.

### Using the global command

As mentioned above, code in a function is normally totally cut off from the code outside the function, and variables which are defined outside of the function cannot be read from inside the function, and vice versa. In some cases however, you may want to access an important variable that has been defined in your main code from within a function, and you can do this with the **global** command, as shown below.

```
<?php  
$name = "DMX Zone";  
  
function outputName(){  
    global $name;  
    echo "This is an example for " . $name;  
}  
  
outputName();  
?>
```

When this code is run, you will see the text "This is an example for DMX Zone" output to the browser. Normally the value in `$name` would not be available inside the function, but because we have defined `$name` with the `global` command inside the function, the value is available to our code within the function.

Once you have defined a variable as being global, you can also change its value from within the function, and this change will be reflected in the code outside of the function. This can be seen in the example code below.

```
<?php
$name = "DMX Zone";

function outputName(){
    global $name;
    echo "This is an example for " . $name;
    $name = "Dreamweaver MX";
}

outputName();
echo "<br>";
echo "New value for Name is: " . $name;
?>
```

When you run this code, you will see the following output in your browser.

```
This is an example for DMX Zone
New value for Name is: Dreamweaver MX
```

First, from our main code we set the variable `$name` to "DMX Zone". Inside the function we define `$name` as a global variable so that we can access it from within the function. We then output the current value of `$name` and change the contents of the variable. Finally, again from our main code we output the contents of `$name`, and you can see that the contents have been changed from inside our function, and are still available once the function has been run.

It's important to be careful where you use this behavior, as because functions do not normally change variables, it can make it harder to debug your code, especially when you come back to your code at a later date, to make a change for example. It's a good idea to insert PHP comments noting which variables are global variables in your code for your own benefit, or for anyone else who may have to change the code in the future.

## Using the Static command

The final command that we're going to look at is the **static** command. Normally, when you run a function, once the code inside the function has completed, the contents of the variables in the functions are lost. It may be useful though in some situations to retain the value of certain variables, so they are available the next time the function is run.

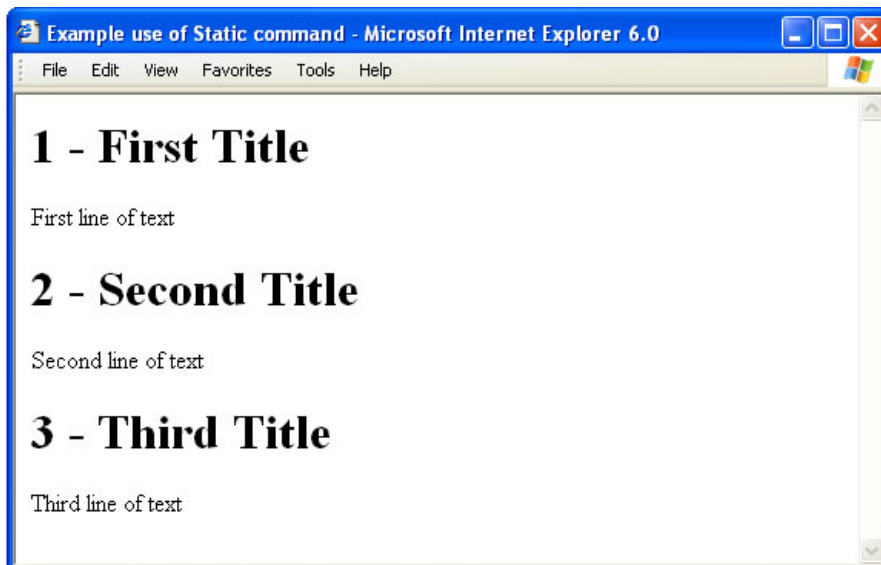
Although you could achieve this using a variable in your main code, accessed from your function using the `global` command, this is more than necessary if you're not intending to use the variable outside of your function and in the rest of your code. If you wish to keep the contents of a variable in your function between calls, but don't need to make the variable available to your main code, you can use the `static` command, as shown in the example below.

```
<?php
function outputTitle($title){
    static $headingNumber = 1;
    echo "<h1>" . $headingNumber . " - " . $title . "</h1>";
    $headingNumber++;
}
outputTitle("First Title");
echo "First line of text";
outputTitle("Second Title");
echo "Second line of text";
outputTitle("Third Title");
echo "Third line of text";
?>
```

First, inside the function, we declare the variable `$headingNumber` with the `static` command, and give it an initial value of 1. This variable is only defined once, when the function is first called, and is not reset back to 1 the next time the function is called, so the value in `$headingNumber` is saved. We then output some text to the browser which prints the value in `$headingNumber` and the value in `$title` which is passed to the function when it's called. We then add 1 to the value in `$headingNumber`.

Next, in our main code, we call the `outputTitle()` function passing it some text, `$headingNumber` is set to 1 and the heading is printed. We then call the function again, and `$headingNumber` will now contain 2, which is printed along with the heading as the value has been saved. Finally, we call the function again and `$headingNumber` will now contain 3.

When the code is run, you will see output similar to that shown below:



As you can see, the contents of `$headingNumber` are stored between function calls.

## Summary

In this tutorial, we've looked at functions and how they can help you lay out your code in a more organized and logical way, which makes debugging much easier, and by avoiding repetition of code makes your code more efficient and easier to update in the future.

We first saw how to create functions, and how to make them accept arguments, so you can pass data for the function to work with, as well as how to specify default arguments which are used if no arguments are passed to the function. We then looked at how to return a value from the function to the calling code, so that it can be output or be passed to other functions for further calculations, for example.

Finally, we looked at variable scope in functions, and how we can use the global command to read and manipulate variables from the main code from within a function. We also looked at the static command which allows you to save the values contained in variables in the function, so the data is not lost once the function has been executed.