CS 85: PHP Programming

Santa Monica College Computer Science & Information Systems Dept.

Module 3: Functions

Functions

include()

required()

# Functions

Functions are an important part of many programming languages. Functions allows programmers to group related PHP statements into a single unit. This allows for more reusable code that leads to more consistent, reliable code that is easier to maintain by developers. In this module we will cover how to code user defined functions and then how to include user defined functions in your scripts.

The advantages of creating user defined functions, is the ability to reuse code instead of writing new code. By reusing code, it reduces the cost to develop, test, document and maintain code by developers. The less code to write, the faster the code is developed. By using existing functions that have been tested, it will allow for reliable code and lastly the code will be more consistent. If a team of developers are reusing the same functions, then it will automatically make different dev team members code consistent.

Some consider the real power of PHP comes from its built in functions; it has more than 1000 built-in functions. A list of built in functions can be found at PHP Manual's function reference http://www.php.net.

# Introduction

*Functions* are a way to group common tasks together that will be repeatedly re-used simply by calling the function name.

Functions in computer programming are much like mathematical functions: You can give the function values to work with and get a result without having to do any calculations yourself. echo(), print() or array sort() are PHP built in functions but a programmer can use user defined functions to meet their specific needs such as validating user input or printing out an array.

# User Defined Function

The first step in including a user defined function in your script is to define the function. Building a function is simple. Use the keyword function followed by the function’s name. Followed by a set of parentheses which would be contain the function parameters. Next use a pair of curly braces ({ statements; }) to combine a series of statement lines into one block of code that is executed when the function is called.

Syntax:

function *functionName*() {  
 *code to be executed*;  
}

<?php

function functionName() {

statement(s);   
statement(s);  
statement(s);

statement(s);  
}

?>

A function declaration starts with the word "function":

**Note:** A function name can start with a letter or underscore (not a number).

**Tip:** Give the function a name that reflects what the function does! Function names are NOT case-sensitive but it is good practice to use the same case when calling the function.

In the example below, we create a function named "writeMsg()". The opening curly brace ( { ) indicates the beginning of the function block of code that will run when the function called and the closing curly brace ( } ) indicates the end of the function’s block of code. The function outputs "Hello world!". To call the function, just write its name with a open and close parathesis for example: writeMsg():

<?php

function writeMsg() {

echo "Hello world!";

}

writeMsg();

?>

## Parameters/Arguments

Parameters (also called Arguments) are variables that exist only within that function. They are provided by the programmer when the function is called and the function can read and change them locally. To pass a function parameter(s), add the values or variable in the parenthesis when calling the function for example: print\_two\_strings("Hello", "World");.The function must also be defined with the parameters declared within the function parentheses. For example: **function** print\_two\_strings($var1, $var2) { … }. Where $var1 and $var2 are new variables that only exist inside the function, who values get assigned by the values passed in from when the function is called for example print\_two\_srings(“StringOne”, “StringTwo”);. The variable $var1 is assigned the value “StringOne” and $var2 is assigned the value “StringTwo”.

When declaring or calling a function that has more than one parameter, you need to separate between different parameters with a comma ','.

A function declaration can look like this:

**function** print\_two\_strings($var1, $var2) {

**echo** $var1;

**echo** "**\n**";

**echo** $var2;

**return** **NULL**;

}

To call this function, you must give the parameters ($var1 and $var) a value. It doesn't matter what the value is, as long as there is one. By calling the function print\_two\_strings("Hello", "World"); with “Hello” and “World” separated by a comma, we are giving the parameter variable $var1 the values “Hello” and the parameter variable $var2 the value “World”:

print\_two\_strings("Hello", "World");

Output:

Hello

World

When declaring a function, you sometimes want to have the freedom not to use all the parameters. Therefore, PHP allows you to give them default values when declaring the function:

**function** print\_two\_strings($var1 = "Hello World", $var2 = "I'm Learning PHP") {

**echo**($var1);

**echo**("**\n**");

**echo**($var2);

}

These values will only be used, if the function call does not include enough parameters. If there is only one parameter provided, then $var2 = "I'm Learning PHP":

print\_two\_strings("Hello");

Output:

Hello

I'm Learning PHP

Another way to have a dynamic number of parameters is to use PHP's built-in func\_num\_args, func\_get\_args, func\_get\_arg and count functions.

**function** mean() {

$sum = 0;

$param\_count = func\_num\_args();

**for** ($i = 0; $i < $param\_count; $i++) {

$sum += func\_get\_arg($i);

}

$mean = $sum/$param\_count;

**echo** "Mean: **{**$mean**}**";

**return** **NULL**;

}

or

**function** mean() {

$sum = 0;

//initialized array $vars with all the parameter/argument values

$vars = func\_get\_args();

/\*count() is a built in function that provides the number of values stored in $vars array.\*/

**for** ($i = 0; $i < count($vars); $i++) {

$sum += $vars[$i];

}

$mean = $sum/count($vars);

**echo** "Mean: **{**$mean**}**";

**return** **NULL**;

}

The above functions would calculate the arithmetic mean of all of the values passed to them and output it. The difference is that the first function uses func\_num\_args and func\_get\_arg, while the second uses func\_get\_args to load the parameters into an array. The output for both of them would be the same. For example:

mean(35, 43, 3);

Output:

Mean: 27

In the examples above, notice how the code is written. The function name is written, followed by parenthesis (open and closed) and any possible parameters in the parenthesis. On the same line, is the open curly braces which is the start of the function block of code that will run when the function is called. On the next line the block of code that is executed when the function is called is written. After the last line of code, there is a closed curly brace. Each line of code inside the open and closed curly braces are indented.

function hereIsMyFunctionName($parameter1, $parameter2) {

echo "<p>$Company</p>";

}

# How to call a function?

Calling a function means causing a particular function to run at a particular point in the script. A function will never run if it is never called. A function must be called for the function block of code to be executed. A function can be called any number of times, there is no limit. When you pass arguments to a function, the value of each argument is assigned to the value of the corresponding formal parameter in the function definition.

Here is an example of the print() function and different ways a function can be called. The basic ways to call a function include:

* Calling the function to write on a new line with the string “I am human, I am”

**print**('I am human, I am.');

* Calling the print () function to write on a new line inside a control statement

**if** ($a == 72) {

**print**('I am human, I am.');

}

* Assigning the returned value of a function to a variable "$var = function()" . More details on return statements in the next section.

$result = sum ($a, 5);

* Calling the count() function inside the argument parentheses (expression) of a while condition statement

**while** ($i < count($one)) {

//while loop block of code

}

In our earlier examples we have called several functions. Most commonly we have called the function print() to print text to the output. The parameter for echo() has been the string we wanted printed for example print("Hello World!")prints "Hello World!" to the output. Now what if we want the function to not print() or echo() output but return back a value or information. If the function returns some information, we can assign it to a variable with a simple assignment operator "=":

$var1 = func\_name();

# Returning a value

It is commonly to have functions return some information. You can return a value from a function to a calling statement by assigning the calling statement to a variable. Example: $valueCalculated = sum(1,2,3); // sum() function is called and returned value is assigned to the variable $valueCalculated.

Generally, there are two reasons why a programmer would want information from a function:

1. The function does tasks such as calculations, and we need the result.
2. A function can return a value to indicate, if the function encountered any errors.

To return a value from a function use the return() statement in the function.

**function** add\_numbers($var1 = 0, $var2 = 0, $var3 = 0) {

$var4 = $var1 + $var2 + $var3;

**return** $var4;

}

Example PHP script:

**function** add\_numbers($var1 = 0, $var2 = 0, $var3 = 0) {

$var4 = $var1 + $var2 + $var3;

**return** $var4;

}

$sum = add\_numbers(1, 6, 9); //$var1 = 1, $var2 = 6 , $var3 = 9

**echo** "The result of 1 + 6 + 9 is **“ .** $sum **. “.”**;

Result:

The result of 1 + 6 + 9 is 16

return() statement ends the function's course.

Notice how if anything appears in a function declaration after the return() statement is executed, it is parsed but never executed. This can come in handy in some cases. For example:

**function** divide ($dividee, $divider) {

**if** ($divider == 0) {

*// Can't divide by 0.*

**return** **false**;

}

$result = $dividee/$divider;

**return** $result;

}

Notice that there is no else after the if. This is due to the fact that, if $divider does equal 0, the return() statement is executed and the function stops. There can be an unlimted number of return() statement within a single function, but only one return statement will ever be executed each time a function is called.

If you want to return multiple variables, you need to return an array rather than a single variable. For example:

**function** maths ($input1, $input2) {

$total = ($input1 + $input2);

$difference = ($input1 - $input2);

$return = **array**($total, $difference);

**return** $return;

}

When calling this from your script, you need to call it into an array. For example:

$return = maths(10, 5);

In this case $return[0] will be the total (e.g. 15), while $return[1] will be the difference (5).

# Variable Scope

When you the programmer start using functions, the location in your code a variable is declared becomes important. You have to be aware of the variable **scope** in the program. A variable can either have a **local** or **global** scope. A local scope variable is declared in a function block of code and it will only exist while the function is running. It cannot be called outside the function it was declared in. If a call to a local scope variable is made outside the function it was declared in, an error message will be generated. A global scope variable is a variable declared outside the function, in the main section of your code.

Global Scope Sample Code:

<?php  
$x = 5; // global scope  
   
function myTest() {  
    // using x inside this function will generate an error  
    echo "<p>Variable x inside function is: $x</p>";  
}   
myTest();  
  
echo "<p>Variable x outside function is: $x</p>";  
?>

Result:

Variable x inside function is:

Variable x outside function is: 5

Local Scope Sample Code:

<?php  
function myTest() {  
    $x = 5; // local scope  
    echo "<p>Variable x inside function is: $x</p>";  
}   
myTest();  
  
// using x outside the function will generate an error  
echo "<p>Variable x outside function is: $x</p>";  
?>

Result:

Variable x inside function is: 5

Variable x outside function is:

# Global Variables

In some programming languages global variables are available in every part of the code, but in PHP a global variable must be declared as a global variable. The global keyword can be used to access a global variable from within a function definition. To declare a global variable, the **global** keyword must be used. Using the keyword global in front of the variable name inside a function can access the variable that was declared outside the function. The global variable declaration statement only needs to include the global keyword along with the name of the variable. For example: global $variableName;.

<?php  
$x = 5;  
$y = 10;  
  
function myTest() {  
    global $x, $y;  
    $y = $x + $y;  
}   
  
myTest();  // run function  
echo $y; // output the new value for variable $y  
?>

Result:

15

# PHP also stores all global variables in an array called $GLOBALS[*index*]. The *index* holds the name of the variable. This array is also accessible from within functions and can be used to update global variables directly.

# PHP include() and require() Files

## include() & require()

The include() and require() statement allows code from one PHP file be included into another PHP file, as if all PHP code is running from a single PHP file. This allow developers to create function libraries that can be used and reused by many PHP scripts. This method of programming is highly effective technic for saving time. No need to rewrite code that already exist, you can simple store the block of code or library of functions in a separate file and use the include() or require() function calls to include it. The include file is commonly saved with a prefix of inc\_ to distinguish it as an include file, as opposed to a complete .php file. An extension of .php is still required so it will be processed by the PHP scripting engine.

The basic syntax of the include() and require() statement can be as seen below where filename.php can be a single PHP filename in the sample directory or a path to a PHP filename to include:

**include** "file.php";

**require** "file.php";

They both perform essentially the same function, but have one major difference: include will only throw a warning if there is a problem during the include process; require, however, will halt execution of the code. Therefore, a script’s dependencies will often be called with require since a script cannot execute fully without it’s dependencies.

A common example of using include() and require() is including the header, footer and menu used across every webpage on a website in the PHP include files. Then each webpage will simply include() the header, footer and menu code using a single include() statement to include the files. Plus, there is the added advantage of having identical header, footers and menus on each webpage of the website.

The include files can also be used to hide sensitive data. By storing the include files outside the server directory accessible by the web services, it is not directly accessible to website visitors. Because the path provided to the include() and require() statements are on the server itself and not limited to the directory the website is stored in on the web server, it is possible to store the include files anywhere on the server. For example, your web folder on the class web server is /home/username/public\_html/. Any file and directory in /home/username/public\_html is browsable.

# Include Once

Additionally, there exist many code libraries, class definitions, and variable declarations that you will want to separate into an include file, but that should only be called into the current script once. To ensure that these libraries are only included once, PHP includes the include\_once() and require\_once()functions.

Each time one of these functions is called, the PHP parser remembers which file it has called. If another include\_once()or require\_once attempts to load the same file, the parser will simply skip the command. It will produce no error or warning, it will simply act as though the command had executed successfully. This is because, in fact, it has.

IMPORTANT: If you include a file once with include\_once() and then later using include(), the file will be included a second time. If a file is included using include\_once() and a call to the same file is made by require\_once(), it will not be included again. include\_once() and require\_once() have the same 'memory,' as it were.

Sample Code:

Filename: footer.php Note: PHP built in date() function called

<?php  
echo "<p>Copyright &copy; 1999-" . date("Y") . " smcclass.com</p>";  
?>

Filename: main.php

<!DOCTYPE html>

<html>

<body>

<h1>Welcome to my home page!</h1>

<p>Some text.</p>

<p>Some more text.</p>

<?php include 'footer.php';?> //include footer.php code here

</body>

</html>

Result:

**Welcome to my home page!**

Some text.

Some more text.

Copyright © 1999-2018 smcclass.com