UNIT-IV:

PHP Programming: Introducing PHP: Creating PHP script, Running PHP script.

Working with variables and constants: Using variables, Using constants, Data types, Operators.

Controlling program flow: Conditional statements, Control statements, Arrays, functions.

Working with forms and Databases such as MySQL.

PHP Programming

Introduction:

⇒ PHP (recursive acronym for PHP: Hypertext Preprocessor) is a widely-used open source general-purpose server-side scripting language that is embedded in HTML.

⇒ PHP scripts are executed on the Web Server and the result is sent to the web browser as plain HTML.

⇒ PHP has a relatively simple architecture compared to other MVC based web frameworks (Python, Ruby, node.js, etc.).

⇒ PHP is usually purely interpreted, as is the case with JavaScript. Recent PHP implementations perform some pre-compilation, at least on complex scripts, which increases the speed of interpretation.

⇒ The syntax and semantics of PHP are closely related to the syntax and semantics of JavaScript.

⇒ PHP uses dynamic typing, as does JavaScript.

⇒ PHP was developed by Rasmus Lerdorf, a member of the Apache Group in 1994.

⇒ Its initial purpose was to provide a tool to help Lerdorf track visitors to his personal Web site.

⇒ In 1995 he developed a package called Personal Home Page Tools, which became the first publicly distributed version of PHP.

⇒ The current major version of PHP is 7.

What can PHP do?

⇒ PHP is used to generate and manage dynamic content, databases, session tracking, and even build an entire e-commerce site. There are three main areas where PHP scripts are used.

⇒ **Server-side scripting:** This is the most traditional and main target field for PHP. You need three things to make this work: the PHP parser (CGI or server module), a web server and a web browser. You need to run the web server, with a connected PHP installation. You can access the PHP program output with a web browser, viewing the PHP page through the server. All these can run on your home machine if you are just experimenting with PHP programming.

⇒ **Command line scripting:** You can make a PHP script to run it without any server or browser. You only need the PHP parser to use it this way. This type of usage is ideal for scripts regularly executed using cron (on *nix or Linux) or Task Scheduler (on Windows). These scripts can also be used for simple text processing tasks.

⇒ **Writing desktop applications:** PHP is probably not the very best language to create a desktop application with a graphical user interface, but if you know PHP very well, and would like to usesome advanced PHP features in your client-side applications you can also use PHP-GTK to write such programs. You also have the ability to write cross-platform applications this way. PHP-GTK is an extension to PHP, not available in the main distribution.

⇒ With PHP, you have the freedom of choosing an operating system and a web server.
  - PHP can be used on all major operating systems, including Linux, many UNIX variants (including HP-UX, Solaris and OpenBSD), Microsoft Windows, macOS, RISC OS, and probably others.
  - PHP also has support for most of the web servers today. This includes Apache, IIS, and many others. And this includes any web server that can utilize the FastCGI PHP binary, like lighttpd and nginx. PHP works as either a module, or as a CGI processor.

⇒ Furthermore, you also have the choice of using procedural programming or object oriented programming (OOP), or a mixture of them both.

⇒ With PHP you are not limited to output HTML. PHP's ability includes outputting images, PDF files and even Flash movies (using libswf and Ming) generated on the fly. You can also output easily any text, such as XHTML and any other XML file. PHP can autogenerate these files, and save them in the file system, instead of printing it out, forming a server-side cache for your dynamic content.

⇒ PHP can be integrated with the number of popular databases using specific extensions, including MySQL, PostgreSQL, Oracle, Microsoft SQL Server, Sybase, and so on or using an abstraction layer like PDO, or connect to any database supporting the Open Database Connection standard via the ODBC extension. Other databases may utilize cURL or sockets, like CouchDB.

⇒ PHP also has support for talking to other services using protocols such as LDAP, IMAP, SNMP, NNTP, POP3, HTTP, COM (on Windows) and countless others. You can also open raw network sockets and interact using any other protocol. PHP has support for the WDDX complex data exchange between virtually all Web programming languages. Talking about interconnection, PHP has support for instantiation of Java objects and using them transparently as PHP objects.

⇒ PHP has useful text processing features, which include the Perl compatible regular expressions (PCRE), and many extensions and tools to parse and access XML documents. PHP standardizes all of the XML extensions on the solid base of libxml2, and extends the feature set adding SimpleXML, XMLReader, and XMLWriter support.

⇒ Some of the frame woks that support PHP are: Laravel, Phalcon, CodeIgniter, Symfony, Cakephp, Zend, FuelPHP, Slim, Phpixie, Aura, Yii 2, …

Basic Syntax:

⇒ PHP scripts either are embedded in markup documents or are in files that arereferenced by such documents. PHP code is embedded in documents by enclosingit between the <?php and ?> tags.

Example:

```
<?php
        echo "Hello, world!";
?>
```

Note: Every PHP statement end with a semicolon (;) - this tells the PHP engine that the end of the current statement has been reached.

⇒ If a PHP script is stored in a different file, it can be brought into a documentwith the include or require construct, which takes the file name as its string parameter.

```
include("path/to/filename"); OR include "path/to/filename";

require("path/to/filename"); OR require "path/to/filename";
```

Example:

```
<?php
        include("header.php");
        require(“header.php”);
?>
```

PHP Comments:

⇒ A comment is simply text that is ignored by the PHP engine.
⇒ The purpose of comments is to make the code more readable. It may help other developer (or you in the future when you edit the source code) to understand what you were trying to do with the PHP.
⇒ PHP supports single-line as well as multi-line comments.
⇒ To write a single-line comment either start the line with either two slashes (//) or a hash symbol (#).

Example:

```
<?php
// This is a single line comment
# This is also a single line comment
echo "Hello, world!";
?>
```

⇒ To write multi-line comments, start the comment with a slash followed by an asterisk (/*) and end the comment with an asterisk followed by a slash (*/).

Example:

```
<?php
/* This is a multiple line comment block
that spans across more than one line */
echo "Hello, world!";
?>
```

Your first PHP-enabled page:

⇒ Create a file named **hello.php** and put it in your web server's root directory (DOCUMENT_ROOT) with the following content:

```
<html>
<head>
<title>PHP Test</title>
```

```
</head>
<body>
<?php
        echo '<p>Hello World</p>';
?>
</body>
</html>
```

⇒ Use your browser to access the file with your web server's URL, ending with the */hello.php* file reference.

⇒ When developing locally this URL will be something like

    http://localhost/hello.php
        OR
    http://127.0.0.1/hello.php

⇒ If everything is configured correctly, this file will be parsed by PHP and the following output will be sent to your browser:

```
<html>
<head>
<title>PHP Test</title>
</head>
<body>
<p>Hello World</p>
</body>
</html>
```

⇒ All it does is display: Hello World using the PHP **echo** statement.

Working with variables and constants:

Variables:

⇒ Variables in PHP are represented by a dollar sign ($) followed by the name of the variable.

⇒ The variable name is case-sensitive.

⇒ A valid variable name starts with a letter or underscore, followed by any number of letters, numbers, or underscores.

⇒ As a regular expression, it would be expressed thus: ^[a-zA-Z_\x80-\xff][a-zA-Z0-9_\x80-\xff]*$

⇒ By default, variables are always assigned by value.

⇒ PHP also offers another way to assign values to variables: assign by reference means the new variable simply references (in other words, "becomes an alias for" or "points to") the original variable. Changes to the new variable affect the original, and vice versa.

⇒ To assign by reference, simply prepend an ampersand (&) to the beginning of the variable which is being assigned (the source variable).

Example:

```
<?php
```

```
$college = 'ST.MARY\'s';              // Assign the value 'ST.MARY's' to $foo
$place = 'Budampadu Chebrolu';        // Assign the value 'Budampadu, Chebrolu' to $place
$clgfullname = "My name is $college, located at $place";
echo $clgfullname      // outputs My name is ST.MARY's,  located at  Budampadu, Chebrolu
?>
```

Example:

```
<?php
// Reference exampe
$college = "ST.MARY's";        // Assign the value ST.MARY's to $foo
$refcollege = & $college;      // Reference $college via $refcollege
$refcollege = "My Name is ST.MARY's Women's";
echo $college;                 // outputs My name is ST.MARY's Women's
echo $refcollege;              // outputs My name is ST.MARY's Women's
?>
```

Removing a reference:

⇒ You delete a reference using the unset() function. When you unset a reference, you're merely removing that reference, not the value that it references.
⇒ The value remains in memory until you unset all references to it, including the original variable.

Example:

```
<?php
$college = "ST.MARY's";        // Assign the value ST.MARY's to $foo
$refcollege = & $college;      // Reference $college via $refcollege
$refcollege = "My Name is ST.MARY's Women's";
unset($refcollege);
echo $college;                 // displays My name is ST.MARY's Women's
?>
```

Note: It is not necessary to initialize variables in PHP however it is a very good practice. Uninitialized variables have a default value of their type depending on the context in which they are used - booleans default to **FALSE**, integers and floats default to zero, strings (e.g. used in echo) are set as an empty string and arrays become to an empty array.

Constants:

⇒ A constant is an identifier (name) for a simple/fixed value. As the name suggests, that value cannot change during the execution of the script (except for **magic constants**, which aren't actually constants).
⇒ A constant is case-sensitive by default. By convention, constant identifiers are always uppercase.
⇒ Constants are very useful for storing data that doesn't change while the script is running. Common examples of such data include configuration settings such as database username and password, website's base URL, company name, etc.

⇒ Constants are defined using PHP's **define()** function, which accepts two arguments: the name of the constant, and its value. Once defined the constant value can be accessed at any time just by referring to its name.

⇒ Name of constants must follow the same rules as variable names.

Note: By convention, constant names are usually written in uppercase letters. This is for their easy identification and differentiation from variables in the source code.

Example:

```
<!DOCTYPE html>
<html lang="en">
<head>
<title>PHP Constants</title>
</head>
<body>
<?php
// Defining constant
define("SITE_URL", "https://www.stmarysgroup.com/");
// Using constant
echo 'Thank you for visiting - ' . SITE_URL;
?>
</body>
</html>
```

Data types:

⇒ The type of a variable is not usually set by the programmer; rather, it is decided at runtime by PHP depending on the context in which that variable is used.

⇒ The values assigned to a PHP variable may be of different data types including simple string and numeric types to more complex data types like arrays and objects.

⇒ PHP supports ten primitive types.
- o Four scalar types: boolean, integer, float (floating-point number, double), string
- o Four compound types: array, object, callable, iterable
- o And finally two special types: resource, NULL

Boolean:

⇒ Booleans are like a switch it has only two possible values either 1 (true) or 0 (false).

Example:

```
<?php
        $show_error = true;             // Assign the value TRUE to a variable
        var_dump($show_error);
?>
```

Integers: Integers are whole numbers, without a decimal point (..., -2, -1, 0, 1, 2, ...).

⇒ Integers can be specified in decimal (base 10), hexadecimal (base 16 - prefixed with 0x) or octal (base 8 - prefixed with 0) notation, optionally preceded by a sign (- or +).

Note: Since PHP 5.4+ you can also specify integers in binary (base 2) notation. To use binary notation precede the number with 0b (e.g. $var = 0b11111111;).

Example:

```
<?php
$a = 123;           // decimal number
var_dump($a);
echo "<br>";
$b = -123;          // a negative number
var_dump($b);
echo "<br>";
$c = 0x1A;          // hexadecimal number
var_dump($c);
echo "<br>";
$d = 0123;          // octal number
var_dump($d);
?>
```

Floating Point Numbers or Doubles:

⇒ Floating point numbers (also known as "floats", "doubles", or "real numbers") are decimal or fractional numbers.

Example:

```
<?php
$a = 1.234;
var_dump($a);
echo "<br>";
$b = 10.2e3;
var_dump($b);
echo "<br>";
$c = 4E-10;
var_dump($c);
?>
```

Strings:

⇒ Strings are sequences of characters, where every character is the same as a byte.
⇒ A string can hold letters, numbers, and special characters and it can be as large as up to 2GB (2147483647 bytes maximum).
⇒ The simplest way to specify a string is to enclose it in single quotes (e.g. 'Hello world!'), however you can also use double quotes ("Hello world!").

Example:

```
<?php
$a = 'Hello world!';
echo $a;
echo "<br>";
$b = "Hello world!";
echo $b;
echo "<br>";
$c = 'Stay here, I\'ll be back.';
echo $c;
?>
```

PHP Arrays:

⇒ An array is a variable that can hold more than one value at a time. It is useful to aggregate a series of related items together, for example a set of country or city names.
⇒ An array is formally defined as an indexed collection of data values.
⇒ Each index (also known as the key) of an array is unique and references a corresponding value.

Example:

```
<?php
$colors = array("Red", "Green", "Blue");
var_dump($colors);
echo "<br>";
$color_codes = array("Red" => "#ff0000", "Green" => "#00ff00", "Blue" => "#0000ff");
var_dump($color_codes);
?>
```

PHP Objects:

⇒ An object is a data type that not only allows storing data but also information on, how to process that data.
⇒ An object is a specific instance of a class which serves as templates for objects.
⇒ Objects are created based on this template via the new keyword.
⇒ Every object has properties and methods corresponding to those of its parent class.
⇒ Every object instance is completely independent, with its own properties and methods, and can thus be manipulated independently of other objects of the same class.

Example:

```
<?php
// Class definition
class greeting
{
// properties
public $str = "Hello World!";
// methods
function show_greeting()
{
return $this->str;
}
}          // End of class declaration
// Create object from class
$message = new greeting;
var_dump($message);
?>
```

PHP NULL:

⇒ The special NULL value is used to represent empty variables in PHP. A variable of type NULL is a variable without any data. NULL is the only possible value of type null.

Example:

```
<?php
$a = NULL;
var_dump($a);
echo "<br>";
$b = "Hello World!";
$b = NULL;
var_dump($b);
?>
```

Note: When a variable is created without a value in PHP like $var; it is automatically assigned a value of null. Many novice PHP developers mistakenly considered both $var1 = NULL; and $var2 = ""; are same, but this is not true. Both variables are different — the $var1 has null value while $var2 indicates no value assigned to it.

PHP Resources:

⇒ A resource is a special variable, holding a reference to an external resource. Resource variables typically hold special handlers to opened files and database connections.

Example:

```
<?php
// Open a file for reading
$handle = fopen("note.txt", "r");
var_dump($handle);
echo "<br>";
// Connect to MySQL database server with default setting
$link = mysql_connect("localhost", "root", "");
```

```
var_dump($link);
?>
```

Type Conversion in PHP:

⇒ When you declare a variable in PHP, you do not specify its type. Its type depends on the value it currently holds. For example, the following variable has been assigned a string value. So it's type is currently string, as we verify by using the gettype() function:

```
$val = 'Hello there!';   // assign string
echo gettype($val);     // output – string
```

⇒ You can change a variable's type by assigning a new value of a different type. We assign an integer to the variable declared above and use the gettype function again to see the result:

```
$val = 21;                  // now assign integer
echo gettype($val);                 // output – integer
```

Automatic Type Conversion:

⇒ A variable's type may vary depending upon the context in which it is used. Some functions and operators may expect a value of a particular type and automatically convert to that type, or try to.

⇒ For example, echo and print expect strings. A conditional expression expects a boolean.

⇒ An attempt to convert may result in an error message when a particular type cannot be successfully converted to the expected type. Let's see what happens when we attempt to use echo to display a variable that currently holds an array value:

```
$val = array('one', 'two', 'three');
echo $val;
// Notice: Array to string conversion in [file] on line [line number] - Array
```

⇒ The echo function is not able to successfully convert an array to a string. All it can do is issue a notice and then display Array. However, you can display the contents of arrays using the var_dump() function.

Number to String Conversion: The concatenation operator expects its arguments to be strings.

Example:

```
<?php
$val = 20; // integer
var_dump('I will be ready in ' . $val . ' minutes.');
?>
```

Output:string(30) "I will be ready in 20 minutes."

The numeric value is seamlessly incorporated into the string.

Boolean to String Conversion:

⇒ In boolean to string conversion, true is converted to '1', while false is converted to an empty string. The following verifies this by concatenating boolean values into a string:

```
var_dump('How many? ' . true);        // Output: string(11) "How many? 1"
var_dump('How many? ' . false);       // Output: string(10) "How many? "
```

Notice that **false** does not add to the length of the string since it is converted to an empty string.

Comparing Strings with Numbers: When you compare a string with a number using operators, the string is converted to a number. Consider the following example:

```
// compare number to string that begins with number
if ( 123 == '123 Go' )
{
   echo 'equal';
}
else
{
   echo 'not equal';
}
```

Output: equal

⇒ Perhaps that result seems surprising. The two values are equal (not identical) because when PHP converts a string to a number, the starting portion of the string is used if it is numeric. If not, the string converts to 0.

⇒ Contrast the above result with what happens when you compare a string with a number using a string comparison function. In this case, the number is converted to a string:

```
// compare number with string using strcmp
echo strcmp(123, '123 Go');            // < 0
```

⇒ The two are not equal. The number is less than the string.

Determining a Variable's Type:

⇒ Before passing a variable to a string function, you might want to determine whether it really is a string.

⇒ PHP's is_string() function returns true or false to indicate whether the value passed to it is a string or not:

Example:

```
<?php
$val = '44';
if ( is_string($val) )
{
   echo 'string';
}
else
{
   echo 'not a string';
}
?>
```

Output: string

Note: PHP also provides functions for determining other types, including: is_int(), is_array(), is_bool(), is_float(), and is_object().

settype() Function:

⇒ PHP's settype() function is used to change the type of a variable.
⇒ For example, suppose you have a variable that is currently assigned a number and you would like to convert it to a string.

Example:

```
<?php
$val = 24;                        // integer
settype($val, 'string');          // set $val type to string
// check type and value of $val
var_dump($val);
?>
```

Output: string(2) "24"

⇒ The variable will retain that type unless you either change the value of the variable to another type, or use settype() again to assign another type to it.
⇒ You can use the settype() function to set the following types in addition to string: integer, float, boolean, a, rrayobject, and null.

Type Conversion Functions:

⇒ PHP provides several functions that can be used to convert a value to a specific type. For example, the **strval()** function can be used to convert a value to a string.

Example:

```
<?php
$val = 88;                        // integer
// pass $val to strval and check return value with var_dump
var_dump( strval($val) );         // string(2) "88"
// check type and value of $val
var_dump($val);                   // int(88)
?>
```

⇒ Notice that although we can see that the strval function has returned a string value, when we check the value of the variable again, we see it is not a string but an integer. In other words, the strval() function returns a converted value but does not change the type of the variable itself.
⇒ Related functions are available for converting to other types as well: intval(), floatval(), and boolval() (for converting to integers, floating point numbers, and booleans).

Type Casting:

⇒ You can also use type casting to change the type of a variable. In order to cast a variable (or value) to another type, specify the desired type in parentheses immediately before the variable you wish to convert. The following demonstrates casting a boolean to a string:

Example:

```
<?php
$val = true;                    // boolean
// cast $val to string and check return value with var_dump
var_dump( (string)$val );       // string(1) "1"
// check type and value of $val
var_dump($val);                 // bool(true)
?>
```

⇒ Notice that casting does not change the type of the variable. It only returns the converted value and makes no lasting change, just as the conversion functions like strval().

⇒ In addition to (string), the following casts are also supported by PHP: (int), (bool), (float), (array), and (object).

Operators:

⇒ An operator is something that takes one or more values (or expressions, in programming jargon) and yields another value (so that the construction itself becomes an expression).

⇒ Operators can be grouped according to the number of values they take.
- o Unary operators take only one value, for example ! (Thelogical not operator) or *++* (the increment operator).
- o Binary operators take two values, such as the familiar arithmetical operators+ (plus) and - (minus), and the majority of PHP operators fall into this category.
- o Finally, there is a single ternary operator, *? :*, which takes three values; this is usually referred to simply as "the ternary operator" (although it could perhaps more properly be called the conditional operator).

Arithmetic Operators:

⇒ The arithmetic operators are used to perform common arithmetical operations, such as addition, subtraction, multiplication etc. Here's a complete list of PHP's arithmetic operators:

Operator	Description	Example	Result
+	Addition	$x + $y	Sum of $x and $y
-	Subtraction	$x - $y	Difference of $x and $y.
*	Multiplication	$x * $y	Product of $x and $y.
/	Division	$x / $y	Quotient of $x and $y
%	Modulus	$x % $y	Remainder of $x divided by $y

Example:

```
<?php
$x = 10;
$y = 4;
echo($x + $y); // 0utputs: 14
echo($x - $y); // 0utputs: 6
echo($x * $y); // 0utputs: 40
echo($x / $y); // 0utputs: 2.5
echo($x % $y); // 0utputs: 2
?>
```

Assignment Operators:

⇒ The basic assignment operator is "=". The assignment operators are used to assign values to variables. In addition to the basic assignment operator, there are "combined operators" for all of the binary arithmetic, array union and string operators that allow you to use a value in an expression and then set its value to the result of that expression.

⇒ Note that the assignment copies the original variable to the new one (assignment by value), so changes to one will not affect the other.

Operator	Description	Example	Is The Same As
=	Assign	$x = $y	$x = $y
+=	Add and assign	$x += $y	$x = $x + $y
-=	Subtract and assign	$x -= $y	$x = $x - $y
*=	Multiply and assign	$x *= $y	$x = $x * $y
/=	Divide and assign quotient	$x /= $y	$x = $x / $y
%=	Divide and assign modulus	$x %= $y	$x = $x % $y

Example:

```
<?php
$x = 10;
echo $x;              // Outputs: 10
$x = 20;
$x += 30;
echo $x;              // Outputs: 50
$x = 50;
$x -= 20;
echo $x;              // Outputs: 30
$x = 5;
$x *= 25;
echo $x;              // Outputs: 125
$x = 50;
$x /= 10;
echo $x;              // Outputs: 5
$x = 100;
$x %= 15;
echo $x;              // Outputs: 10
$a = ($b = 4) + 5;    // $a is equal to 9 now, and $b has been set to 4.
?>
```

Comparison Operators:

⇒ The comparison operators are used to compare two values in a Boolean fashion.

Operator	Name	Example	Result
==	Equal	$x == $y	True if $x is equal to $y
===	Identical	$x === $y	True if $x is equal to $y, and they are of the same type
!=	Not equal	$x != $y	True if $x is not equal to $y
<>	Not equal	$x <> $y	True if $x is not equal to $y
!==	Not identical	$x !== $y	True if $x is not equal to $y, or they are not of the same type
<	Less than	$x < $y	True if $x is less than $y
>	Greater than	$x > $y	True if $x is greater than $y
>=	Greater than or equal to	$x >= $y	True if $x is greater than or equal to $y
<=	Less than or equal to	$x <= $y	True if $x is less than or equal to $y

Example:

```
<?php
$x = 25;
$y = 35;
$z = "25";
var_dump($x == $z);  // Outputs: boolean true
var_dump($x === $z); // Outputs: boolean false
var_dump($x != $y);  // Outputs: boolean true
var_dump($x !== $z); // Outputs: boolean true
var_dump($x < $y);   // Outputs: boolean true
var_dump($x > $y);   // Outputs: boolean false
var_dump($x <= $y);  // Outputs: boolean true
var_dump($x >= $y);  // Outputs: boolean false
?>
```

Incrementing (++) and Decrementing (--) Operators:

⇒ PHP supports C-style pre- and post-increment and decrement operators.The increment/decrement operators only affect numbers and strings. Arrays, objects, booleans, and resources are not affected. Decrementing NULL values has no affect too, but incrementing them results in 1.

Example:

```
<?php
echo "<h3>Postincrement</h3>";
$a = 5;
echo "Should be 5: " . $a++ . "<br />\n";
echo "Should be 6: " . $a . "<br />\n";
echo "<h3>Preincrement</h3>";
$a = 5;
echo "Should be 6: " . ++$a . "<br />\n";
echo "Should be 6: " . $a . "<br />\n";
echo "<h3>Postdecrement</h3>";
$a = 5;
echo "Should be 5: " . $a-- . "<br />\n";
echo "Should be 4: " . $a . "<br />\n";
echo "<h3>Predecrement</h3>";
$a = 5;
echo "Should be 4: " . --$a . "<br />\n";
echo "Should be 4: " . $a . "<br />\n";
?>
```

Note: PHP follows Perl's convention when dealing with arithmetic operations on character variables and not C's. For example, in PHP and Perl $a = 'Z'; $a++;turns $a into 'AA', while in C a = 'Z'; a++; turns a into '[' (ASCII value of *'Z'* is 90, ASCII value *of* '[' is 91). Note that character variables can be incremented but not decremented and even so only plain ASCII alphabets and digits (a-z, A-Z and 0-9) are supported. Incrementing/decrementing other character variables has no effect, the original string is unchanged.

Logical Operators:

⇒ The logical operators are typically used to combine conditional statements.

Operator	**Name**	**Example**	**Result**
and	And	$x and $y	True if both $x and $y are true
or	Or	$x or $y	True if either $x or $y is true
xor	Xor	$x xor $y	True if either $x or $y is true, but not both
&&	And	$x && $y	True if both $x and $y are true
||	Or	$x || $y	True if either $$x or $y is true
!	Not	!$x	True if $x is not true

String Operators:

⇒ There are two operators which are specifically designed for strings.

Operator	Description	Example	Result
.	Concatenation	$str1 . $str2	Concatenation of $str1 and $str2
.=	Concatenation assignment	$str1 .= $str2	Appends the $str2 to the $str1

Example:

```
<?php
$x = "Hello";
$y = " World!";
echo $x . $y; // Outputs: Hello World!
$x .= $y;
echo $x; // Outputs: Hello World!     ?>
```

Array Operators:

Operator	Name	Example	Result
+	Union	$x + $y	Union of $x and $y
==	Equality	$x == $y	True if $x and $y have the same key/value pairs
===	Identity	$x === $y	True if $x and $y have the same key/value pairs in the same order and of the same types
!=	Inequality	$x != $y	True if $x is not equal to $y
<>	Inequality	$x <> $y	True if $x is not equal to $y
!==	Non-identity	$x !== $y	True if $x is not identical to

Example:

```
<?php
$x = array("a" => "Red", "b" => "Green", "c" => "Blue");
$y = array("u" => "Yellow", "v" => "Orange", "w" => "Pink");
$z = $x + $y;                        // Union of $x and $y
var_dump($z);
var_dump($x == $y);                  // Outputs: boolean false
var_dump($x === $y);                 // Outputs: boolean false
var_dump($x != $y);                  // Outputs: boolean true
var_dump($x <> $y);                  // Outputs: boolean true
var_dump($x !== $y);                    // Outputs: boolean true
?>
```

Bitwise operators:

⇒ Bitwise operators allow evaluation and manipulation of specific bits within an **integer**.

Example	Name	Result
$a & $b	And	Bits that are set in both $a and $b are set.
$a | $b	Or (inclusive or)	Bits that are set in either $a or $b are set.
$a ^ $b	Xor (exclusive or)	Bits that are set in $a or $b but not both are set.
~ $a	Not	Bits that are set in $a are not set, and vice versa.
$a << $b	Shift left	Shift the bits of ab steps to the left (each step means "multiply by two")
$a >> $b	Shift right	Shift the bits of ab steps to the right (each step means "divide by two")

⇒ Bit shifting in PHP is arithmetic.
⇒ Bits shifted off either end are discarded.
- Left shifts have zeros shifted in on the right while the sign bit is shifted out on the left, meaning the sign of an operand is not preserved.
- Right shifts have copies of the sign bit shifted in on the left, meaning the sign of an operand is preserved.

⇒ Use parentheses to ensure the desired precedence.
- For example, $a & $b == true evaluates the equivalency then the bitwise and; while ($a & $b) == true evaluates the bitwise and then the equivalency.

⇒ If both operands for &, | and ^ operators are strings, then the operation will be performed on the ASCII values of the characters that make up the strings and the result will be a string. In all other cases, both operands will be converted to integers and the result will be an integer.
⇒ If the operand for the ~ operator is a string, the operation will be performed on the ASCII values of the characters that make up the string and the result will be a string, otherwise the operand and the result will be treated as integers.
⇒ Both operands and the result for the << and >> operators are always treated as integers.

Error control operators:

⇒ PHP supports one error control operator: at sign (@).
- When prepended to an expression in PHP, any error messages that might be generated by that expression will be ignored.

⇒ If you have set a custom error handler function with **set_error_handler(),**then it will still get called, but this custom error handler can (and should) call**error_reporting()** which will return 0 when the call that triggered the error was preceded by an @.

⇒ If the **track_errors** feature is enabled, any error message generated by the expression will be saved in the variable**$php_errormsg**. This variable will be overwritten on each error, so check early if you want to use it.

Example:

```
/* Intentional file error */
$my_file = @file ('non_existent_file') or
     die ("Failed opening file: error was '$php_errormsg'");
// this works for any expression, not just functions:
$value = @$cache[$key];
// will not issue a notice if the index $key doesn't exist.
?>
```

Note: The @-operator works only on expressions. A simple rule of thumb is: if you can take the value of something, you can prepend @ operator to it.

⇒ For instance, you can prepend it to variables, function and include calls, constants, and so forth. You cannot prepend it to function or class definitions, or conditional structures such as if and foreach, and so forth.

Type operators:

⇒ **instanceof** is used to determine whether a PHP variable is an instantiated object of a certain class.

Note: The **instanceof** operator was introduced in PHP 5. Before this time is_a() was used but is_a() has since been deprecated in favor of **instanceof.** Note that as of PHP 5.3.0, is_a() is no longer deprecated.

Operator precedence:

⇒ The precedence of an operator specifies how "tightly" it binds two expressions together.
  - For example, in the expression 1 + 5 * 3, the answer is 16 and not 18 because the multiplication ("*") operator has a higher precedence than the addition ("+") operator.

⇒ Parentheses may be used to force precedence, if necessary.
  - For instance: (1 + 5) * 3 evaluates to 18.

⇒ When operators have equal precedence their associativity decides how the operators are grouped.
  - For example "-" is left-associative, so 1 - 2 - 3 is grouped as (1 - 2) - 3 and evaluates to -4. "=" on the other hand is right-associative, so $a = $b = $c is grouped as $a = ($b = $c).

⇒ Operators of equal precedence that are non-associative cannot be used next to each other.
  - For example 1 < 2 > 1 is illegal in PHP. The expression 1 <= 1 == 1 on the other hand is legal, because the == operator has lesser precedence than the <= operator.

⇒ Use of parentheses, even when not strictly necessary, can often increase readability of the code by making grouping explicit rather than relying on the implicit operator precedence and associativity.

⇒ The following table lists the operators in order of precedence, with the highest-precedence ones at the top. Operators on the same line have equal precedence, in which case associativity decides grouping.

Operator Precedence		
Associativity	**Operators**	**Additional Information**
non-associative	clonenew	clone and new
left	[	array()
right	**	arithmetic
right	++--~(int)(float)(string)(array)(object)(bool)@	types and increment/decrement
non-associative	instanceof	types
right	!	logical
left	*/%	arithmetic
left	+-.	arithmetic and string
left	<<>>	bitwise
non-associative	<<=>>=	comparison
non-associative	==!====!==<><=>	comparison
left	&	bitwise and references
left	^	bitwise
left	|	bitwise
left	&&	logical
left	||	logical
right	??	comparison
left	? :	ternary
right	=+=-=*=**=/=.=%=&=|=^=<<=>>=	assignment
right	yield from	yield from
right	yield	yield
left	and	logical
left	xor	logical
left	or	logical

Controlling program flow:

⇒ Any PHP script is built out of a series of statements.
⇒ A statement can be an assignment, a function call, a loop, a conditional statement or even a statement that does nothing (an empty statement).
⇒ Statements usually end with a semicolon.
⇒ In addition, statements can be grouped into a statement-group by encapsulating a group of statements with curly braces.
⇒ A statement-group is a statement by itself as well. The various statement types are described below.

Conditional statements:

⇒ Like most programming languages, PHP also allows you to write code that perform different actions based on the results of a logical or comparative test conditions at run time. This means, you can create test conditions in the form of expressions that evaluates to either true or false and based on these results you can perform certain actions.
⇒ PHP also has the IF, SWITCH … CASE statement(s).

if Statement: The **if**statement is used to execute a block of code only if the specified condition evaluates to true. This is the simplest PHP's conditional statements and can be written like:

```
if(expression)
{
   // Code to be executed
}
```

⇒ The following example will output "Have a nice weekend!" if the current day is Friday:

```
<?php
$d = date("D");
if($d == "Fri")
{
   echo "Have a nice weekend!";
}
?>
```

if...else Statement: You can enhance the decision making process by providing an alternative choice through adding an else statement to the **if** statement. The**if...else** statement allows you to execute one block of code if the specified expression is evaluates to true and another block of code if it is evaluates to false. It can be written, like this:

```
if(condition)
{
// Code to be executed if condition is TRUE
}
else
```

```
{
// Code to be executed if condition is FALSE
}
```

⇒ The following example will output "Have a nice weekend!" if the current day is Friday, otherwise it will output "Have a nice day!"

```
<?php
$d = date("D");
if($d == "Fri")
{
   echo "Have a nice weekend!";
}
else
{
   echo "Have a nice day!";
}
?>
```

if...else if...else Statement: Theif...else if...elsea special statement that is used to combine multiple if...elsestatements.

```
If(expression1)
{
// Code to be executed if condition1 is TRUE
}
else if (expression2)
{
// Code to be executed if the condition1 is false and condition2 is TRUE
}
else
{
// Code to be executed if both condition1 and condition2 are FALSE
}
```

⇒ The following example will output "Have a nice weekend!" if the current day is Friday, and "Have a nice Sunday!" if the current day is Sunday, otherwise it will output "Have a nice day!"

```
<?php
$d = date("D");
if($d == "Fri")
{
   echo "Have a nice weekend!";
}
else if($d == "Sun")
{
   echo "Have a nice Sunday!";
}
else
{
   echo "Have a nice day!";
}
?>
```

The Ternary Operator: The ternary operator provides a shorthand way of writing the if...else statements. The ternary operator is represented by the question mark (?) symbol and it takes three operands: an expression to check, a result for TRUE, and a result for FALSE.

Example:

```
<?php echo ($age < 18) ? 'Child' : 'Adult'; ?>
```

⇒ The ternary operator in the example above selects the value on the left of the colon (i.e. 'Child') if the expression evaluates to true (i.e. if $age is less than 18), and selects the value on the right of the colon (i.e. 'Adult') if the expression evaluates to false.

switch … case statement: The switch-case statement is an alternative to the if-else if-else statement, which does almost the same thing.

⇒ The switch-case statement tests a variable against a series of values until it finds a match, and then executes the block of code corresponding to that match.

```
switch (n)
{
case label1:
        // Code to be executed if n=label1
        break;
case label2:
        // Code to be executed if n=label2
        break;
...
default:
        // Code to be executed if n is different from all labels
}
```

⇒ Consider the following example, which display a different message for each day.

```
<?php
$today = date("D");
switch($today)
{
   case "Mon":
        echo "Today is Monday. Clean your house.";
        break;
   case "Tue":
        echo "Today is Tuesday. Buy some food.";
        break;
    case "Wed":
        echo "Today is Wednesday. Visit a doctor.";
        break;
    case "Thu":
        echo "Today is Thursday. Repair your car.";
        break;
    case "Fri":
        echo "Today is Friday. Party tonight.";
        break;
    case "Sat":
        echo "Today is Saturday. Its movie time.";
        break;
    case "Sun":
        echo "Today is Sunday. Do some rest.";
        break;
    default:
        echo "No information available for that day.";
        break;
}
?>
```

Note: The switch-case statement differs from the if-else if-else statement in one important way.

⇒ The switch statement executes line by line (i.e. statement by statement) and once PHP finds a case statement that evaluates to true, it's not only executes the code corresponding to that case statement, but also executes all the subsequent case statements till the end of the switch block automatically.
⇒ To prevent this add a break statement to the end of each case block.
⇒ The break statement tells PHP to break out of the switch-case statement block once it executes the code associated with the first true case.

Different Types of Looping structures in PHP

⇒ Loops are used to execute the same block of code again and again, until a certain condition is met.
⇒ The basic idea behind a loop is to automate the repetitive tasks within a program to save the time and effort.
⇒ PHP supports four different types of loops.
- **while** - loops through a block of code until the condition is evaluate to true.
- **do…while** - the block of code executed once and then condition is evaluated. If the condition is true the statement is repeated as long as the specified condition is true.
- **for** - loops through a block of code until the counter reaches a specified number.
- **for each** - loops through a block of code for each element in an array.

while loop: The while statement will loops through a block of code until the condition in the while statement evaluate to true.

```
while (expression)
{
// Code to be executed
}
```

⇒ The value of the expression is checked each time at the beginning of the loop, so even if this value changes during the execution of the nested statement(s), execution will not stop until the end of the iteration (each time PHP runs the statements in the loop is one iteration). Sometimes, if the **while**expression evaluates to FALSE from the very beginning, the nested statement(s) won't even be run once.

⇒ Alternate syntax:

```
while (expression):
        statement
        ...
endwhile;
```

Example:

```
<?php
$i = 1;
while ($i <= 10)
{
   echo $i++;  /* the printed value would be $i before the increment (post-increment) */
}
/* using alternate syntax */
$i = 1;
while ($i <= 10):
        echo $i;
        $i++;
endwhile;
?>
```

do … while loop: These loops are very similar to while loops, except the truth expression is checked at the end of each iteration instead of in the beginning. The main difference from regular while loops is that the first iteration of a do-while loop is guaranteed to run (the truth expression is only checked at the end of the iteration), whereas it may not necessarily run with a regular while loop (the truth expression is checked at the beginning of each iteration, if it evaluates to **FALSE** right from the beginning, the loop execution would end immediately).

⇒ There is just one syntax for do-while loops:

```
do
{
   // Code to be executed
}
while (expression);
```

Example:

```
<?php
$i = 1;
do
{
   echo "The number is " . $i . "<br>";     $i++;
}
while ($i <= 10);
?>
```

for Loop: for loops are the most complex loops in PHP. They behave like their C counterparts. The syntax of a for loop is:

```
for (expr1; expr2; expr3)
{
Statement(s)
}
```

⇒ The first expression (expr1) is evaluated (executed) once unconditionally at the beginning of the loop.
⇒ In the beginning of each iteration, expr2 is evaluated. If it evaluates to TRUE, the loop continues and the nested statement(s) are executed. If it evaluates to FALSE, the execution of the loop ends.
⇒ At the end of each iteration, expr3 is evaluated (executed).
⇒ Each of the expressions can be empty or contain multiple expressions separated by commas. In expr2, all expressions separated by a comma are evaluated but the result is taken from the last part. expr2 being empty means the loop should be run indefinitely (PHP implicitly considers it as TRUE, like C). This may not be as useless as you might think, since often you'd want to end the loop using a conditional break statement instead of using the for truth expression.

Example:

```
<?php
/* example 1 */
for ($i = 1; $i <= 10; $i++)
{
   echo $i;
}
/* example 2 */
for ($i = 1; ; $i++)
{
if ($i > 10)
{
break;
}
echo $i;
}
/* example 3 */
$i = 1;
for (; ; )
{
if ($i > 10)
{
break;
}
echo $i;
$i++;
}
/* example 4 */
for ($i = 1, $j = 0; $i <= 10; $j += $i, print $i, $i++);
?>
```

foreach Loop: The foreach construct provides an easy way to iterate over arrays. foreach works only on arrays and objects, and will issue an error when you try to use it on a variable with a different data type or an uninitialized variable. There are two syntaxes:

```
foreach (array_expression as $value)
{
Statement(s)
}
foreach (array_expression as $key => $value)
{
Statement(s)
}
```

⇒ The first form loops over the array given by array_expression. On each iteration, the value of the current element is assigned to $value and the internal array pointer is advanced by one (so on the next iteration, you'll be looking at the next element).

⇒ The second form will additionally assign the current element's key to the $key variable on each iteration.

Example:

```
<?php
$arr = array(1, 2, 3, 4);
foreach ($arr as &$value)
{
$value = $value * 2;
}
```

```
// $arr is now array(2, 4, 6, 8)
unset($value);          // break the reference with the last element
?>
```

break:break ends execution of the current for, foreach, while, do-while or switch structure.

⇒ break accepts an optional numeric argument which tells it how many nested enclosing structures are to be broken out of. The default value is 1, only the immediate enclosing structure is broken out of.

Continue: This is used within looping structures to skip the rest of the current loop iteration and continue execution at the condition evaluation and then the beginning of the next iteration.

⇒ continue accepts an optional numeric argument which tells it how many levels of enclosing loops it should skip to the end of. The default value is 1, thus skipping to the end of the current loop.

goto: The goto operator can be used to jump to another section in the program.

⇒ The target point is specified by a label followed by a colon, and the instruction is given as goto followed by the desired target label.
⇒ This is not a full unrestricted goto.
⇒ The target label must be within the same file and context, meaning that you cannot jump out of a function or method, nor can you jump into one.
⇒ You also cannot jump into any sort of loop or switch structure. You may jump out of these, and a common use is to use a goto in place of a multi-level break.

Example:

```
<?php
for($i=0,$j=50; $i<100; $i++)
{
while($j--)
{
   if($j==17) goto end;
}
}
echo "i = $i";
end:
echo 'j hit 17';
?>
```

⇒ The above code will produce **j hit 17.**

```
<?php
goto loop;
for($i=0,$j=50; $i<100; $i++)
{
while($j--)
{
```

```
loop:
}
}
echo "$i = $i";
?>
```

⇒ The above code will output **Fatal error: 'goto' into loop or switch statement is disallowed in script on line 2.**

Arrays:

⇒ Arrays in PHP are unlike those of any other common programming language.
⇒ They are best described as a combination of the arrays of a typical language and associative arrays, or hashes, found in some other languages, such as Ruby and Python.
⇒ An array in PHP is actually an ordered map.
⇒ A map is a type that associates values to keys.
- o If the array has a logical structure that is similar to an array in another language, the keys just happen to be non-negative integers and are always in ascending order (indexed array).
- o If the array has a logical structure that is similar to a hash, its keys are strings and the order of its elements is determined with a system-designed hashing function (associative array).

Note: PHP does not distinguish between indexed and associative arrays.

⇒ This type is optimized for several different uses; it can be treated as an array, list (vector), hash table (an implementation of a map), dictionary, collection, stack, queue, and probably more.
⇒ As array values can be other arrays, trees and multidimensional arrays are also possible.

Array creation:

⇒ There are two ways to create an array in PHP:using the array() language construct, and using short syntax ([]).

array() function to create an indexed array:

```
$colors = array("Red", "Green", "Blue");
```

⇒ The above is equivalent to the following example, in which indexes are assigned manually:

```
<?php
$colors[0] = "Red";
$colors[1] = "Green";
$colors[2] = "Blue";
?>
```

Note: In an indexed or numeric array, the indexes are automatically assigned and start with 0, and the values can be any data type.

short syntax ([]) to create an indexed array:

```
$countries = ["USA", "India", "China", "Australia", "UK"];
```

array() function to create an associative array:

```
$countrycodes = array("USA"=>1, "India"=>91, "China"=>86, "Australia"=>61,"UK"=>44);
```

⇒ The following example is equivalent to the previous example, but shows a different way of creating associative arrays.

```
<?php
$countrycodes["USA"]=1;
$countrycodes["India"]=91;
$countrycodes["China"]=86;
$countrycodes["Australia"]=61;
$countrycodes["UK"]=44;
?>
```

View Array Structure and Values:

⇒ You can see the structure and values of any array by using one of two functions -var_dump() or print_r().
⇒ The print_r(), however, gives somewhat less information.

Example:

```
<?php
// Define array
$cities = array("Hyderabad", "Banglore", "Chennai","Thiruvananthapuram");
// Display the cities
print_r($cities);
?>
```

Output:
Array ([0] => Hyderabad [1] => Banglore [2] => Chennai [3] => Thiruvananthapuram). This output shows the key and the value for each element in the array.

var_dum($cities)shows the data type of each element, such as a string of 9 characters, in addition to the key and value. Following is the output:

```
array(4) { [0]=> string(9) "Hyderabad" [1]=> string(8) "Banglore" [2]=> string(7) "Chennai" [3]=> string(18) "Thiruvananthapuram"}
```

⇒ You can access the individual elements of an array using square bracket syntax.

Example:

```
<?php
// Define array
$cities = array("Hyderabad", "Banglore", "Chennai","Thiruvananthapuram");
echo $cities[2];
?>
```

Produces the output: Chennai

Add elements to array:[] square Bracket Syntax:

⇒ When you want to add individual elements to the end of an array in PHP, square bracket syntax is the most efficient.
⇒ You can include an index (integer or string) in the square brackets, or leave them empty, in which case PHP will use the next available integer.

Example:

```
<?php
// Define array
$cities = array("Hyderabad", "Banglore", "Chennai","Thiruvananthapuram");
$cities[4]= "Delhi";
print_r($cities);
?>
```

array_push():

⇒ This function is used to add multiple elements at the end of an array.
⇒ Pass the array as the first argument followed by any number of elements in the order in which you would like them to be added.
⇒ This function returns the number of elements in the modified array.

Example:

```
<?php
// Define array
$cities = array("Hyderabad", "Banglore", "Chennai","Thiruvananthapuram");
echo "No. of elements before add:",count($cities);
// Add elements using array_push()
$ne = array_push($cities,"Delhi","Mumbai","Kolkata","Srinagar","Bhopal");
echo "<br>No. of elements after add:",count($cities);
echo "<br>The Array elements are: <br>";
print_r($cities);
?>
```

Output:

```
No. of elements before add:4
No. of elements after add:9
The Array elements are:
Array ( [0] => Hyderabad [1] => Banglore [2] => Chennai [3] => Thiruvananthapuram [4] => Delhi
[5] => Mumbai [6] => Kolkata [7] => Srinagar [8] => Bhopal )
```

array_unshift():

⇒ This function adds elements to the beginning of an array. As with array_push(), pass the array first, followed by any number of elements you would like to add to the array.

⇒ Arrays with numeric indexes have those indexes re-numbered starting from zero.

Example:

```
<?php
// Define array
$cities = array("Hyderabad", "Banglore", "Chennai","Thiruvananthapuram");
echo "No. of elements before add:",count($cities);
// Add elements using array_unshift()
$ne = array_unshift($cities,"Delhi","Mumbai");
echo "<br>No. of elements after add:",count($cities);
echo "<br>The Array elements are: <br>";
print_r($cities);
?>
```

Output:

```
No. of elements before add:4
No. of elements after add:6
The Array elements are:
Array ( [0] => Delhi [1] => Mumbai [2] => Hyderabad [3] => Banglore [4] => Chennai [5] =>
Thiruvananthapuram )
```

Remove Elements from Array:

⇒ Various functions to remove elements from arrays are array_pop(), array_shift(), and unset().

array_pop():

⇒ This function removes the last element from the array passed to it.
⇒ It returns that removed element and reduces the length of the array by one.

Example:

```
<?php
// Define array
$cities = array("Hyderabad", "Banglore", "Chennai","Thiruvananthapuram");
echo "No. of elements before remove:",count($cities);
echo "<br>The array elements are: ";
print_r($cities);
// Remove elements using array_pop()
$ne = array_pop($cities);
echo "<br>No. of elements after remove:",count($cities);
echo "<br>The Array elements are: <br>";
print_r($cities);
?>
```

Output:

```
No. of elements before remove: 4
```

```
The array elements are: Array ( [0] => Hyderabad [1] => Banglore [2] => Chennai [3] =>
Thiruvananthapuram )
No. of elements after remove: 3
The Array elements are:
Array ( [0] => Hyderabad [1] => Banglore [2] => Chennai )
```

array_shift():

⇒ This function is similar to array_pop() except that instead of removing the last element, it removes the first.

Example:

```
<?php
// Define array
$cities = array("Hyderabad", "Banglore", "Chennai","Thiruvananthapuram");
echo "No. of elements before remove:",count($cities);
echo "<br>The array elements are: ";
print_r($cities);
// Remove elements using array_pop()
$ne = array_shift($cities);
echo "<br>No. of elements after remove:",count($cities);
echo "<br>The Array elements are: <br>";
print_r($cities);
?>
```

Output:

```
No. of elements before remove: 4
The array elements are: Array ( [0] => Hyderabad [1] => Banglore [2] => Chennai [3] =>
Thiruvananthapuram )
No. of elements after remove: 3
The Array elements are:
Array ( [0] => Banglore [1] => Chennai [2] => Thiruvananthapuram )
```

unset()

⇒ This function can be used to remove individual elements from an array.

Example:

```
<?php
// Define array
$cities = array("Hyderabad", "Banglore", "Chennai","Thiruvananthapuram","Delhi","Mumbai");
echo "<br>Elements before remove:";
print_r($cities);
// Remove elements using unset()
unset($cities[2]);
echo "<br>Elements after remove:";
```

```
print_r($cities);
?>
```

Output:

```
Elements before remove: Array ( [0] => Hyderabad [1] => Banglore [2] => Chennai [3] =>
Thiruvananthapuram [4] => Delhi [5] => Mumbai )
Elements after remove: Array ( [0] => Hyderabad [1] => Banglore [3] => Thiruvananthapuram [4]
=> Delhi [5] => Mumbai )
```

Note: unset() function leaves a gap in numerically indexed array. Use the array_values() function to return a re-indexed copy.

array_splice():

⇒ This function can be used to add elements, remove elements, and/or replace elements anywhere in an array.
⇒ The first argument to array_splice() is the array you wish to modify.
⇒ The second argument is the offset, i.e., the location in the array where elements are to be added, removed, or replaced.
⇒ The (optional) third argument is the number of elements to remove from the array.
⇒ The fourth argument is an array of elements you wish to add.

Note:

⇒ Pass 0 (zero) as the third argument when you want to add elements to an array, but not to remove or replace.
⇒ This function supports a negative offset which allows you to specify a location starting from the end of the array.
⇒ The modified array is re-indexed starting from 0.

Example: Add Elements to an Array with array_splice()

```
<?php
// Define array
$cities = array("Hyderabad", "Banglore", "Chennai","Thiruvananthapuram");
echo "Array values before call to array_splice() <br>";
print_r($cities);
array_splice($cities,2,0,["Delhi","Mumbai"]);
echo "<br>Array values after call to array_splice() <br>";
print_r($cities);
?>
```

Output:

```
Array values before call to array_splice()
Array ( [0] => Hyderabad [1] => Banglore [2] => Chennai [3] => Thiruvananthapuram )
Array values after call to array_splice()
Array ( [0] => Hyderabad [1] => Banglore [2] => Delhi [3] => Mumbai [4] => Chennai [5] =>
Thiruvananthapuram )
```

Example: Remove Elements from an Array with array_splice()

```
<?php
// Define array
$cities = array("Hyderabad", "Banglore", "Chennai","Thiruvananthapuram","Delhi","Mumbai");
echo "Array values before call to array_splice() <br>";
print_r($cities);
array_splice($cities,3,2);
echo "<br>Array values after call to array_splice() <br>";
print_r($cities);
?>
```

Output:

```
Array values before call to array_splice()
Array ( [0] => Hyderabad [1] => Banglore [2] => Chennai [3] => Thiruvananthapuram [4] => Delhi
[5] => Mumbai )
Array values after call to array_splice()
Array ( [0] => Hyderabad [1] => Banglore [2] => Chennai [3] => Mumbai )
```

Note: If you leave out the third argument, the elements from the offset to the end of the array will be removed:

Example: Remove the last elements using negative offset

```
<?php
// Define array
$cities = array("Hyderabad", "Banglore", "Chennai","Thiruvananthapuram","Delhi","Mumbai");
echo "Array values before call to array_splice() <br>";
print_r($cities);
array_splice($cities, -2, 2 );
echo "<br>Array values after call to array_splice() <br>";
print_r($cities);
?>
```

Output:

```
Array values before call to array_splice()
Array ( [0] => Hyderabad [1] => Banglore [2] => Chennai [3] => Thiruvananthapuram [4] => Delhi
[5] => Mumbai )
Array values after call to array_splice()
Array ( [0] => Hyderabad [1] => Banglore [2] => Chennai [3] => Thiruvananthapuram )
```

Example: Replace Elements in an Array with array_splice()

```
<?php
// Define array
$cities = array("Hyderabad", "Banglore", "Chennai","Thiruvananthapuram","Delhi","Mumbai");
echo "Array values before call to array_splice() <br>";
print_r($cities);
```

```
array_splice($cities, 3, 1, ["Budampadu"] );
echo "<br>Array values after call to array_splice() <br>";
print_r($cities);
?>
```

Output:

```
Array values before call to array_splice()
Array ( [0] => Hyderabad [1] => Banglore [2] => Chennai [3] => Thiruvananthapuram [4] => Delhi [5] =>
Mumbai )
Array values after call to array_splice()
Array ( [0] => Hyderabad [1] => Banglore [2] => Chennai [3] => Budampadu [4] => Delhi [5] => Mumbai )
```

Traversing Arrays:

⇒ PHP provides several ways to traverse arrays using both array iteration functions and language constructs: array_walk(), array_map(), array_filter(), foreach, list/each, and for loops.

array_walk():

⇒ This function applies a callback function you specify to every element of the array passed to it. Each element's value and key are passed to the callback function in turn.
⇒ By default, the array_walk() function does not modify the array passed to it. However, the value can be passed by reference to change the values in the array.
⇒ The array_walk() function returns true or false depending on the success or failure of its invocation.

Example:

```
<?php
$pets = ['Morie', 'Miki', 'Halo', 'lab' => 'Winnie'];
// arguments: array, callback function, additional data (optional)
array_walk( $pets, function($val, $key) { echo "$key => $val \n"; } );
?>
```

Output:

```
0 => Morie 1 => Miki 2 => Halo lab => Winnie
```

array_map():

⇒ Pass a callback function and an array to array_map() and it returns a new array based on the return value of the callback for each iteration as the values of the original array are passed to it in turn.

Example:

```
$ar = [1, 2, 3, 4];
// arguments: callback function, array
$ar2 = array_map( function($v) { return $v*2; }, $ar );
print_r($ar2);
```

Output:Array ([0] => 2 [1] => 4 [2] => 6 [3] => 8)

Example:uses built in function“ucfirst”

```
<?php
$ar = [ 'jan' => 'january', 'feb' => 'february', 'mar' => 'march', 'apr' => 'april' ];
$ar2 = array_map('ucfirst', $ar);
print_r($ar2);
?>
```

array_filter():

⇒ This function creates a new array by using a callback function to screen elements of the array passed to it.
⇒ The callback function inspects each value of the passed array in turn and returns true to include that value in the resulting array. Notice that the array returned by the array_filter() function retains the keys of the function passed to it.
⇒ A flag, ARRAY_FILTER_USE_KEY or ARRAY_FILTER_USE_BOTH, can be passed as the third argument to array_filter() to use the key, or both value and key in the callback function.

Example:

```
<?php
$ar = [1, 'two', 'three', 4, 5, 'six'];
// arguments: array, callback function
$ar2 = array_filter( $ar, function($v) { return is_int($v); } );
print_r($ar2);
?>
```

Output:

```
Array ( [0] => 1 [3] => 4 [4] => 5 )
```

Example:

```
<?php
$ar = [1, 'two' => 2, 'three', 4 => 'four', 5];
$ar2 = array_filter( $ar, function($v, $k)
        {
        if ( is_int($v) && is_int($k) )
        {
       return true;
        }
        }, ARRAY_FILTER_USE_BOTH );
print_r($ar2);
?>
```

Output:Array ([0] => 1 [5] => 5)

⇒ If no callback argument is passed to the array_filter() function, array values that convert to false will be excluded from the returned array.

Example:

```
<?php
```

```
$ar = [0, 1, true, false, [], null, 'Yo'];
$ar2 = array_filter($ar);
var_dump($ar2)
?>
```

Output:

```
array(3) { [1]=> int(1) [2]=> bool(true) [6]=> string(2) "Yo" }
```

Search Arrays:

⇒ PHP provides several functions that can be used to search arrays, including array_search(), array_keys() (when passed a search value), in_array(), and array_key_exists().

array_search():

⇒ Pass a value and an array to the array_search() function and it will return the first key with a matching value. Or, if no match is found, it will return false.

Example:

```
<?php
$ar = ['apple', 'orange', 'pear', 'grape'];
// search for 'apple' in $ar
var_dump( array_search('pear', $ar) );
?>
```

Output:

```
int(2)
```

Example:

```
<?php
$food_colors = ['apple' => 'red','grape' => 'purple','tomato' => 'red','kale' => 'green'];
// search for value 'red' in $food_colors
var_dump( array_search('red', $food_colors) ); // string(5) "apple"
?>
```

Output:

```
string(5) "apple"
```

⇒ This function only returns the key for the first match.

⇒ The array_search() function provides an optional third argument to specify whether a strict comparison of values should be performed. When no third argument is passed, the default is false and type conversion will take place. This can result in false positives. Pass true as the third argument so no type conversion will take place when the values are compared.

⇒ The array_keys() and in_array() functions also provide arguments for strict comparison.

Example: array_keys()

```
<?php
$food_colors = [
   'apple' => 'red',
   'grape' => 'purple',
   'tomato' => 'red',
   'kale' => 'green'
];
// arguments: array, search value (optional)
print_r( array_keys($food_colors, 'red') );
?>
```

Output:

```
Array ( [0] => apple [1] => tomato )
```

Example: in_array()

```
<?php
$ar = ['Jon', 'Mark', 'Jess'];
// search for 'Jess' in $ar above
var_dump( in_array('Jess', $ar) ); // bool(true)
// search for 'jess' returns false
var_dump( in_array('jess', $ar) ); // bool(false)
?>
```

Output:

```
bool(true) bool(false)
```

Exmple: array_key_exists()

```
<?php
$person = array(
   'name' => 'Jon',
   'age' => 26,
   'marital status' => null,
   'friends' => array('Matt', 'Kaci', 'Jess')
);
var_dump( array_key_exists('marital status', $person) ); // bool(true)
?>
```

Output:

```
bool(true)
```

Sort Arrays:

⇒ PHP provides a variety of functions and options for sorting arrays.
⇒ The sort function includes options for specifying string or numeric sort, case insensitive sort, and/or or natural order sort. Plus there are functions designed to sort associative arrays, sort by key, sort according to the natural order algorithm, sort in reverse, and perform custom sorts with user-defined callback functions.
⇒ All of the sort functions change the order of the elements in the array passed to the function rather than returning a new array of sorted elements.
⇒ The sort functions return true or false depending upon success or failure of the invocation.

Options for Sort Functions:

⇒ Several of the sort functions provide a set of options that influence the sort order. These can be passed as the second argument to the functions that support them. For example:
  - SORT_STRING specifies the array will be sorted as strings.
  - SORT_NUMERIC specifies the array will be sorted as numbers.
  - SORT_NATURAL specifies the sort is according to the natural order algorithm.
  - SORT_FLAG_CASE (with SORT_STRING or SORT_NATURAL) specifies a case-insensitive sort.

Example:

```
<?php
$ar = ['football', 'Running', 'cycling', 'Soccer', 'basketball'];
sort($ar);
print_r($ar);
?>
```

Output:

```
Array ( [0] => Running [1] => Soccer [2] => basketball [3] => cycling [4] => football )
```

Example:SORT_FLAG_CASE option

```
<?php
$ar = ['football', 'Running', 'cycling', 'Soccer', 'basketball'];
sort($ar, SORT_STRING | SORT_FLAG_CASE);
print_r($ar);
?>
```

Output:

```
Array ( [0] => basketball [1] => cycling [2] => football [3] => Running [4] => Soccer )
```

Note: The SORT_FLAG_CASE option must be combined with SORT_STRING or SORT_NATURAL (with a |) to be effective.

Example: natsort()

```
<?php
$ar = ['img12.png', 'img3.png', 'img22.png', 'img8.png'];
natsort($ar);
print_r($ar);
?>
```

Output:

```
Array ( [1] => img3.png [3] => img8.png [0] => img12.png [2] => img22.png )
```

⇒ Functions asort() and ksort() are for sorting associative arrays, along with arsort() and krsort()for reverse sorts.

⇒ The asort() and arsort() functions sort the values while ksort and krsort sort by key.

Example: Uses asort() to sort an associative array by its values:

```
<?php
```

```
$ar = ['fruit' => 'apple', 'vegetable' => 'cabbage', 'grain' => 'wheat', 'nut' => 'peanut'];
asort($ar);
print_r($ar);
?>
```

Output:

```
Array ( [fruit] => apple [vegetable] => cabbage [nut] => peanut [grain] => wheat )
```

Example: Uses ksort() to sort an associative array by its keys:

```
<?php
$ar = ['fruit' => 'apple', 'vegetable' => 'cabbage', 'grain' => 'wheat', 'nut' => 'peanut'];
ksort($ar);
print_r($ar);
?>
```

Output:

```
Array ( [fruit] => apple [grain] => wheat [nut] => peanut [vegetable] => cabbage )
```

Example: Uses rsort() to sort an associate array by its values in reverse.

```
<?php
$ar = ['football', 'Running', 'cycling', 'Soccer', 'basketball'];
rsort($ar, SORT_STRING | SORT_FLAG_CASE);
print_r($ar);
?>
```

Output:Array ([0] => Soccer [1] => Running [2] => football [3] => cycling [4] => basketball)

Fill Arrays:

⇒ PHP provides several functions that fill arrays with values, including array_fill(), array_fill_keys(), range(), and array_pad().
⇒ These functions all return new arrays and provide various ways to fill their elements.

array_fill():

⇒ This function fills an array with a value you specify.
⇒ Parameters are starting index, the number of elements to fill, and the fill value.

Example:

```
<?php
$ar = array_fill(0, 4, 0);
print_r($ar);
?>
```

Output:Array ([0] => 0 [1] => 0 [2] => 0 [3] => 0)

array_fill_keys():

⇒ This function is similar to array_fill() with the added capability to provide keys for the new array it returns.

- ⇒ The first argument is an array that specifies the keys for the returned array.
- ⇒ The second argument is the value to assign for each key.

Example:

```
<?php
$months = array('Jan', 'Feb', 'Mar', 'Apr', 'May', 'June', 'July', 'Aug', 'Sept', 'Oct', 'Nov', 'Dec');
$ar = array_fill_keys($months, 0);
print_r($ar);
?>
```

Output:

```
Array ( [Jan] => 0 [Feb] => 0 [Mar] => 0 [Apr] => 0 [May] => 0 [June] => 0 [July] => 0 [Aug] => 0
[Sept] => 0 [Oct] => 0 [Nov] => 0 [Dec] => 0 )
```

range():

- ⇒ This function fills an array with a series of values.
- ⇒ You can specify numbers or characters as the start and end values for your series.
- ⇒ This function also includes an optional step argument.

Example:

```
<?php
$ar = range(1, 10);
print_r($ar);
?>
```

Output:

```
Array ( [0] => 1 [1] => 2 [2] => 3 [3] => 4 [4] => 5 [5] => 6 [6] => 7 [7] => 8 [8] => 9 [9] => 10 )
```

array_pad():

- ⇒ This function creates a new array by copying an existing array that you pass to it and padding it to the length you specify with the value you specify.
- ⇒ Takes three arguments: the array you want to copy and pad, the new length you wish it to have, and the value for the new elements.

Example:

```
$ar = array('a', 'b', 'c');
// pad $ar to length of 8 with character 'd'
$ar2 = array_pad($ar, 8, 'd');
print_r($ar2);
```

Output:

```
Array ( [0] => a [1] => b [2] => c [3] => d [4] => d [5] => d [6] => d [7] => d )
```

Note: A negative number inserts new elements at the start of the returned array.

Example:

```
<?php
$ar = array('a', 'b', 'c');
// pass negative size to pad beginning of $ar
$ar2 = array_pad($ar, -6, 'z');
print_r($ar2);
?>
```

Output:

```
Array ( [0] => z [1] => z [2] => z [3] => a [4] => b [5] => c )
```

Combine/Merge Arrays:

array_combine():

⇒ This function creates a new array from two arrays that you pass as arguments to it.
⇒ The first argument provides the keys for the new array while the second argument provides the values.
⇒ If the two arrays passed to this function do not have an equal number of elements, false will be returned.

Example:

```
<?php
$months = array('Jan', 'Feb', 'Mar', 'Apr', 'May', 'June','July', 'Aug', 'Sept', 'Oct', 'Nov', 'Dec');
$ar = array_combine(range(1, 12), $months);
print_r($ar);
?>
```

Output:

```
Array ( [1] => Jan [2] => Feb [3] => Mar [4] => Apr [5] => May [6] => June [7] => July [8] => Aug
[9] => Sept [10] => Oct [11] => Nov [12] => Dec )
```

Note: In the example above, we used the range() function to create an array of values from 1 through 12. That array is used to provide the keys while the $months array provides the values for the array returned by array_combine().

array_merge():

⇒ Any number of arrays can be passed to array_merge(), and the values of each will be appended to the previous in the new array returned.

Example:

```
<?php
$ar1 = ['a', 'b', 'c'];
$ar2 = ['d', 'e', 'f'];
$ar3 = array_merge($ar1, $ar2);
print_r($ar3);
?>
```

Output:
```
Array ( [0] => a [1] => b [2] => c [3] => d [4] => e [5] => f )
```

Note: The keys of the numerically indexed arrays passed to array_merge() are renumbered starting from zero in the array returned.

⇒ When arrays with string keys are passed to array_merge(), values in subsequent arrays will be overwritten by earlier values for matching string keys.

Example:
```php
<?php
$ar1 = ['fruit' => 'apple', 'vegetable' => 'peas', 'grain' => 'oats'];
$ar2 = ['fruit' => 'orange', 'vegetable' => 'kale', 'nut' => 'cashew'];
$ar3 = array_merge($ar1, $ar2);
print_r($ar3);
?>
```
Output:
```
Array ( [fruit] => orange [vegetable] => kale [grain] => oats [nut] => cashew )
```

Array Union Operator:

⇒ The array union operator (+) can also be used to merge arrays.
⇒ If keys in the arrays match, values earlier in the expression will overwrite those later in the expression. This is true for both numeric and string keys.

Example:
```php
<?php
$ar1 = ['a', 'b', 'c'];
$ar2 = ['d', 'e', 'f'];
$ar3 = $ar1 + $ar2;
print_r($ar3);
?>
```

Output:
```
Array ( [0] => a [1] => b [2] => c )
```
Note: In the example above, the values from the second array were not included in the result because they have the same keys as those in the first.

Example:
```php
<?php
$ar1 = [0 => 'a', 1 => 'b', 2 => 'c'];
$ar2 = [5 => 'd', 8 => 'e', 3 => 'f'];
$ar3 = $ar1 + $ar2;
print_r($ar3);
?>
```
Output:Array ([0] => a [1] => b [2] => c [5] => d [8] => e [3] => f)

Note: Explode on your own way - implode and explode of PHP.

PHP array() functions:PHP array functions are listed below with brief descriptions.

Function	Description
array()	Create an array
array_change_key_case()	Changes the case of all keys in an array (either lowercase or uppercase)
array_chunk()	Split an array into chunks of arrays
array_column()	Return the values from a single column in the input array
array_combine()	Creates an array by using one array for keys and another for its values
array_count_values()	Counts all the values of an array
array_diff()	Compare arrays values, and returns the differences
array_diff_assoc()	Compare arrays keys and values, and returns the differences
array_diff_key()	Compare arrays keys, and returns the differences
array_diff_uassoc()	Compare arrays keys and values, using a user-defined key comparison function, and returns the differences
array_diff_ukey()	Compare array keys, using a user-defined key comparison function, and returns the differences
array_fill()	Fill an array with values
array_fill_keys()	Fill an array with values, specifying keys
array_filter()	Filters elements of an array using a user-defined function
array_flip()	Flips or Exchanges all keys with their associated values in an array
array_intersect()	Compare arrays values, and returns the matches
array_intersect_assoc()	Compare arrays keys and values, and returns the matches
array_intersect_key()	Compare arrays keys, and returns the matches
array_intersect_uassoc()	Compare arrays keys and values, using a user-defined key comparison function, and returns the matches
array_intersect_ukey()	Compare arrays keys, using a user-defined key comparison function, and returns the matches

Function	Description
array_keys()	Return all the keys or a subset of the keys of an array
array_key_exists()	Checks if the specified key exists in the array
array_map()	Sends the elements of the given arrays to a user-defined function, which may use it to returns new values
array_merge()	Merges one or more arrays into one array
array_merge_recursive()	Merges one or more arrays into one array recursively
array_multisort()	Sorts multiple or multi-dimensional arrays
array_pad()	Inserts a specified number of items, with a specified value, to an array
array_pop()	Removes the last element of an array, and returns the value of the removed element
array_product()	Calculates the product of the values in an array
array_push()	Inserts one or more elements to the end of an array
array_rand()	Returns one or more random keys from an array
array_reduce()	Reduce the array to a single value by using a user-defined callback function
array_replace()	Replaces the values of the first array with the values from following arrays
array_replace_recursive()	Replaces the values of the first array with the values from following arrays recursively
array_reverse()	Return an array with elements in reverse order
array_search()	Searches an array for a given value and returns the corresponding key if successful
array_shift()	Removes the first element from an array, and returns the value of the removed element
array_slice()	Extract a slice from an array
array_splice()	Remove a portion of the array and replace it with something else
array_sum()	Calculate the sum of values in an array
array_udiff()	Compares only arrays values by using a user-defined comparison

Function	Description
	callback function, and returns the differences
array_udiff_assoc()	Compares arrays values by using a user-defined comparison callback function, with additional keys comparison using an internal (or built-in) function, and returns the differences
array_udiff_uassoc()	Compares arrays keys and values by using two separate user-defined comparison callback functions, and returns the differences
array_uintersect()	Compares only arrays values by using a user-defined comparison callback function, and returns the matches
array_uintersect_assoc()	Compares arrays values by using a user-defined comparison callback function, while uses an internal (or built-in) function for comparing the key, and returns the matches
array_uintersect_uassoc()	Compares arrays keys and values by using two separate user-defined comparison callback functions, and returns the matches
array_unique()	Removes duplicate values from an array
array_unshift()	Adds one or more elements to the beginning of an array
array_values()	Return all the values of an array
array_walk()	Applies a user-defined function to each element of an array
array_walk_recursive()	Applies a user-defined function recursively to each element of an array
asort()	Sort an associative array by value, in ascending order
arsort()	Sort an associative array by value, in reverse or descending order
compact()	Create array containing variables and their values
count()	Count all elements in an array
current()	Return the current element in an array
each()	Return the current key and value pair from an array and advance the array cursor
end()	Sets the internal pointer of an array to its last element
extract()	Import variables into the current symbol table from an array
in_array()	Checks if a value exists in an array

Function	Description
key_exists()	Checks if the specified key exists in the array. Alias of array_key_exists()
key()	Fetches a key from an array
ksort()	Sort an associative array by key, in ascending order
krsort()	Sort an associative array by key, in reverse or descending order
list()	Assign variables as if they were an array
natcasesort()	Sort an array using a case insensitive "natural order" algorithm
natsort()	Sort an array using a "natural order" algorithm
next()	Advance the internal array pointer of an array
pos()	Return the current element in an array. Alias of current()
prev()	Rewind the internal array pointer
range()	Create an array containing a range of elements
reset()	Set the internal pointer of an array to its first element
rsort()	Sort an array in reverse or descending order
shuffle()	Shuffle an array
sizeof()	Count all elements in an array. Alias of count()
sort()	Sort an array in ascending order
uasort()	Sort an array using a user-defined comparison function and maintain index association
uksort()	Sort an array by keys using a user-defined comparison function
usort()	Sort an array by values using a user-defined comparison function

Functions:

⇒ The language PHP contains hundreds of built-in functions that you can use for all sorts of handy jobs — from manipulating text through to reading files, sending email messages and processing images.

⇒ You're not limited to using just the predefined functions, though — you can create your own functions too.

Why make your own functions?

⇒ The first reason is **reusability**. Once a function is defined, it can be used over and over and over again. You can invoke the same function many times in your program, which saves you work. Another aspect of reusability is that a single function can be used in several different (and separate) scripts/pages.

⇒ The second reason is **abstraction.** In order to use a particular function you need to know the following things:
- The name of the function;
- What the function does;
- What arguments/parameters you must give to the function; and
- What kind of result the function returns.

⇒ But remember: If you just want to use the function in your program, you don't have to know how it works inside! You don't have to understand anything about what goes on inside the function.
- It's sort of like driving a car or using a telephone. With an automobile, you don't need to understand every detail about the engine and drive train and wheels, if all you want to do is drive the car. Similarly, with a telephone, you don't have to understand everything about the phone system in order to make a call.

⇒ The only time you need to know how a function works inside is when you need to write the function, or change how it works. (It's like a car again; you need to know how a car works in order to build one or fix one.) But once a function is written and working, you never need to look at its insides again.

⇒ Together, these two reasons make functions extremely useful--practically essential!

⇒ The ability to divide a program into abstract, reusable pieces is what makes it possible to write large scripts/programs that actually work right.

How to create a function?

⇒ The basic syntax of creating a custom function can be given with:
```
function functionName()
{
// Code to be executed
}
```

⇒ The declaration of a user-defined function start with the word **function**, followed by the name of the function you want to create followed by parentheses i.e. () and finally place your function's code between curly brackets {}.

⇒ Function names follow the same rules as other labels in PHP.

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

Output:Today is Wednesday

Functionswith arguments/parameters:

⇒ Information may be passed to functions via the argument list, which is a comma-delimited list of expressions. The arguments are evaluated from left to right.

⇒ PHP supports passing arguments by value (the default), passing by reference, and default argument values. Variable-length argument lists are also supported.

Example:

```
<?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);
?>
```

Output:

```
Sum of the two numbers 10 and 20 is: 30
```

Functions with Optional Parameters and Default Values:

Example:

```
<?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");        // default value will be takes for the parameter $size which is 1.5
?>
```

Output:

```
Hello, world!

Hello, world!

Hello, world!
```

Returning Values from a Function:

⇒ A function can return a value back to the script that called the function using the return statement. The value may be of any type, including arrays and objects.

Example:

```
<?php
// Defining function
function getSum($num1, $num2)
{
   $total = $num1 + $num2;
return $total;
}
// Printing returned value
$total = getSum(5, 10);
echo "<br>Total = " . $total
?>
```

Output:Total = 15

Note: A function cannot return multiple values. However, you can obtain similar results by returning an array.

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.

Example:

```
<?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 "<br>mynum = " . $mynum; // Outputs: 5
selfMultiply($mynum);
echo "<br>After function call mynum = " . $mynum; // Outputs: 25
?>
```

Output:

```
mynum = 5
After function call mynum = 25
```

Variable-length argument lists:

Example:

```
<?php
function sum(...$numbers)
{
        $acc = 0;
        foreach ($numbers as $n)
        {
        $acc += $n;
        }
return $acc;
}
echo "<br>Sum = " . sum(1, 2, 3, 4);
?>
```

Output:Sum = 10

Understanding the Variable Scope:

⇒ The scope of a variable in PHP is the context in which the variable was created, and in which it can be accessed. Essentially, PHP has 2 scopes:
  - Global: The variable is accessible from anywhere in the script
  - Local: The variable is only accessible from within the function (or method) that created it

Example:

```
<?php
$globalName = "ST.Mary's";
function sayHello()
{
  $localName = "Women's Engineering College";
  echo "Hello, localName!: " . $localName . "<br>";
```

```
 //global $globalName;
 echo "<br>Hello, globalName!" . $globalName . "<br>";
}
sayHello();
echo "The value of globalName is: " . $globalName . "<br>";
echo "The value of localName is: " . $localName . "<br>";
?>
```

Output:

```
Hello, localName!: Women's Engineering College
Notice: Undefined variable: globalName in C:\Apache24\htdocs\examples\test.php on line 8
Hello, globalName!
The value of globalName is: ST.Mary's
Notice: Undefined variable: localName in C:\Apache24\htdocs\examples\test.php on line 12
The value of localName is:
```

Accessing global variables from within functions:

⇒ To access a global variable from within a function, however, you first need to declare the variable as **global** within the function by using the **global** keyword.

Example:

```
<?php
$globalName = "ST.Mary's";
function sayHello()
{
 $localName = "Women's Engineering College";
 echo "Hello, localName!: " . $localName . "<br>";
 global $globalName;
 echo "<br>Hello, globalName!" . $globalName . "<br>";
}
sayHello();
echo "The value of globalName is: " . $globalName . "<br>";
echo "The value of localName is: " . $localName . "<br>";
?>
```

Output:

```
Hello, localName!: Women's Engineering College
Hello, globalName!ST.Mary's
The value of globalName is: ST.Mary's
Notice: Undefined variable: localName in C:\Apache24\htdocs\examples\test.php on line 12
The value of localName is:
```

Static scope:

⇒ 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.

Example:

```
<?php
function createWidget()
{
  static $numWidgets = 0;
  return ++$numWidgets;
}
echo "Creating some widgets...<br>";
echo createWidget() . " created so far.<br>";
echo createWidget() . " created so far.<br>";
echo createWidget() . " created so far.<br>";
?>
```

Output:

```
Creating some widgets...
1 created so far.
2 created so far.
3 created so far.
```

Working with HTML Forms:

⇒ An HTML form consists of HTML input elements that allow you to enter the data and submit it to the web server. Those input elements include text input field, password field, checkbox, radio button, file select field, submit button…etc.

⇒ There are two important attributes of the <form> tag:

- **action**: This attribute accepts a relative URL (/contact.php) or an absolute URL (http://localhost/examples/contact.php) or where the data is submitted to when user submits the form.
- **method**: This attribute specifies how the web browser sends the form data. You can use POST or GET. These two methods communicate with web server typically using HTTP (Hypertext Transfer Protocol.
  - The POST method allows you to send large amounts of data, while the GET method is useful for sending small amounts of data via URL.
  - GETalso applies to the QUERY_STRING (the information after the '?' in a URL). So, for example, http://localhost/examples/test.php?id=3 contains GET data which is accessible with $_GET['id']

⇒ In order to read the data from the HTML form, you need to use one of the following superglobal variables:

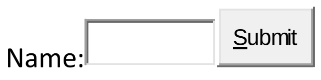
- $_GET array contains a list of field names and their values of the form that uses the GET method.

- $_POST array contains a list of field names and their values of the form that uses the POST method.
- $_REQUEST array contains field names and their values of both $_GET and $_POST, as well as values in the $_COOKIE superglobal array.

Example:

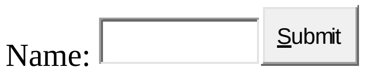
```
<html lang="en">
<head>
        <title>Example of PHP GET method</title>
</head>
<body>
<?php
        if(isset($_GET["name"]))
        {
        echo "<p>Hi, " . $_GET["name"] . "</p>";
        }
?>
<form method="get" action="<?php echo $_SERVER["PHP_SELF"];?>">
        <label for="inputName">Name:</label>
        <input type="text" name="name" id="inputName">
        <input type="submit" value="Submit">
</form>
</body>
</form>
```

Output:



Enter data into text box and click on submit button. You will see the following in browser window:

Hi, ST.MARY's Women's Engineering College



Example:

```
<html lang="en">
<head>
<title>Example of PHP POST method</title>
</head>
<body>
<?php
        if(isset($_POST["name"]))
```

```
		{
		echo "<p>Hi, " . $_POST["name"] . "</p>";
		}
	?>
	<form method="post" action="<?php echo $_SERVER["PHP_SELF"];?>">
		<label for="inputName">Name:</label>
		<input type="text" name="name" id="inputName">
		<input type="submit" value="Submit">
	</form>
	</body>
	</html>
```

Working with MySQL Database:

Introduction to database:

⇒ You deal with data every day…
- When you want to listen to your favorite songs, you open your playlist from your smartphone. In this case, the playlist is a database.
- When you take a photo and upload it to your account on a social network like Facebook, your photo gallery is a database.
- When you browse an e-commerce website to buy shoes, clothes, etc., you use the shopping cart database.

Databases are everywhere. So what is a database?

⇒ By definition, a database is merely a structured collection of data.
⇒ The data relate to each other by nature, e.g., a product belonged to a product category and associated with multiple tags. Therefore, we use the term relational database.
⇒ In the relational database, we model data like products, categories, tags, etc., using tables. A table contains columns and rows. It is like a spreadsheet.
⇒ A table may relate to another table using a relationship, e.g., one-to-one and one-to-many relationships.
⇒ Because we deal with a significant amount of data, we need a way to define the databases, tables, etc., and process data more efficiently. Besides, we want to turn the data into information.
⇒ And this is where SQL comes to play.

SQL – the language of database

⇒ SQL stands for structured query language.
⇒ SQL is the standardized language used to access the database.
⇒ ANSI/SQL defines the SQL standard. The current version of SQL is SQL:2003. Whenever we refer to the SQL standard, we mean the current SQL version.
⇒ SQL contains three parts:
- **Data definition language** includes statements that help you define the database and its objects, e.g., tables, views, triggers, stored procedures, etc.

- **Data manipulation language** contains statements that allow you to update and query data.
- **Data control language** allows you to grant the permissions to a user to access specific data in the database.

What is MySQL:

⇒ MySQL is a database management system that allows you to manage relational databases.
⇒ It is open source software backed by Oracle. It means you can use MySQL without paying. Also, if you want, you can change its source code to suit your needs.
⇒ Even though MySQL is open source software, you can buy a commercial license version from Oracle to get a premium support services.
⇒ MySQL is pretty easy to master in comparison with other database software like Oracle Database, or Microsoft SQL Server.
⇒ MySQL can run on various platforms UNIX, Linux, Windows, etc. You can install it on a server or even on a desktop. Besides, MySQL is reliable, scalable, and fast.
⇒ MySQL is an essential component of LAMP (Linux, Apache, MySQL, and PHP)/WAMP(Windows, Apache, MySQL, and PHP) stack.

MySQL PHP drivers:

⇒ Depending on the version of PHP, there are either two or three PHP APIs for accessing the MySQL database.
⇒ PHP 5 users can choose between the deprecated mysql extension, mysqli, or PDO_MySQL.
⇒ PHP 7 removes the mysql extension, leaving only the latter two options.

Terminology overview:

API:

⇒ An Application Programming Interface, or API, defines the classes, methods, functions and variables that your application will need to call in order to carry out its desired task.
⇒ In the case of PHP applications that need to communicate with databases the necessary APIs are usually exposed via PHP extensions.
⇒ APIs can be procedural or object-oriented. With a procedural API you call functions to carry out tasks, with the object-oriented API you instantiate classes and then call methods on the resulting objects. Of the two the latter is usually the preferred interface, as it is more modern and leads to better organized code.
⇒ When writing PHP applications that need to connect to the MySQL server there are several API options available.

Connector:

⇒ In the MySQL documentation, the term connector refers to a piece of software that allows your application to connect to the MySQL database server.
⇒ MySQL provides connectors for a variety of languages, including PHP.
⇒ If your PHP application needs to communicate with a database server you will need to write PHP code to perform such activities as
  - Connecting to the database server,
  - Querying the database and other database-related functions.

⇒ Software is required to provide the API that your PHP application will use, and also handle the communication between your application and the database server, possibly using other intermediate libraries where necessary. This software is known generically as a connector, as it allows your application to connect to a database server.

Driver:

⇒ A driver is a piece of software designed to communicate with a specific type of database server. The driver may also call a library, such as the MySQL Client Library or the MySQL Native Driver. These libraries implement the low-level protocol used to communicate with the MySQL database server.

⇒ By way of an example, the PHP Data Objects (PDO) database abstraction layer may use one of several database-specific drivers. One of the drivers it has available is the PDO MYSQL driver, which allows it to interface with the MySQL server.

⇒ Sometimes people use the terms connector and driver interchangeably, this can be confusing. In the MySQL-related documentation the term driver is reserved for software that provides the database-specific part of a connector package.

Extension:

⇒ In the PHP documentation you will come across another term - extension. The PHP code consists of a core, with optional extensions to the core functionality.

⇒ PHP's MySQL-related extensions, such as the mysqli extension, and the mysql extension, are implemented using the PHP extension framework.

⇒ An extension typically exposes an API to the PHP programmer, to allow its facilities to be used programmatically. However, some extensions which use the PHP extension framework do not expose an API to the PHP programmer.

⇒ The PDO MySQL driver extension, for example, does not expose an API to the PHP programmer, but provides an interface to the PDO layer above it.

⇒ The terms API and extension should not be taken to mean the same thing, as an extension may not necessarily expose an API to the programmer.

Note: It is recommended to use either the mysqli or PDO_MySQL extensions. The PHP's MySQLi extension provides both speed and feature benefits over the PDO extension, so it could be a better choice for MySQL-specific projects.

PHP MySQLi Functions: mysqli_connect, mysqli_select_db, mysqli_query, mysqli_num_rows, mysqli_fetch_array, mysqli_close:

mysqli_connect ():

⇒ This function is used to connect to a MySQL database server. This function returns a database connection resource variable and the syntax is as follows:

```
mysqli_connect(db_server_name, db_user_name, db_password);
```

Where

⇒ db_server_name is the name or IP address of the server hosting MySQL server.

⇒ db_user_name is a valid user name in MySQL server.

⇒ db_password is a valid password associated with a user name in MySQL server.

mysqli_select_db():

⇒ This function is used to select a database and the syntax is as follows:

mysqli_select_db(db_handle,database_name);

Where

⇒ db_handler is optional, it is used to pass in the server connection link (return value of mysqli_connect() function).

⇒ database_name is the name of the database.

mysqli_query():

⇒ This function is used to execute SQL queries. The function can be used to execute the query types – insert, select, update, update. The syntax is as follows:

mysqli_query(db_handle, query);

Where

⇒ db_handle is the return value of mysqil_connect() function.

⇒ query is the SQL query to be executed.

mysqli_num_rows():

⇒ This function is used to get the number of rows returned by a select query. It has the following syntax.

mysqli_num_row(query);

Where

⇒ query is mysqli_query result set.

mysqli_fetch_array():

⇒ This function is used to fetch row arrays from a query result set and the syntax is as follows:

mysqli_fetch_array(result);

Where

⇒ result is the result returned by the mysqli_query() function.

mysqli_close():

⇒ This function is used to close an open database connection.It has the following syntax.

mysqli_close(db_handle);

Where

⇒ db_handler is optional, it is used to pass in the server connection resource.

PHP Data Access Object PDO:

⇒ The PDO is a class that allows us to manipulate different database engines such as MySQL, PostGres, MS SQL Server etc.

⇒ Connections are established by creating instances of the PDO base class.

⇒ It doesn't matter which driver you want to use; you always use the PDO class name.

⇒ The constructor accepts parameters for specifying the database source (known as the DSN) and optionally for the username and password (if any).

⇒ If there are any connection errors, a PDOException object will be thrown. You may catch the exception if you want to handle the error condition, or you may opt to leave it for an application global exception handler that you set up via set_exception_handler().

Example: Procedural way of connecting to MySQL database.

```
<?php
/* Attempt MySQL server connection. Assuming you are running MySQLserver with default
setting (user 'root' with no password) */
$link = mysqli_connect("localhost", "root", "");
// Check connection
if($link === false)
{
   die("ERROR: Could not connect. ".mysqli_connect_error());
}
// Print host information
echo "Connect Successfully. Host info: ".mysqli_get_host_info($link);
?>
```

Example: Object-oriented way of connecting to MySQL database.

```
<?php
/* Attempt MySQL server connection. Assuming you are running MySQLserver with default
setting (user 'root' with no password) */
$mysqli = new mysqli("localhost", "root", "", "demo");

// Check connection
if($mysqli === false)
{
   die("ERROR: Could not connect. " . $mysqli->connect_error);
}
// Print host information
echo "Connect Successfully. Host info: " . $mysqli->host_info;
// Close connection
$mysqli->close();
?>
```

Example: PDO way of connecting to MySQL database.

```
<?php
/* Attempt MySQL server connection. Assuming you are running MySQLserver with default
setting (user 'root' with no password) */
try
{
$pdo = new PDO("mysql:host=localhost;dbname=demo", "root", "");
// Set the PDO error mode to exception
$pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
// Print host information
echo "Connect Successfully. Host info: " .
$pdo->getAttribute(constant("PDO::ATTR_CONNECTION_STATUS"));
}
catch(PDOException $e)
{
die("ERROR: Could not connect. " . $e->getMessage());
}
// Close connection
unset($pdo);
```

Note: Setting the PDO::ATTR_ERRMODE attribute to PDO::ERRMODE_EXCEPTION tells PDO to throw exceptions whenever a database error occurs.

Example: Proceduralway of creating a database in PHP using mysqli_query()

```
<?php
$link = mysqli_connect("localhost", "root", "");
// Check connection
if($link === false)
{
   die("ERROR: Could not connect. " . mysqli_connect_error());
}
// Attempt create database query execution
$sql = "CREATE DATABASE demo";
if(mysqli_query($link, $sql))
{
   echo "Database created successfully";
}
else
{
   echo "ERROR: Could not able to execute $sql. " . mysqli_error($link);
}
// Close connection
mysqli_close($link);
?>
```

Example:Object oriented way of creating a database in PHP using mysqli_query()

```
<?php
$mysqli = new mysqli("localhost", "root", "");
// Check connection
if($mysqli === false)
{
   die("ERROR: Could not connect. " . $mysqli->connect_error);
}
// Attempt create database query execution
```

```
$sql = "CREATE DATABASE demo";
if($mysqli->query($sql) === true)
{
   echo "Database created successfully";
}
else
{
   echo "ERROR: Could not able to execute $sql. " . $mysqli->error;
}
// Close connection
$mysqli->close();
?>
```

Example: PDO way to create a database

```
<?php
try
{
   $pdo = new PDO("mysql:host=localhost;", "root", "");
   // Set the PDO error mode to exception
   $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
}
catch(PDOException $e)
{
   die("ERROR: Could not connect. " . $e->getMessage());
}
// Attempt create database query execution
try
{
   $sql = "CREATE DATABASE demo";
   $pdo->exec($sql);
   echo "Database created successfully";
}
catch(PDOException $e)
{
   die("ERROR: Could not able to execute $sql. " . $e->getMessage());
}
// Close connection
unset($pdo);
?>
```

Example: Procedural way to create Tables inside MySQL Database

```
<?php
```

```
$link = mysqli_connect("localhost", "root", "", "demo");
// Check connection
if($link === false)
{
   die("ERROR: Could not connect. " . mysqli_connect_error());
}
// Attempt create table query execution
$sql = "CREATE TABLE persons(
   id INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
   first_name VARCHAR(30) NOT NULL,
   last_name VARCHAR(30) NOT NULL,
   email VARCHAR(70) NOT NULL UNIQUE
)";
if(mysqli_query($link, $sql))
{
   echo "Table created successfully.";
}
else
{
   echo "ERROR: Could not able to execute $sql. " . mysqli_error($link);
}
// Close connection
mysqli_close($link);
?>
```

Example: Object-oriented way to create tables inside MySQL database

```
<?php
$mysqli = new mysqli("localhost", "root", "", "demo");
// Check connection
if($mysqli === false)
{
   die("ERROR: Could not connect. " . $mysqli->connect_error);
}
// Attempt create table query execution
$sql = "CREATE TABLE persons(
   id INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
   first_name VARCHAR(30) NOT NULL,
   last_name VARCHAR(30) NOT NULL,
   email VARCHAR(70) NOT NULL UNIQUE
)";
if($mysqli->query($sql) === true)
{
    echo "Table created successfully.";
}
else
{
    echo "ERROR: Could not able to execute $sql. " . $mysqli->error;
}
// Close connection
$mysqli->close();
?>
```

Example: PDO way to create table inside MySQL database

```
<?php
try
{
    $pdo = new PDO("mysql:host=localhost;dbname=demo", "root", "");
    // Set the PDO error mode to exception
    $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
}
catch(PDOException $e)
{
    die("ERROR: Could not connect. " . $e->getMessage());
}
// Attempt create table query execution
try
{
    $sql = "CREATE TABLE persons(
        id INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
        first_name VARCHAR(30) NOT NULL,
        last_name VARCHAR(30) NOT NULL,
        email VARCHAR(70) NOT NULL UNIQUE
    )";
    $pdo->exec($sql);
    echo "Table created successfully.";
}
catch(PDOException $e)
{
    die("ERROR: Could not able to execute $sql. " . $e->getMessage());
}
// Close connection
unset($pdo);
?>
```

Example: Procedural way of inserting data into a table using mysqli_query()

```
<?php
$link = mysqli_connect("localhost", "root", "", "demo");
// Check connection
if($link === false)
{
   die("ERROR: Could not connect. " . mysqli_connect_error());
}
// Attempt insert query execution
$sql = "INSERT INTO persons (first_name, last_name, email) VALUES ('Peter', 'Parker',
'peterparker@mail.com')";
if(mysqli_query($link, $sql))
{
   echo "Records inserted successfully.";
}
else
{
   echo "ERROR: Could not able to execute $sql. " . mysqli_error($link);
}
// Close connection
mysqli_close($link);
?>
```

Example: Object-oriented way of inserting data into a table using mysqli_query()

```
<?php
$mysqli = new mysqli("localhost", "root", "", "demo");
// Check connection
if($mysqli === false)
{
   die("ERROR: Could not connect. " . $mysqli->connect_error);
}
// Attempt insert query execution
$sql = "INSERT INTO persons (first_name, last_name, email) VALUES ('Peter', 'Parker',
'peterparker@mail.com')";
if($mysqli->query($sql) === true)
{
   echo "Records inserted successfully.";
}
else
{
   echo "ERROR: Could not able to execute $sql. " . $mysqli->error;
}
```

```
// Close connection
$mysqli->close();
?>
```

Example: PDO way of inserting data into a table

```
<?php
try
{
   $pdo = new PDO("mysql:host=localhost;dbname=demo", "root", "");
   // Set the PDO error mode to exception
   $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
}
catch(PDOException $e)
{
   die("ERROR: Could not connect. " . $e->getMessage());
}
// Attempt insert query execution
try
{
   $sql = "INSERT INTO persons (first_name, last_name, email) VALUES ('Peter', 'Parker',
'peterparker@mail.com')";
   $pdo->exec($sql);
   echo "Records inserted successfully.";
}
catch(PDOException $e)
{
   die("ERROR: Could not able to execute $sql. " . $e->getMessage());
}
// Close connection
unset($pdo);
?>
```

Example: Insert Data into a Database from an HTML Form

Step 1: Creating HTML form (save as persons.html)

```
<html lang="en">
<head>
<meta charset="UTF-8">
<title>Add Record Form</title>
</head>
<body>
<form action="insert.php" method="post">
<p>
```

```
<label for="firstName">First Name:</label>
<input type="text" name="first_name" id="firstName">
</p>
<p>
<label for="lastName">Last Name:</label>
<input type="text" name="last_name" id="lastName">
</p>
<p>
<label for="emailAddress">Email Address:</label>
<input type="text" name="email" id="emailAddress">
</p>
<input type="submit" value="Submit">
</form>
</body>
</html>
```

Step 2:Retrieving and Inserting the Form Data

⇒ When a user clicks the submit button of the HTML form in the example above, the form data is sent to 'insert.php' file. The 'insert.php' file connects to the MySQL database server, retrieves forms fields using the PHP's $_REQUEST variables and finally execute the insert query to add the records.

```
<?php
$mysqli = new mysqli("localhost", "root", "", "demo");
// Check connection
if($mysqli === false)
{
   die("ERROR: Could not connect. " . $mysqli->connect_error);
}
// Escape user inputs for security
$first_name = $mysqli->real_escape_string($_REQUEST['first_name']);
$last_name = $mysqli->real_escape_string($_REQUEST['last_name']);
$email = $mysqli->real_escape_string($_REQUEST['email']);
// Attempt insert query execution
$sql = "INSERT INTO persons (first_name, last_name, email) VALUES ('$first_name',
'$last_name', '$email')";
if($mysqli->query($sql) === true)
{
   echo "Records inserted successfully.";
}
else
{
   echo "ERROR: Could not able to execute $sql. " . $mysqli->error;
}
// Close connection
```

```
$mysqli->close();
?>
```

PHP MySQL Prepared Statements:

⇒ A prepared statement (also known as parameterized statement) is simply a SQL query template containing placeholder instead of the actual parameter values.
⇒ These placeholders will be replaced by the actual values at the time of execution of the statement.
⇒ MySQLi supports the use of anonymous positional placeholder (?).
⇒ The prepared statement execution consists of two stages: prepare and execute.
- Prepare — at the prepare stage a SQL statement template is created and sent to the database server. The server parses the statement template, performs a syntax check and query optimization, and stores it for later use.
- Execute — during execute the parameter values are sent to the server. The server creates a statement from the statement template and these values to execute it.

⇒ A prepared statement is very useful, particularly in situations when you execute a particular statement multiple times with different values, for example, a series of INSERT statements.

```
<?php
/* Attempt MySQL server connection. Assuming you are running MySQL
server with default setting (user 'root' with no password) */
$mysqli = new mysqli("localhost", "root", "", "demo");
// Check connection
if($mysqli === false)
{
   die("ERROR: Could not connect. " . $mysqli->connect_error);
}
// Prepare an insert statement
$sql = "INSERT INTO persons (first_name, last_name, email) VALUES (?, ?, ?)";
if($stmt = $mysqli->prepare($sql))
{
   // Bind variables to the prepared statement as parameters
   $stmt->bind_param("sss", $first_name, $last_name, $email);
   /* Set the parameters values and execute the statement again to insert another row */
   $first_name = "Hermione";
   $last_name = "Granger";
   $email = "hermionegranger@mail.com";
   $stmt->execute();
   /* Set the parameters values and execute the statement to insert a row */
   $first_name = "Ron";
   $last_name = "Weasley";
   $email = "ronweasley@mail.com";
   $stmt->execute();
echo "Records inserted successfully.";
}
else
{
   echo "ERROR: Could not prepare query: $sql. " . $mysqli->error;
}
// Close statement
$stmt->close();
// Close connection
$mysqli->close();
?>
```

Note: Prepared statements provide strong protection against SQL injection, because parameter values are not embedded directly inside the SQL query string. The parameter values are sent to the database server separately from the query using a different protocol and thus cannot interfere with it. The server uses these values directly at the point of execution, after the statement template is parsed. That's why the prepared statements are less error-prone, and thus considered as one of the most critical element in database security.

Selecting Data from Database Tables:

```
<?php
/* Attempt MySQL server connection. Assuming you are running MySQLserver with default
setting (user 'root' with no password) */
$mysqli = new mysqli("localhost", "root", "", "demo");
// Check connection
if($mysqli === false)
{
   die("ERROR: Could not connect. " . $mysqli->connect_error);
}
// Attempt select query execution
$sql = "SELECT * FROM persons";
if($result = $mysqli->query($sql))
{
        if($result->num_rows > 0)
        {
        echo "<table>";
       echo "<tr>";
        echo "<th>id</th>";
        echo "<th>first_name</th>";
        echo "<th>last_name</th>";
        echo "<th>email</th>";
        echo "</tr>";
        while($row = $result->fetch_array())
        {
```

```
        echo "<tr>";
          echo "<td>" . $row['id'] . "</td>";
          echo "<td>" . $row['first_name'] . "</td>";
          echo "<td>" . $row['last_name'] . "</td>";
          echo "<td>" . $row['email'] . "</td>";
        echo "</tr>";
         }
         echo "</table>";
         // Free result set
         $result->free();
         }
         else
         {
         echo "No records matching your query were found.";
         }
}
else
{
   echo "ERROR: Could not able to execute $sql. " . $mysqli->error;
}
// Close connection
$mysqli->close();
?>
```

Updating Database Table Data:

```
<?php
/* Attempt MySQL server connection. Assuming you are running MySQLserver with default
setting (user 'root' with no password) */
$mysqli = new mysqli("localhost", "root", "", "demo");
// Check connection
if($mysqli === false)
{
   die("ERROR: Could not connect. " . $mysqli->connect_error);
}
// Attempt update query execution
$sql = "UPDATE persons SET email='peterparker_new@mail.com' WHERE id=1";
if($mysqli->query($sql) === true)
{
   echo "Records were updated successfully.";
}
else
{
   echo "ERROR: Could not able to execute $sql. " . $mysqli->error;
}
// Close connection
$mysqli->close();
?>
```

Deleting Database Table Data

```
<?php
/* Attempt MySQL server connection. Assuming you are running MySQLserver with default
setting (user 'root' with no password) */
$mysqli = new mysqli("localhost", "root", "", "demo");
// Check connection
if($mysqli === false)
{
   die("ERROR: Could not connect. " . $mysqli->connect_error);
}
// Attempt delete query execution
$sql = "DELETE FROM persons WHERE first_name='John'";
if($mysqli->query($sql) === true)
{
   echo "Records were deleted successfully.";
}
else
{
   echo "ERROR: Could not able to execute $sql. " . $mysqli->error;
}
// Close connection
$mysqli->close();
?>
```

PHP MySQL Login System

Step 1: Creating the Database Table
Step 2: Creating the config File

```
<?php
/* Database credentials. Assuming you are running MySQL
server with default setting (user 'root' with no password) */
define('DB_SERVER', 'localhost');
define('DB_USERNAME', 'root');
define('DB_PASSWORD', '');
define('DB_NAME', 'demo');
/* Attempt to connect to MySQL database */
$mysqli = new mysqli(DB_SERVER, DB_USERNAME, DB_PASSWORD, DB_NAME);
```

```
// Check connection
if($mysqli === false)
{
   die("ERROR: Could not connect. " . $mysqli->connect_error);
}
?>
```

Step 3: Creating the Registration Form

```
<?php
// Include config file
require_once "config.php";
// Define variables and initialize with empty values
$username = $password = $confirm_password = "";
$username_err = $password_err = $confirm_password_err = "";
// Processing form data when form is submitted
if($_SERVER["REQUEST_METHOD"] == "POST")
{
   // Validate username
        if(empty(trim($_POST["username"]))){
        $username_err = "Please enter a username.";
}
else
{
      // Prepare a select statement
      $sql = "SELECT id FROM users WHERE username = ?";
      if($stmt = $mysqli->prepare($sql)){
         // Bind variables to the prepared statement as parameters
         $stmt->bind_param("s", $param_username);
         // Set parameters
         $param_username = trim($_POST["username"]);
         // Attempt to execute the prepared statement
         if($stmt->execute())
                  {
           // store result
           $stmt->store_result();
           if($stmt->num_rows == 1)
                  {
              $username_err = "This username is already taken.";
           }
                  else
                  {
              $username = trim($_POST["username"]);
           }
```

```
            }
            else
            {
              echo "Oops! Something went wrong. Please try again later.";
            }
        }
         // Close statement
        $stmt->close();
    }
    // Validate password
    if(empty(trim($_POST["password"])))
            {
        $password_err = "Please enter a password.";
            }
            elseif(strlen(trim($_POST["password"])) < 6)
            {
        $password_err = "Password must have atleast 6 characters.";
            }
            else
            {
        $password = trim($_POST["password"]);
            }
    // Validate confirm password
    if(empty(trim($_POST["confirm_password"]))){
        $confirm_password_err = "Please confirm password.";
    } else{
        $confirm_password = trim($_POST["confirm_password"]);
        if(empty($password_err) && ($password != $confirm_password)){
            $confirm_password_err = "Password did not match.";
        }
    }

    // Check input errors before inserting in database
    if(empty($username_err) && empty($password_err) && empty($confirm_password_err)){

        // Prepare an insert statement
        $sql = "INSERT INTO users (username, password) VALUES (?, ?)";

        if($stmt = $mysqli->prepare($sql)){
            // Bind variables to the prepared statement as parameters
            $stmt->bind_param("ss", $param_username, $param_password);
            // Set parameters
            $param_username = $username;
            $param_password = password_hash($password, PASSWORD_DEFAULT); // Creates a
password hash

            // Attempt to execute the prepared statement
            if($stmt->execute()){
                // Redirect to login page
                header("location: login.php");
            } else{
                echo "Something went wrong. Please try again later.";
            }
        }

        // Close statement
        $stmt->close();
    }

    // Close connection
    $mysqli->close();
}
?>

<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<title>Sign Up</title>
<link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.css">
<style type="text/css">
        body{ font: 14px sans-serif; }
        .wrapper{ width: 350px; padding: 20px; }
</style>
</head>
<body>
<div class="wrapper">
<h2>Sign Up</h2>
<p>Please fill this form to create an account.</p>
<form action="<?php echo htmlspecialchars($_SERVER["PHP_SELF"]); ?>" method="post">
<div class="form-group <?php echo (!empty($username_err)) ? 'has-error' : ''; ?>">
<label>Username</label>
```

```
<input type="text" name="username" class="form-control" value="<?php echo $username;
?>">
<span class="help-block"><?php echo $username_err; ?></span>
</div>
<div class="form-group <?php echo (!empty($password_err)) ? 'has-error' : ''; ?>">
<label>Password</label>
<input type="password" name="password" class="form-control" value="<?php echo
$password; ?>">
<span class="help-block"><?php echo $password_err; ?></span>
</div>
<div class="form-group <?php echo (!empty($confirm_password_err)) ? 'has-error' : ''; ?>">
<label>Confirm Password</label>
<input type="password" name="confirm_password" class="form-control" value="<?php echo
$confirm_password; ?>">
<span class="help-block"><?php echo $confirm_password_err; ?></span>
</div>
<div class="form-group">
<input type="submit" class="btn btn-primary" value="Submit">
<input type="reset" class="btn btn-default" value="Reset">
</div>
<p>Already have an account? <a href="login.php">Login here</a>.</p>
</form>
</div>
</body>
</html>
```

References:

⇒ Programming the World Wide Web 8th Edition Robert W Sebesta Pearson – Chapter 9 – Page 357 to 383, Page 386 to 392, Page 572 to 580.
⇒ https://www.php.net/manual/en/langref.php
⇒ https://www.dyn-web.com/php/

*******END*******