PHP Functions

PHP Built-in Functions

A function is a self-contained block of code that performs a specific task.

PHP has a huge collection of internal or built-in functions that you can call directly within your PHP scripts to perform a specific task, like `gettype()`, `print_r()`, `var_dump`, etc.

Please check out PHP reference section for a complete list of useful PHP built-in functions.

PHP User-Defined Functions

In addition to the built-in functions, PHP also allows you to define your own functions. It is a way to create reusable code packages that perform specific tasks and can be kept and maintained separately form main program. Here are some advantages of using functions:

- **Functions reduces the repetition of code within a program** — Function allows you to extract commonly used block of code into a single component. Now you can perform the same task by calling this function wherever you want within your script without having to copy and paste the same block of code again and again.
- **Functions makes the code much easier to maintain** — Since a function created once can be used many times, so any changes made inside a function automatically implemented at all the places without touching the several files.
- **Functions makes it easier to eliminate the errors** — When the program is subdivided into functions, if any error occur you know exactly what function causing the error and where to find it. Therefore, fixing errors becomes much easier.

- **Functions can be reused in other application** — Because a function is separated from the rest of the script, it's easy to reuse the same function in other applications just by including the php file containing those functions.

The following section will show you how easily you can define your own function in PHP.

EX

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>PHP Function</title>
</head>
<body>

<?php
// Defining function
function whatIsToday(){
    echo "Today is " . date('l', mktime());
}
// Calling function
whatIsToday();
?>

</body>
</html>
```

Functions with Parameters

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>PHP Function with Parameters</title>
</head>
<body>

<?php
// Defining function
function getSum($num1, $num2){
  $sum = $num1 + $num2;
  echo "Sum of the two numbers $num1 and $num2 is : $sum";
```

```
}

// Calling function
getSum(10, 20);
?>

</body>
</html>
```

Functions with Optional Parameters and Default Values

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>PHP Function with Optional Parameters</title>
</head>
<body>

<?php
// Defining function
function customFont($font, $size=1.5){
    echo "<p style=\"font-family: $font; font-size: {$size}em;\">Hello, world!</p>";
}

// Calling function
customFont("Arial", 2);
customFont("Times", 3);
customFont("Courier");
?>

</body>
</html>
```

Returning Values from a Function

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>PHP Function Returning Values</title>
</head>
```

```
<body>

<?php
// Defining function
function getSum($num1, $num2){
    $total = $num1 + $num2;
    return $total;
}

// Printing returned value
echo getSum(5, 10); // Outputs: 15
?>

</body>
</html>
```

A function can not return multiple values. However, you can obtain similar results by returning an array, as demonstrated in the following example.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>PHP Returning Array with Functions</title>
</head>
<body>

<?php
// Defining function
function divideNumbers($dividend, $divisor){
    $quotient = $dividend / $divisor;
    $array = array($dividend, $divisor, $quotient);
    return $array;
}

// Assign variables as if they were an array
list($dividend, $divisor, $quotient) = divideNumbers(10, 2);
echo $dividend . "<br>";  // Outputs: 10
echo $divisor  . "<br>";   // Outputs: 2
echo $quotient . "<br>";  // Outputs: 5
?>

</body>
</html>
```

Passing Arguments to a Function by Reference

In PHP there are two ways you can pass arguments to a function: *by value* and *by reference*. By default, function arguments are passed by value so that if the value of the argument within the function is changed, it does not get affected outside of the function. However, to allow a function to modify its arguments, they must be passed by reference.

Passing an argument by reference is done by prepending an ampersand (`&`) to the argument name in the function definition, as shown in the example below:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>PHP Passing Arguments by Reference</title>
</head>
<body>

<?php
/* Defining a function that multiply a number
by itself and return the new value */
function selfMultiply(&$number){
    $number *= $number;
    return $number;
}

$mynum = 5;
echo $mynum . "<br>"; // Outputs: 5

selfMultiply($mynum);
echo $mynum . "<br>"; // Outputs: 25
?>

</body>
</html>
```

Understanding the Variable Scope

However, you can declare the variables anywhere in a PHP script. But, the location of the declaration determines the extent of a variable's visibility within the PHP program i.e. where the variable can be used or accessed. This accessibility is known as *variable scope*.

By default, variables declared within a function are local and they cannot be viewed or manipulated from outside of that function, as demonstrated in the example below:

```
<?php
// Defining function
function test(){
    $greet = "Hello World!";
    echo $greet;
}

test(); // Outputs: Hello World!

echo $greet; // Generate undefined variable error
?>

But
<?php
$greet = "Hello World!";

// Defining function
function test(){
    echo $greet;
}

test();  // Generate undefined variable error

echo $greet; // Outputs: Hello World!
?>
```

The global Keyword

There may be a situation when you need to import a variable from the main program into a function, or vice versa. In such cases, you can use the `global` keyword before the variables inside a function. This keyword turns the variable

into a global variable, making it visible or accessible both inside and outside the function, as show in the example below:

```php
<?php
$greet = "Hello World!";

// Defining function
function test(){
    global $greet;
    echo $greet;
}

test(); // Outpus: Hello World!
echo $greet; // Outpus: Hello World!

// Assign a new value to variable
$greet = "Goodbye";

test(); // Outputs: Goodbye
echo $greet; // Outputs: Goodbye
?>
```

PHP is a Loosely Typed Language

In the example above, notice that we did not have to tell PHP which data type the variable is.

PHP automatically associates a data type to the variable, depending on its value. Since the data types are not set in a strict sense, you can do things like adding a string to an integer without causing an error.

In PHP 7, type declarations were added. This gives us an option to specify the expected data type when declaring a function, and by adding the `strict` declaration, it will throw a "Fatal Error" if the data type mismatches.

In the following example we try to send both a number and a string to the function without using `strict`:

```
<?php

function addNumbers(int $a, int $b) {

  return $a + $b;

}

echo addNumbers(5, "5 days");

// since strict is NOT enabled "5 days" is changed to int(5), and it will return 10

?>
```

To specify `strict` we need to set `declare(strict_types=1);`. This must be on the very first line of the PHP file.

In the following example we try to send both a number and a string to the function, but here we have added the `strict` declaration:

```
<?php declare(strict_types=1); // strict requirement
```

```
function addNumbers(int $a, int $b) {
  return $a + $b;
}
echo addNumbers(5, "5 days");
// since strict is enabled and "5 days" is not an integer, an error will be thrown
?>
```

Example - dynamical function

```
<?php
  function write($text)
  {
    print($text);
  }
```

```
  function writeBold($text)
  {
    print("<b>$text</b>");
  }

  $myFunction = "write";
  $myFunction("Hello!");
  print("<br>\n");

  $myFunction = "writeBold";
  $myFunction("Goodbye!");
  print("<br>\n");
?>
```

Creating Recursive Functions

A recursive function is a function that calls itself again and again until a condition is satisfied. Recursive functions are often used to solve complex mathematical

calculations, or to process deeply nested structures e.g., printing all the elements of a deeply nested array.

The following example demonstrates how a recursive function works.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>PHP Recursive Function</title>
</head>
<body>

<?php
// Defining recursive function
function printValues($arr) {
    global $count;
    global $items;

    // Check input is an array
    if(!is_array($arr)){
        die("ERROR: Input is not an array");
    }

    /*
    Loop through array, if value is itself an array recursively
call the function,
    else add the value found to the output items array,
    and increment counter by 1 for each value found
    */
    foreach($arr as $a){
        if(is_array($a)){
            printValues($a);
        } else{
            $items[] = $a;
            $count++;
        }
    }

    // Return total count and values found in array
    return array('total' => $count, 'values' => $items);
}
```

```
// Define nested array
$species = array(
    "birds" => array(
        "Eagle",
        "Parrot",
        "Swan"
    ),
    "mammals" => array(
        "Human",
        "cat" => array(
            "Lion",
            "Tiger",
            "Jaguar"
        ),
        "Elephant",
        "Monkey"
    ),
    "reptiles" => array(
        "snake" => array(
            "Cobra" => array(
                "King Cobra",
                "Egyptian cobra"
            ),
            "Viper",
            "Anaconda"
        ),
        "Crocodile",
        "Dinosaur" => array(
            "T-rex",
            "Alamosaurus"
        )
    )
);

// Count and print values in nested array
$result = printValues($species);
echo $result['total'] . ' value(s) found: ';
echo implode(', ', $result['values']);
?>

</body>
```

```
</html>
```