Function in PHP

Functions

A function is a named block of code that performs a specific task, possibly acting upon a set of values given to it, or parameters , and possibly returning a single value. Functions provide a way to eliminate repeating the same lines of code over and over in your programs Functions save on compile time no matter how many times you call them, functions are compiled only once for the page

There are hundreds of built-in functions in PHP. For example, `print_r` is a function that prints readable information about a variable in plain English rather than code. If given a string, integer, or float, the value itself is printed with the `print_r` function. If given an array, values are shown as keys and elements. A similar format is used for objects.

Functions in a PHP program can be built-in (or, by being in an extension, effectively built-in) or user-defined. Regardless of their source, all functions are evaluated in the same way:

```
Syntax

Function function_name([argument_list…])
{
[statements]
[return return_value;]
}
```

The number of parameters a function requires differs from function to function (and, as we'll see later, may even vary for the same function). The parameters supplied to the function may be any valid expression and should be in the specific order expected by the function.

When you create a function, you first need to give it a name, like *myFunction.* It's with this function name that you will be able to call upon your function, so make it easy to type and understand. Fo example

```
<?php
function myFunction()
{
}
?>
```

You can write any php code inside the function like:

```
<?php
myFunction();
function myFunction()
{
    echo "This is from function!<br />";
}
?>
```

You have to call the function in the main program.

myFunction();

It is defined as

function myFunction()

followed by the statements inside the functions within brackets as

```
{
    echo "This is from function!<br />";
}
```

Lets explain with a simple example

```
<?php
echo "Welcome to  xxx.com<br />";
echo "Well, thanks for visiting our site! <br />";
echo "and remember... <br />";
?>
```

Now the above code can be modified with functions as

```
<?php
function myFunction(){
    echo "Welcome to my website!<br />";
}
echo "Welcome to xxx.com <br />";
myFunction();
echo "Well, thanks for visiting our site! <br />";
echo "and remember... <br />";
myFunction();
?>
```

Output

```
Welcome to xxx.com
Welcome to my website!
Well, thanks for visiting our site!
and remember...
Welcome to my website!
```

Now you can see wherever you call the function with its name the function codes are executed.

Function with parameters

W can pass data to the functions so the functions can operate them. This can be done by passing parameters to the functions. A parameter appears within the parentheses "( )" and looks just like a normal PHP variable. Let's create a new function that creates a custom greeting based off of a person's name.

Our parameter will be the person's name and our function will concatenate this name onto a greeting string. Here's what the code would look like.

```
<?php
function welcome($firstName){
    echo "Hello there ". $firstName . "!<br />";
}
welcome("Ram");
welcome("Rani");
welcome("Hari");
welcome("David");
?>
```

Output

```
Hello there Ram!
Hello there Rani!
Hello there Hari!
Hello there David!
```

```
It is also possible to have multiple parameters in a function. To separate
multiple parameters PHP uses a comma ",". Let's modify our function to also
include last names
<?php
function welcome($firstName,$lastName){
    echo "Hello there ". $firstName . " ",$lastName."!<br />";
}
welcome("Ram",”Prasad”);
welcome("Rani",”ragav”);
welcome("Hari",”ram”);
welcome("David",”britto”);
?>
```

Output

```
Hello there Ram Prasad!
Hello there Rani ragav!
Hello there Hari ram!
Hello there David britto!
```

Php functions – returning values

You can return values to main program from functions. To capture the value in main program it has to be allocated to a variable like this:

$myVar = somefunction();

Let's demonstrate this returning of a value by using a simple function that returns the sum of two integers.

```
<?php
function SumFunction($X, $Y)
{
    $total = $X + $Y;
```

```
    return $total;
}

$ans = 0;
echo "Before the function, ans = ". $ans ."<br />";
$ans = SumFunction(7,8); // Store the result of SumFunction in $ans
echo "After the function, ans = " . $ans ."<br />";
?>
```

Output

Before the function, ans = 0
After the function, ans = 15

When we first print out the value of $ans it is still set to the original value of 0. However, when we set $ans equal to the function *SumFunction*, $ans is set equal to SumFunction's result. In this case, the result was 7 + 8 = 15, which was successfully stored into $ans and displayed in the second echo statement!

Parameters can also contain default values. With a default value, you actually don't need to pass the function any input for it to set the default

Example: without default parameter

Function display($greeting,$message)
{
Echo $greeting;
Echo $message;
}

When you call this from main program as

Display(“welcome”)

You will get warning message as missing argument as php expects 2 parameters to be sent where only one is supplied. This can be fixed by using default parameter like this

Example: with default parameter

Function display($greeting,$message = ”friends”)
{
Echo $greeting;
Echo $message;
}
Now when you call the function as
display(“welcome”);

no error or warning comes as even if you don’t supply any parameter in calling function as you have defined the default parameter inside the function definition it takes these values for the missing parameter.

Passing arrays to functions

You can pass arrays as parameter to functions like strings and numbers.

Example

```
<?php
 $scores = array(65, 32, 78, 98, 66);

 averager($scores);

 function averager($array)
 {
  $total = 0;
  foreach ($array as $value)
 {
    $total += $value;
  }

  if(count($array) > 0)
 {
   echo "The average was ", $total/count($array);
  }
 Else
  {
   echo "No elements to average!";
  }
 }
?>
```

In the above program the statement

```
 $scores = array(65, 32, 78, 98, 66);
```

Declares the variable $scores to be an array. In the next statement

```
averager($scores);
```

The function with the name averager has $scores array as input argument.

```
 function averager($array)
 {
  $total = 0;
  foreach ($array as $value)
  {
    $total += $value;
  }
```

```
  if(count($array) > 0)
 {
   echo "The average was ", $total/count($array);
  }
Else
 {
   echo "No elements to average!";
  }
}
```

In the above function the values in the array are added and result is placed in $total variable. Then the average of the array value is calculated and printed.

Returning the values of array to main program

```
<?php
  $data = create_array(3);
  echo "Here's the first array:<br>";
  print_r($data);
  echo "<br>";
  $data_2 = create_array(4);
  echo "Here's the second array:<br>";
  print_r($data_2);
  function create_array($number)
  {
    for ($i = 0; $i < $number; $i++){
      $x[] = $i;
    }
    return $x;
  }
?>
```

In the above program create_array() function is used to create array and add values into the array. The array is returned to the main program. The array values are printed in the main program.

$data = create_array(3);
In this line the create_array(3) is called and returning value is stored in $data. The value stored is printed with print_r() function. Another array is created using the same array as

$data_2 = create_array(4);

Parameter References

When you pass an argument to the function, a local copy is made in the function to store the value. Any changes made to that value don't affect the source of the parameter. You can define parameters that modify the source variable by defining reference parameters.

Reference parameters define references by placing an ampersand (`&`) directly before the parameter in the functions definition.

```
<html>
  <head>
    <title>
      Passing data to functions by value
    </title>
  </head>
  <body>
    <h1>
      Passing data to functions by value
    </h1>
    <?php
      $value = 10;
      echo "Before the call, \$value holds $value <br>";
      addNumber($value);
      echo "After the call, \$value holds $value <br>";

      function addNumber($number)
      {
        $number = $number+5;
      }
    ?>
  </body>
</html>
```

Output:

Passing data to functions by reference

Before the call, $value holds 10
After the call, $value holds 10

In the above coding the first echo statement displays the value of $value. Then it is passed as parameter like
addNumber($value);

In the function the value is added with 5.

Again in the main program there is another echo statement displaying the $value. The same value is printed. This is because of the scope of the parameters within the function. Once the function completes its execution whatever the operations are done on the local parameter inside the functions they become undefined outside. As the parameters are passed only as copy of them the values modified inside a function will have no effect outside. That is why you see again the same value after the function call in the above program.

See the below program where the **reference variable** is used

```
<html>
  <head>
    <title>
      Passing data to functions by reference
    </title>
  </head>
  <body>
    <h1>
      Passing data to functions by reference
    </h1>
    <?php
      $value = 10;
      echo "Before the call, \$value holds $value <br>";
      addNumber($value);
      echo "After the call, \$value holds $value <br>";

      function addNumber(&$number)
      {
        $number = $number+5;
      }
    ?>
  </body>
</html>
```

Output:

Passing data to functions by reference

Before the call, $value holds 10
After the call, $value holds 15

When you pass an argument by reference, that gives the code in the function direct access to that argument back in the calling code. So in the above program after the function is called with argument with the character(&) before the parameter it becomes a reference variable as shown

function addNumber(**&**$number)

now after the execution of the above function the variable $value is directly manipulated beyond its scope and you can see the change in the main program.

Accessing Global data with global variables

The global keyword

```
<?php
$a = 2;
$b = 2;
$c = 0;
function multiply()
{
    global $a, $b;

    $c = $a * $b;
}

multiply();
echo $c;
?>
```

The above script will output 4. By declaring *$a , $b and $c* global within the function, all references to these variable will refer to the global version. There is no limit to the number of global variables that can be manipulated by a function.

A second way to access variables from the global scope is to use the special PHP-defined *$GLOBALS* array. The previous example can be rewritten as:

```
<?php
$a = 2;
$b = 2;
$c = 0;
function multiply()
{
    $GLOBALS['c'] = $GLOBALS['a'] + $GLOBALS['b'];
}

multiply();
echo $c;
?>
```

Using *static* variables

Another important feature of variable scoping is the *static* variable. A static variable exists only in a local function scope, but it does not lose its value when program execution leaves this scope. Consider the following example:

```
<?php
```

```php
function test()
{
    $a = 0;
    echo $a;
    $a++;
}
?>
```

Output :
0

This function is quite useless since every time it is called it sets *$a* to *0* and prints *0*. The *$a*++ which increments the variable serves no purpose since as soon as the function exits the *$a* variable disappears. To make a useful counting function which will not lose track of the current count, the *$a* variable is declared static:

```php
<?php
function test()
{
    static $a = 0;
    echo $a;
    $a++;
}
?>
```

Now, *$a* is initialized only in first call of function and every time the *test()* function is called it will print the value of *$a* and increment it.

Static variables also provide one way to deal with recursive functions. A recursive function is one which calls itself. Care must be taken when writing a recursive function because it is possible to make it recurse indefinitely. You must make sure you have an adequate way of terminating the recursion. The following simple function recursively counts to 10, using the static variable *$count* to know when to stop:

```php
<?php
function test()
{
    static $count = 0;

    $count++;
    echo $count;
    if ($count < 10) {
        test();
    }
    $count--;
}
?>
```

Static variables may be declared as seen in the examples above. Trying to assign values to these variables which are the result of expressions will cause a parse error.

Including and Requiring PHP Files

You can insert the content of one PHP file into another PHP file before the server executes it, with the include() or require() function.

The two functions are identical in every way, except how they handle errors:

- include() generates a warning, but the script will continue execution
- require() generates a fatal error, and the script will stop

These two functions are used to create functions, headers, footers, or elements that will be reused on multiple pages.

To make your code more readable, you can place your functions in a separate file. PHP provides four functions that enable you to insert code from other files.

- `include`
- `require`
- `include_once`
- `require_once`

  All the `include` and `require` functions can take a local file or URL as input, but they cannot import a remote file. `require` and `include` functions are pretty similar in their functionality except for the way in which they handle an irretrievable resource. For example, `include` and `include_once` provide a warning if the resource cannot be retrieved and try to continue execution of the program. The `require` and `require_once` functions provide stop processing of the particular page if they can't retrieve the resource. Now we're going to get more specific about these four functions.

The include Statement

The `include` statement allows you to include and attach other PHP scripts to your own script. You can think of it as simply taking the included file and inserting them into your PHP file. A

A sample file called **add.php**

```
<?php
function add( $x, $y )
{
  return $x + $y;
```

```
}
?>
```

assumes that add.php is in the same directory as the program using the `include` function.

Lets us include the above add.php as in **includeadd.php** as

```
<?php
include('add.php');
echo add(2, 2);
?>
```

When executed, this produces:

```
4
```

As seen above the `include` statement attaches other PHP scripts so that you can access other variables, functions, and classes.

The include_once statement

A problem may arise when you include many nested PHP scripts, because the `include` statement doesn't check for scripts that have already been included.

For example, if you did this:

```
<?php
include('add.php');
include('add.php');
echo add(2, 2);
?>
```

You'd get this error:

```
Fatal error: Cannot redeclare add() (previously declared in
/home/www/public_html/includeeg/add.php:2) in
/home/www/public_html/includeeg/add.php on line 2
```

The above directory may not be where your file is located; your file will go wherever you've designated a place for it. To avoid this type of error, you should use the `include_once` statement.

Using include_once to include a file

```
<?php
include_once('add.php');
include_once('add.php');
echo add(2, 2);
```

```
?>
```

This outputs the following when executed:

```
4
```

Obviously, you're not going to place the same `include` statements right next to each other, but it's far more likely that you may include a file, which includes another file you've already included. You should always use `include_once`, as there really isn't any drawback to using it instead of `include`.

require and require_once functions

To make sure that a file is included and to stop your program, immediately use `require` and its counter part, `require_once`. These are exactly the same as `include` and `include_once` except that they make sure that the file is present; otherwise, the PHP script's execution is halted, which wouldn't be a good thing! Examples of when you should use `require` instead of `include` are if the file you're including defines critical functions that your script won't be able to execute or variable definitions such as database connection details.

For example, if you attempt to require a file that doesn't exist:

```
<?php
require_once('add_wrong.php');
echo add(2, 2);
?>
```

you'd get this error:

```
Warning: main(add_wrong.php): failed to open stream: No such
file or directory in
/home/www/public_html/requireeg/require_once.php on line 2

Fatal error: main(): Failed opening required 'add_wrong.php'
(include_path='.:/usr/share/php:/usr/share/pear') in
/home/www/pulic_html/requireeg/require_once.php on line 2
```

All file paths are contingent on where your files are located.