UNIT-3
PHP Basics

PHP Features:

Every user has specific reasons for using PHP to implement a mission-critical application, although one could argue that such motives tend to fall into four key categories: practicality, power, possibility, and price.

i) Practicality

From the very start, the PHP language was created with practicality in mind. After all, Lerdorf's original intention was not to design an entirely new language, but to resolve a problem that had no readily available solution. Furthermore, much of PHP's early evolution was not the result of the explicit intention to improve the language itself, but rather to increase its utility to the user. The result is a language that allows the user to build powerful applications even with a minimum of knowledge.

PHP is a *loosely typed* language, meaning there is no need to explicitly create, typecast, or destroy a variable, although you are not prevented from doing so. PHP handles such matters internally, creating variables on the fly as they are called in a script, and employing a best-guess formula for automatically typecasting variables. For instance, PHP considers the following set of statements to be perfectly valid:

```
<?php
$number = "5"; // $number is a string
$sum = 15 + $number; // Add an integer and string to produce integer
$sum = "twenty"; // Overwrite $sum with a string.
?>
```

PHP will also automatically destroy variables and return resources to the system when the script completes.

Power

PHP developers have almost 200 native libraries containing well over 1,000 functions, in addition to thousands of third-party extensions. Although you're likely aware of PHP's ability to interface with databases, manipulate form information, and create pages dynamically, you might not know that PHP can also do the following:

- Create and manipulate Adobe Flash and Portable Document Format (PDF) files.
- Evaluate a password for guessability by comparing it to language dictionaries and easily broken patterns.
- Parse even the most complex of strings using the POSIX and Perl-based regular expression libraries.
- Authenticate users against login credentials stored in flat files, databases, and even Microsoft's Active Directory.
- Communicate with a wide variety of protocols, including LDAP, IMAP, POP3, NNTP, and DNS, among others.

• Tightly integrate with a wide array of credit-card processing solutions.

Possibility

PHP developers are rarely bound to any single implementation solution. On the contrary, a user is typically fraught with choices offered by the language. For example, consider PHP's array of database support options. Native support is offered for more than 25 database products, including Adabas D,dBase, Empress, FilePro, FrontBase, Hyperwave, IBM DB2, Informix, Ingres, InterBase, mSQL, Microsoft SQL Server, MySQL, Oracle, Ovrimos, PostgreSQL, Solid, Sybase, Unix dbm, and Velocis.

PHP's flexible string-parsing capabilities offer users of differing skill sets the opportunity to not only immediately begin performing complex string operations but also to quickly port programs of similar functionality (such as Perl and Python) over to PHP.

Price

PHP is available free of charge! Since its inception, PHP has been without usage, modification, and redistribution restrictions. In recent years, software meeting such open licensing qualifications has been referred to as *open source software*. Open source software and the Internet go together like bread and butter. Open source projects such as Sendmail, Bind, Linux, and Apache all play enormous roles in the ongoing operations of the Internet at large. Although open source software's free availability has been the point most promoted by the media, several other characteristics are equally important:

Free of licensing restrictions imposed by most commercial products: Open source software users are freed of the vast majority of licensing restrictions one would expect of commercial counterparts. Although some discrepancies do exist among license variants, users are largely free to modify, redistribute, and integrate the software into other products.

Open development and auditing process: Although not without incidents, open source software has long enjoyed a stellar security record. Such high-quality standards are a result of the open development and auditing process. Because the source code is freely available for anyone to examine, security holes and potential problems are rapidly found and fixed. This advantage was perhaps best summarized by open source advocate Eric S. Raymond, who wrote "Given enough eyeballs, all bugs are shallow."

Participation is encouraged: Development teams are not limited to a particular organization. Anyone who has the interest and the ability is free to join the project. The absence of member restrictions greatly enhances the talent pool for a given project, ultimately contributing to a higher-quality product.

Embedding PHP Code in Your Web Pages

One of PHP's advantages is that you can embed PHP code directly alongside HTML. The engine needs some means to immediately determine which areas of the page are PHP-enabled. This is logically accomplished by delimiting the PHP code. There are four delimitation variants.

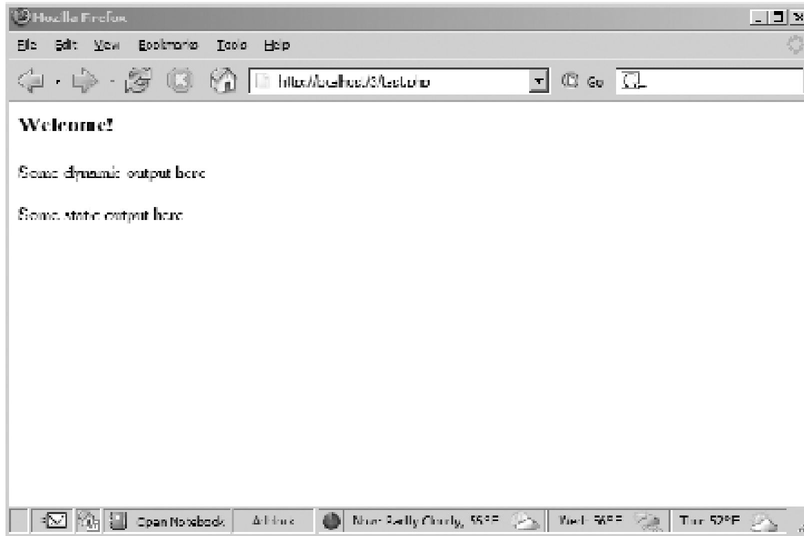
Default Syntax

The default delimiter syntax opens with <?php and concludes with ?>, like this:

```
<h3>Welcome!</h3>
```

```
<?php
echo "<p>Some dynamic output here</p>";
?>
<p>Some static output here</p>
```

Expected output:



Short-Tags

For less motivated typists, an even shorter delimiter syntax is available. Known as *short-tags*, this syntax forgoes the php reference required in the default syntax. However, to use this feature, you need to enable PHP's short_open_tag directive. An example follows:

```
<?
print "This is another PHP example.";
?>
```

When short-tags syntax is enabled and you want to quickly escape to and from PHP to output a bitof dynamic text, you can omit these statements using an output variation known as *short-circuit syntax:*

```
<?="This is another PHP example.";?>
```

This is functionally equivalent to both of the following variations:

```
<? echo "This is another PHP example."; ?>
<?php echo "This is another PHP example.";?>
```

Script

Certain editors have historically had problems dealing with PHP's more commonly used escape syntax variants. Therefore, support for another mainstream delimiter variant, <script>, is offered:

```
<script language="php">
print "This is another PHP example.";
</script>
```

ASP Style

Microsoft ASP pages employ a delimiting strategy similar to that used by PHP, delimiting static from dynamic syntax by using a predefined character pattern: opening dynamic syntax with <%, and concluding with %>. If you're coming from an ASP background and prefer to continue using this escape syntax, PHP supports it. Here's an example:

```
<%
print "This is another PHP example.";
%>
```

Embedding Multiple Code Blocks

You can escape to and from PHP as many times as required within a given page. For instance, the following example is perfectly acceptable:

```
<html>
<head>
<title><?php echo "Welcome to my web site!";?></title>
</head>
<body>
<?php
$date = "July 26, 2010";
?>
<p>Today's date is <?=$date;?></p>
</body>
</html>
```

Outputting Data to the Browser

a. The print() Statement

The print() statement outputs data passed to it . Its prototype looks like this:

int print(argument)

All of the following are plausible print() statements:

```
<?php
print("<p>I love the summertime.</p>");
?>
```

```
<?php
$season = "summertime";
print "<p>I love the $season.</p>";
?>
```

```
<?php
print "<p>I love the
summertime.</p>";
?>
```

All these statements produce identical output:

I love the summertime.

b. The echo() Statement

Alternatively, you could use the echo() statement for the same purposes as print().echo()'s prototype looks like this:

void echo(string argument1 [, ...string argumentN])

To use echo(), just provide it with an argument just as was done with print():

```
echo "I love the summertime.";
```

As you can see from the prototype, echo() is capable of outputting multiple strings. Here's an example:

```
<?php
$heavyweight = "Lennox Lewis";
$lightweight = "Floyd Mayweather";
echo $heavyweight, " and ", $lightweight, " are great fighters.";
?>
```

This code produces the following:
Lennox Lewis and Floyd Mayweather are great fighters.

c. The printf() Statement

The printf() statement is ideal when you want to output a blend of static text and dynamic information stored within one or several variables. It's ideal for two reasons.

- First, it neatly separates the static and dynamic data into two distinct sections, allowing for easy maintenance.
- Second, printf() allows you to wield considerable control over how the dynamic information is rendered to the screen in terms of its type, precision, alignment, and position.

Its prototype looks like this:

integer printf(string format [, mixed args])

For example, suppose you wanted to insert a single dynamic integer value into an otherwise static string:

```
printf("Bar inventory: %d bottles of tonic water.", 100);
```

Executing this command produces the following:
Bar inventory: 100 bottles of tonic water.

Commonly Used Type Specifiers
%b Argument considered an integer; presented as a binary number
%c Argument considered an integer; presented as a character corresponding to that ASCII value
%d Argument considered an integer; presented as a signed decimal number
%f Argument considered a floating-point number; presented as a floating-point number
%o Argument considered an integer; presented as an octal number
%s Argument considered a string; presented as a string
%u Argument considered an integer; presented as an unsigned decimal number
%x Argument considered an integer; presented as a lowercase hexadecimal number
%X Argument considered an integer; presented as an uppercase hexadecimal number

d. The sprintf() Statement
The sprintf() statement is functionally identical to printf() except that the output is assigned to a string rather than rendered to the browser. The prototype follows:
string sprintf(string format [, mixed arguments])

An example follows:
```
$cost = sprintf("$%.2f", 43.2); // $cost = $43.20
```

PHP's Supported Data Types

A *datatype* is the generic name assigned to any data sharing a common set of characteristics. Common data types include Boolean, integer, float, string, and array.

Scalar Data Types
Scalar data types are used to represent a single value. Several data types fall under this category, including Boolean, integer, float, and string.

Boolean
The Boolean datatype is named after George Boole (1815–1864), a mathematician who is considered to be one of the founding fathers of information theory. The *Boolean* data type represents truth, supporting only two values: TRUE and FALSE (case insensitive). Alternatively, you can use zero to represent FALSE, and any nonzero value to represent TRUE.
A few examples follow:

```
$alive = false; // $alive is false.
$alive = 1; // $alive is true.
$alive = -1; // $alive is true.
$alive = 5; // $alive is true.
$alive = 0; // $alive is false.
```

Integer

An *integer* is representative of any whole number or, in other words, a number that does not contain fractional parts. PHP supports integer values represented in base 10 (decimal), base 8 (octal), and base 16 (hexadecimal) numbering systems. Several examples follow:

```
42 // decimal
-678900 // decimal
0755 // octal
0xC4E // hexadecimal
```

The maximum supported integer size is platform-dependent, although this is typically positive or negative 2^31 for PHP version 5 and earlier. PHP 6 introduced a 64-bit integer value, meaning PHP will support integer values up to positive or negative 2^63 in size.

Float

Floating-point numbers, also referred to as *floats*, *doubles*, or *real numbers*, allow you to specify numbers that contain fractional parts. Floats are used to represent monetary values, weights, distances, and a whole host of other representations in which a simple integer value won't suffice. PHP's floats can be specified in a variety of ways, several of which are demonstrated here:

```
4.5678
4.0
8.7e4
1.23E+11
```

String

Simply put, a string is a sequence of characters treated as a contiguous group. *Strings* are delimited by single or double quotes. The following are all examples of valid strings:

```
"PHP is a great language"
"whoop-de-do"
'*9subway\n'
"123$%^789"
```

For example, consider the following string:

```
$color = "maroon";
```

You could retrieve a particular character of the string by treating the string as an array, like this:

```
$parser = $color[2]; // Assigns 'r' to $parser
```

Compound Data Types

Compound data types allow for multiple items of the same type to be aggregated under a single representative entity. The *array* and the *object* fall into this category.

Array

It's often useful to aggregate a series of similar items together, arranging and referencing them in some specific way. This data structure, known as an *array*, is formally defined as an indexed collection of data values. Each member of the array index (also known as the *key*) references a

corresponding value and can be a simple numerical reference to the value's position in the series, or it could have some direct correlation to the value.

For example, if you were interested in creating a list of U.S. states, you could use a numerically indexed array, like so:

```
$state[0] = "Alabama";
$state[1] = "Alaska";
$state[2] = "Arizona";
...
$state[49] = "Wyoming";
```

But what if the project required correlating U.S. states to their capitals? Rather than base the keys on a numerical index, you might instead use an associative index, like this:

```
$state["Alabama"] = "Montgomery";
$state["Alaska"] = "Juneau";
$state["Arizona"] = "Phoenix";
...
$state["Wyoming"] = "Cheyenne";
```

Object

The other compound datatype supported by PHP is the object. The *object* is a central concept of the object-oriented programming paradigm. Unlike the other data types contained in the PHP language, an object must be explicitly declared. This declaration of an object's characteristics and behavior takes place within something called a *class*.

Here's a general example of a class definition and subsequent invocation:

```
class Appliance {
private $_power;
function setPower($status) {
$this->_power = $status;
}
}
...
$blender = new Appliance;
```

A class definition creates several attributes and functions pertinent to a data structure, in this case a data structure named Appliance. There is only one attribute, power, which can be modified by using the method setPower().

Remember, however, that a class definition is a template and cannot itself be manipulated. Instead, objects are created based on this template. This is accomplished via the new keyword. Therefore, in the last line of the previous listing, an object of class Appliance named blender is created.

The blender object's power attribute can then be set by making use of the method setPower():

```
$blender->setPower("on");
```

Converting Between Data Types Using Type Casting

Converting values from one datatype to another is known as *type casting*. A variable can be evaluated once as a different type by casting it to another. This is accomplished by placing the intended type in front of the variable to be cast.

Cast Operators	Conversion
(array)	Array
(bool) or (boolean)	Boolean
(int) or (integer)	Integer
(object)	Object
(real) or (double) or (float)	Float
(string)	String

Let's consider several examples. Suppose you'd like to cast an integer as a double:

```
$score = (double) 13; // $score = 13.0
```

Type casting a double to an integer will result in the integer value being rounded down, regardless of the decimal value. Here's an example:

```
$score = (int) 14.8; // $score = 14
```

What happens if you cast a string datatype to that of an integer? Let's find out:

```
$sentence = "This is a sentence";
echo (int) $sentence; // returns 0
```

While likely not the expected outcome, it's doubtful you'll want to cast a string like this anyway. You can also cast a datatype to be a member of an array. The value being cast simply becomes the first element of the array:

```
$score = 1114;
$scoreboard = (array) $score;
echo $scoreboard[0]; // Outputs 1114
```

One final example: any datatype can be cast as an object. The result is that the variable becomes an attribute of the object, the attribute having the name scalar:

```
$model = "Toyota";
$obj = (object) $model;
```

The value can then be referenced as follows:

```
print $obj->scalar; // returns "Toyota"
```

Adapting Data Types with Type Juggling

Because of PHP's lax attitude toward type definitions, variables are sometimes automatically cast to best fit the circumstances in which they are referenced. Consider the following snippet:

```
<?php
$total = 5; // an integer
$count = "15"; // a string
$total += $count; // $total = 20 (an integer)
?>
```

The outcome is the expected one; $total is assigned 20, converting the $count variable from a string to an integer in the process. Here's another example demonstrating PHP's type-juggling capabilities:

```
<?php
$total = "45 fire engines";
$incoming = 10;
$total = $incoming + $total; // $total = 55
?>
```

The integer value at the beginning of the original $total string is used in the calculation. However, if it begins with anything other than a numerical representation, the value is 0.

Consider another example:

```
<?php
$total = "1.0";
if ($total) echo "We're in positive territory!";
?>
```

In this example, a string is converted to Boolean type in order to evaluate the *if* statement. Consider one last particularly interesting example. If a string used in a mathematical calculation includes ., e, or E (representing scientific notation), it will be evaluated as a float:

```
<?php
$val1 = "1.2e3"; // 1,200
$val2 = 2;
echo $val1 * $val2; // outputs 2400
?>
```

Type-Related Functions

A few functions are available for both verifying and converting data types.

Retrieving Types

The gettype() function returns the type of the provided variable. In total, eight possible return values are available: array, boolean, double, integer, object, resource, string, and unknown type. Its prototype follows:

string gettype(mixed var)

Converting Types

The settype() function converts a variable to the type specified by type. Seven possible type values are available: array, boolean, float, integer, null, object, and string. If the conversion is successful, TRUE is returned; otherwise, FALSE is returned. Its prototype follows:

boolean settype(mixed var, string type)

Type Identifier Functions

A number of functions are available for determining a variable's type, including is_array(), is_bool(),is_float(), is_integer(), is_null(), is_numeric(), is_object(), is_resource(), is_scalar(), and is_string(). Because all of these functions follow the same naming convention, arguments, and return values, their introduction is consolidated into a single example. The generalized prototype follows:

boolean is_name(mixed var)

All of these functions are grouped in this section because each ultimately accomplishes the same task. Each determines whether a variable, specified by var, satisfies a particular condition specified by the function name. If var is indeed of the type tested by the function name, TRUE is returned; otherwise, FALSE is returned. An example follows:

```
<?php
$item = 43;
printf("The variable \$item is of type array: %d <br />", is_array($item));
printf("The variable \$item is of type integer: %d <br />", is_integer($item));
printf("The variable \$item is numeric: %d <br />", is_numeric($item));
?>
```

This code returns the following:

The variable $item is of type array: 0
The variable $item is of type integer: 1
The variable $item is numeric: 1

Identifiers

Identifier is a general term applied to variables, functions, and various other user-defined objects. There are several properties that PHP identifiers must abide by:
An identifier can consist of one or more characters and must begin with a letter or an underscore. Furthermore, identifiers can consist of only letters, numbers, underscore characters, and other ASCII characters from 127 through 255.

Valid	**Invalid**
my_function	This&that
Size	!counter
_someword	4ward

- Identifiers are case sensitive. Therefore, a variable named $recipe is different from a variable named $Recipe, $rEciPe, or $recipE.
- Identifiers can be any length. This is advantageous because it enables a programmer to accurately describe the identifier's purpose via the identifier name.
- An identifier name can't be identical to any of PHP's predefined keywords.

Variables

A variable is a symbol that can store different values at different times. For example, suppose you create a web-based calculator capable of performing mathematical tasks.

Variable Declaration

A variable always begins with a dollar sign, $, which is then followed by the variable name. Variable names follow the same naming rules as identifiers. That is, a variable name can begin with either a letter or an underscore and can consist of letters, underscores, numbers, or other ASCII characters rangingfrom 127 through 255. The following are all valid variables:

- $color
- $operating_system
- $_some_variable

- $model

Note that variables are case sensitive. For instance, the following variables bear no relation to one another:

- $color
- $Color
- $COLOR

Value Assignment

Assignment by value simply involves copying the value of the assigned expression to the variable assignee. This is the most common type of assignment. A few examples follow:

```
$color = "red";
$number = 12;
$age = 12;
$sum = 12 + "15"; // $sum = 27
```

.

Reference Assignment

PHP 4 introduced the ability to assign variables by reference, which essentially means that you can create a variable that refers to the same content as another variable does. Therefore, a change to any variable referencing a particular item of variable content will be reflected among all other variables referencing that same content. You can assign variables by reference by appending an ampersand (&) to the equal sign. Let's consider an example:

```
<?php
$value1 = "Hello";
$value2 =& $value1; // $value1 and $value2 both equal "Hello"
$value2 = "Goodbye"; // $value1 and $value2 both equal "Goodbye"
?>
```

An alternative reference-assignment syntax is also supported, which involves appending the ampersand to the front of the variable being referenced. The following example adheres to this new syntax:

```
<?php
$value1 = "Hello";
$value2 = &$value1; // $value1 and $value2 both equal "Hello"
$value2 = "Goodbye"; // $value1 and $value2 both equal "Goodbye"
?>
```

Variable Scope

Scope can be defined as the range of availability a variable has to the program in which it is declared. PHP variables can be one of four scope types −

- Local variables
- Function parameters
- Global variables
- Static variables

Local Variables

A variable declared in a function is considered local; that is, it can be referenced solely in that function. Any assignment outside of that function will be considered to be an entirely different variable from the one contained in the function −

```
<?php
  $x = 4;

  function assignx () {
    $x = 0;
    print "\$x inside function is $x. <br />";
  }

  assignx();
  print "\$x outside of function is $x. <br />";
?>
```

This will produce the following result −

```
$x inside function is 0.
$x outside of function is 4.
```

Function Parameters

Function parameters are declared after the function name and inside parentheses. They are declared much like a typical variable would be −

```
<?php
  // multiply a value by 10 and return it to the caller
  function multiply ($value) {
    $value = $value * 10;
    return $value;
  }

  $retval = multiply (10);
  Print "Return value is $retval\n";
?>
```

This will produce the following result −

```
Return value is 100
```

Global Variables

In contrast to local variables, a global variable can be accessed in any part of the program. However, in order to be modified, a global variable must be explicitly declared to be global in the function in which it is to be modified. This is accomplished, conveniently enough, by placing the keyword **GLOBAL** in front of the variable that should be recognized as global. Placing this keyword in front of an already existing variable tells PHP to use the variable having that name. Consider an example −

```
<?php
  $somevar = 15;
```

```
  function addit() {
    GLOBAL $somevar;
    $somevar++;

    print "Somevar is $somevar";
  }

  addit();
?>
```

This will produce the following result −

Somevar is 16

Static Variables

The final type of variable scoping that I discuss is known as static. In contrast to the variables declared as function parameters, which are destroyed on the function's exit, a static variable will not lose its value when the function exits and will still hold that value should the function be called again.

You can declare a variable to be static simply by placing the keyword STATIC in front of the variable name.

```
<?php
  function keep_track() {
    STATIC $count = 0;
    $count++;
    print $count;
    print "<br />";
  }

  keep_track();
  keep_track();
  keep_track();
?>
```

This will produce the following result –

1

2

3

PHP’s Superglobal Variables

PHP offers a number of useful predefined variables that are accessible from anywhere within the executing script and provide you with a substantial amount of environment-specific information. You can sift through these variables to retrieve details about the current user session, the user's operating environment, the local operating environment, and more. PHP creates some of the variables, while the availability and value of many of the other variables are specific to the operating system and web server.
Therefore, rather than attempt to assemble a comprehensive list of all possible predefined variables and their possible values, the following code will output all predefined variables pertinent to any given web server and the script's execution environment:

```
foreach ($_SERVER as $var => $value) {
echo "$var => $value <br />";
}
```

Sr.No	Variable & Description
1	**$_GLOBALS:** Contains a reference to every variable which is currently available within the global scope of the script. The keys of this array are the names of the global variables.
2	**$_SERVER:** This is an array containing information such as headers, paths, and script locations. The entries in this array are created by the web server. There is no guarantee that every web server will provide any of these. See next section for a complete list of all the SERVER variables.
3	**$_GET:** An associative array of variables passed to the current script via the HTTP GET method.
4	**$_POST:** An associative array of variables passed to the current script via the HTTP POST method.
5	**$_FILES:** An associative array of items uploaded to the current script via the HTTP POST method.
6	**$_REQUEST:** An associative array consisting of the contents of $_GET, $_POST, and $_COOKIE.
7	**$_COOKIE:** An associative array of variables passed to the current script via HTTP cookies.
8	**$_SESSION:** An associative array containing session variables available to the current script.
9	**$_PHP_SELF:** A string containing PHP script file name in which it is called.
10	**$php_errormsg:** $php_errormsg is a variable containing the text of the last error message generated by PHP.

Variable Variables

On occasion, you may want to use a variable whose content can be treated dynamically as a variable in itself. Consider this typical variable assignment:

```
$recipe = "spaghetti";
```

Interestingly, you can treat the value spaghetti as a variable by placing a second dollar sign in front of the original variable name and again assigning another value:

```
$$recipe = "& meatballs";
```

This in effect assigns *& meatballs* to a variable named spaghetti. Therefore, the following two snippets of code produce the same result:

```
echo $recipe $spaghetti;
echo $recipe ${$recipe};
```

The result of both is the string *spaghetti & meatballs*.

Constants

A *constant* is a value that cannot be modified throughout the execution of a program. Constants are particularly useful when working with values that definitely will not require modification, such as Pi (3.141592) or the number of feet in a mile (5,280). Once a constant has been defined, it cannot be changed (or redefined) at any other point of the program. Constants are defined using the define() function.

Defining a Constant

The define() function defines a constant by assigning a value to a name. Its prototype follows:

boolean define(string name, mixed value [, bool case_insensitive])

If the optional parameter *case_insensitive* is included and assigned TRUE, subsequent references to the constant will be case insensitive. Consider the following example in which the mathematical constant Pi is defined:

define("PI", 3.141592);

The constant is subsequently used in the following listing:

```
printf("The value of Pi is %f", PI);
$pi2 = 2 * PI;
printf("Pi doubled equals %f", $pi2);
```

This code produces the following results:

The value of pi is 3.141592.
Pi doubled equals 6.283184.

Expressions

An *expression* is a phrase representing a particular action in a program. All expressions consist of at least one operand and one or more operators.
A few examples follow:

```
$a = 5; // assign integer value 5 to the variable $a
$a = "5"; // assign string value "5" to the variable $a
$sum = 50 + $some_int; // assign sum of 50 + $some_int to $sum
$wine = "Zinfandel"; // assign "Zinfandel" to the variable $wine
$inventory++; // increment the variable $inventory by 1
```

Operands

Operands are the inputs of an expression. You might already be familiar with the manipulation and use of operands not only through everyday mathematical calculations, but also through prior programming experience. Some examples of operands follow:

```
$a++; // $a is the operand
$sum = $val1 + val2; // $sum, $val1 and $val2 are operands
```

Operators

An *operator* is a symbol that specifies a particular action in an expression. Many operators may be familiar to you. Regardless, you should remember that PHP's automatic type conversion will convert types based on the type of operator placed between the two operands, which is not always the case in other programming languages.
The precedence and associativity of operators are significant characteristics of a programming language.

Operator Precedence, Associativity, and Purpose

Operator	Associativity	Purpose
new	NA	Object instantiation
( )	NA	Expression subgrouping
[ ]	Right	Index enclosure
! ~ ++ --	Right	Boolean NOT, bitwise NOT, increment, decrement
@	Right	Error suppression
/ * %	Left	Division, multiplication, modulus
+ - .	Left	Addition, subtraction, concatenation
<< >>	Left	Shift left, shift right (bitwise)

< <= > >=	NA	Less than, less than or equal to, greater than, greater than or equal to
== != === <>	NA	Is equal to, is not equal to, is identical to, is not equal to
& ^ |	Left	Bitwise AND, bitwise XOR, bitwise OR
&& ||	Left	Boolean AND, Boolean OR
?:	Right	Ternary operator
= += *= /= .= %=&= |= ^= <<= >>=	Right	Assignment operators
AND XOR OR	Left	Boolean AND, Boolean XOR, Boolean OR
,	Left	Expression separation

Operator Precedence

Operator precedence is a characteristic of operators that determines the order in which they evaluate the operands surrounding them. PHP follows the standard precedence rules used in elementary school math class. Consider a few examples:

```
$total_cost = $cost + $cost * 0.06;
```

This is the same as writing

```
$total_cost = $cost + ($cost * 0.06);
```

Because the multiplication operator has higher precedence than the addition operator.

Operator Associativity

The *associativity* characteristic of an operator specifies how operations of the same precedence are evaluated as they are executed. Associativity can be performed in two directions, left-to-right or right-to-left. Left-to-right associativity means that the various operations making up the expression are evaluated from left to right.

Consider the following example:

```
$value = 3 * 4 * 5 * 7 * 2;
```

The preceding example is the same as the following:

```
$value = ((((3 * 4) * 5) * 7) * 2);
```

Arithmetic Operators

There are following arithmetic operators supported by PHP language −

Assume variable A holds 10 and variable B holds 20 then −

Operator	Description	Example
+	Adds two operands	A + B will give 30
-	Subtracts second operand from the first	A - B will give -10

*	Multiply both operands	A * B will give 200
/	Divide numerator by de-numerator	B / A will give 2
%	Modulus Operator and remainder of after an integer division	B % A will give 0
++	Increment operator, increases integer value by one	A++ will give 11
--	Decrement operator, decreases integer value by one	A-- will give 9

Assignment Operators

There are following assignment operators supported by PHP language −

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	C = A + B will assign value of A + B into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	C %= A is equivalent to C = C % A

String Operators

PHP's *string operators* provide a convenient way in which to concatenate strings together. There are two such operators, including the concatenation operator (.) and the concatenation assignment operator (.=).

Example	Label	Outcome
$a = "abc"."def";	Concatenation	$a is assigned the string "abcdef"
$a .= "ghijkl";	Concatenation-assignment	$a equals its current value concatenated with "ghijkl"

Increment and Decrement Operators

The *increment* (++) and *decrement* (--) operators present a minor convenience in terms of code clarity, providing shortened means by which you can add 1 to or subtract 1 from the current value of a variable.

Example	Label	Outcome
++$a, $a++	Increment	Increment $a by 1
--$a, $a--	Decrement	Decrement $a by 1

Logical Operators

There are following logical operators supported by PHP language

Assume variable A holds 10 and variable B holds 20 then −

Operator	Description	Example
and	Called Logical AND operator. If both the operands are true then condition becomes true.	(A and B) is true.
or	Called Logical OR Operator. If any of the two operands are non zero then condition becomes true.	(A or B) is true.
&&	Called Logical AND operator. If both the operands are non zero then condition becomes true.	(A && B) is true.
||	Called Logical OR Operator. If any of the two operands are non zero then condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is false.

Equality Operators

Equality operators are used to compare two values, testing for equivalence.

Example	Label	Outcome
$a == $b	Is equal to	True if $a and $b are equivalent
$a != $b	Is not equal to	True if $a is not equal to $b
$a === $b	Is identical to	True if $a and $b are equivalent and $a and $b have the same type

Comparison Operators

There are following comparison operators supported by PHP language

Assume variable A holds 10 and variable B holds 20 then −

Operator	Description	Example
==	Checks if the value of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

Bitwise Operators

Bitwise operators examine and manipulate integer values on the level of individual bits that make up the

integer value (thus the name).

<table>
<tr><th>Example</th><th>Label</th><th>Outcome</th></tr>
<tr><td>$a & $b</td><td>AND</td><td>And together each bit contained in $a and $b</td></tr>
<tr><td>$a | $b</td><td>OR</td><td>Or together each bit contained in $a and $b</td></tr>
<tr><td>$a ^ $b</td><td>XOR</td><td>Exclusive—or together each bit contained in $a and $b</td></tr>
<tr><td>~ $b</td><td>NOT</td><td>Negate each bit in $b</td></tr>
<tr><td>$a << $b</td><td>Shift left $a</td><td>will receive the value of $b shifted left two bits</td></tr>
<tr><td>$a >> $b</td><td>Shift right $a</td><td>will receive the value of $b shifted right two bits</td></tr>
</table>

String Interpolation

To offer developers the maximum flexibility when working with string values, PHP offers a means for both literal and figurative interpretation. For example, consider the following string:

The $animal jumped over the wall.\n

You might assume that $animal is a variable and that \n is a newline character, and therefore both should be interpreted accordingly. However, what if you want to output the string exactly as it is written, or perhaps you want the newline to be rendered but want the variable to display in its literal form ($animal), or vice versa? All of these variations are possible in PHP, depending on how the strings are enclosed and whether certain key characters are escaped through a predefined sequence.

Double Quotes

Strings enclosed in double quotes are the most commonly used in PHP scripts because they offer the most flexibility. This is because both variables and escape sequences will be parsed accordingly.

Consider the following example:

```
<?php
$sport = "boxing";
echo "Jason's favorite sport is $sport.";
?>
```

This example returns the following:
Jason's favorite sport is boxing.

Escape Sequences

Escape sequences are also parsed. Consider this example:

```
<?php
$output = "This is one line.\nAnd this is another line.";
echo $output;
?>
```

This returns the following (as viewed from within the browser source):

```
This is one line.
And this is another line.
```

Single Quotes

Enclosing a string within single quotes is useful when the string should be interpreted exactly as stated. This means that both variables and escape sequences will not be interpreted when the string is parsed.

For example, consider the following single-quoted string:

```
print 'This string will $print exactly as it\'s \n declared.';
```

This produces the following:

This string will $print exactly as it's \n declared.

Curly Braces

While PHP is perfectly capable of interpolating variables representing scalar data types, you'll find that variables representing complex data types such as arrays or objects cannot be so easily parsed when embedded in an echo() or print() string. You can solve this issue by delimiting the variable in curly braces, like this:

```
echo "The capital of Ohio is {$capitals['ohio']}.";
```

Heredoc

Heredoc syntax offers a convenient means for outputting large amounts of text. Rather than delimiting strings with double or single quotes, two identical identifiers are employed. An example follows:

```
<?php
$website = "http://www.romatermini.it";
echo <<<EXCERPT
<p>Rome's central train station, known as <a href = "$website">Roma Termini</a>,
was built in 1867. Because it had fallen into severe disrepair in the late 20th
century, the government knew that considerable resources were required to
rehabilitate the station prior to the 50-year <i>Giubileo</i>.</p>
EXCERPT;
?>
```

Several points are worth noting regarding this example:

• The opening and closing identifiers (in the case of this example, EXCERPT) must be identical.
• The opening identifier must be preceded with three left-angle brackets (<<<).
• Heredoc syntax follows the same parsing rules as strings enclosed in double quotes. That is, both variables and escape sequences are parsed. The only difference is that double quotes do not need to be escaped.
• The closing identifier must begin at the very beginning of a line. It cannot be preceded with spaces or any other extraneous character.

Nowdoc

Introduced in PHP 5.3, *nowdoc* syntax operates identically to heredoc syntax, except that none of the text delimited within a nowdoc is parsed. If you would like to display, for instance, a snippet of code in the browser, you could embed it within a nowdoc statement; when subsequently outputting the nowdoc variable, you can be sure that PHP will not attempt to interpolate any of the string as code.

Control Structures

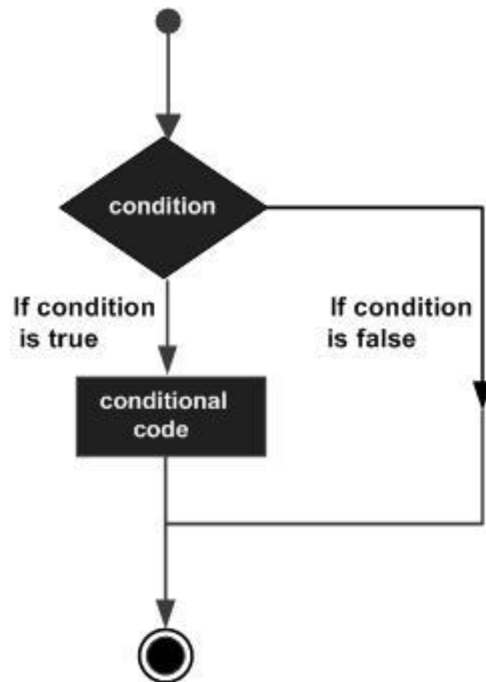
Control structures determine the flow of code within an application, defining execution characteristics such as whether and how many times a particular code statement will execute, as well as when a code block will relinquish execution control.

Conditional Statements

Conditional statements make it possible for your computer program to respond accordingly to a wide variety of inputs, using logic to discern between various conditions based on input value.

You can use conditional statements in your code to make your decisions. PHP supports following three decision making statements −

- **if...else statement** − use this statement if you want to execute a set of code when a condition is true and another if the condition is not true
- **elseif statement** − is used with the if...else statement to execute a set of code if **one** of the several condition is true
- **switch statement** − is used if you want to select one of many blocks of code to be executed, use the Switch statement. The switch statement is used to avoid long blocks of if..elseif..else code.



The If...Else Statement

If you want to execute some code if a condition is true and another code if a condition is false, use the if....else statement.

Syntax

if (*condition*)
  code to be executed if condition is true;
else
  code to be executed if condition is false;

Example

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

```
<html>
  <body>

    <?php
      $d = date("D");

      if ($d == "Fri")
        echo "Have a nice weekend!";

      else
        echo "Have a nice day!";
    ?>
```

```
  </body>
</html>
```

It will produce the following result −

Have a nice day!

The ElseIf Statement

If you want to execute some code if one of the several conditions are true use the elseif statement

Syntax

```
if (condition)
  code to be executed if condition is true;
elseif (condition)
  code to be executed if condition is true;
else
  code to be executed if condition is false;
```

Example

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!" −

```
<html>
  <body>

    <?php
      $d = date("D");

      if ($d == "Fri")
        echo "Have a nice weekend!";

      elseif ($d == "Sun")
        echo "Have a nice Sunday!";

      else
        echo "Have a nice day!";
    ?>

  </body>
</html>
```

It will produce the following result −

Have a nice day!

The Switch Statement

If you want to select one of many blocks of code to be executed, use the Switch statement.

The switch statement is used to avoid long blocks of if..elseif..else code.

Syntax

```
switch (expression){
  case label1:
    code to be executed if expression = label1;
    break;

  case label2:
    code to be executed if expression = label2;
    break;
    default:

  code to be executed
  if expression is different
  from both label1 and label2;
}
```

Example

The *switch* statement works in an unusual way. First it evaluates given expression then seeks a lable to match the resulting value. If a matching value is found then the code associated with the matching label will be executed or if none of the lable matches then statement will execute any specified default code.

```
<html>
  <body>

    <?php
      $d = date("D");

      switch ($d){
        case "Mon":
          echo "Today is Monday";
          break;

        case "Tue":
          echo "Today is Tuesday";
          break;

        case "Wed":
          echo "Today is Wednesday";
          break;

        case "Thu":
          echo "Today is Thursday";
```

```
            break;

         case "Fri":
            echo "Today is Friday";
            break;

         case "Sat":
            echo "Today is Saturday";
            break;

         case "Sun":
            echo "Today is Sunday";
            break;

         default:
            echo "Wonder which day is this ?";
      }
   ?>

   </body>
</html>
```

It will produce the following result −

Today is Wednesday

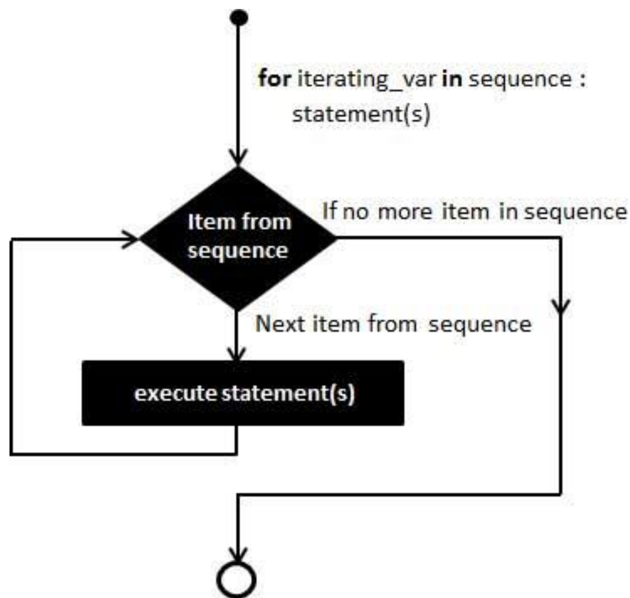
Looping Statements

Loops in PHP are used to execute the same block of code a specified number of times. PHP supports following four loop types.

- **for** − loops through a block of code a specified number of times.
- **while** − loops through a block of code if and as long as a specified condition is true.
- **do...while** − loops through a block of code once, and then repeats the loop as long as a special condition is true.
- **foreach** − loops through a block of code for each element in an array.
-

The for loop statement

- The for statement is used when you know how many times you want to execute a statement or a block of statements.



Syntax

```
for (initialization; condition; increment){
  code to be executed;
}
```

The initializer is used to set the start value for the counter of the number of loop iterations. A variable may be declared here for this purpose and it is traditional to name it $i.

Example

The following example makes five iterations and changes the assigned value of two variables on each pass of the loop –

```
<html>
  <body>

    <?php
      $a = 0;
      $b = 0;

      for( $i = 0; $i<5; $i++ ) {
        $a += 10;
        $b += 5;
      }

      echo ("At the end of the loop a = $a and b = $b" );
    ?>

  </body>
</html>
```

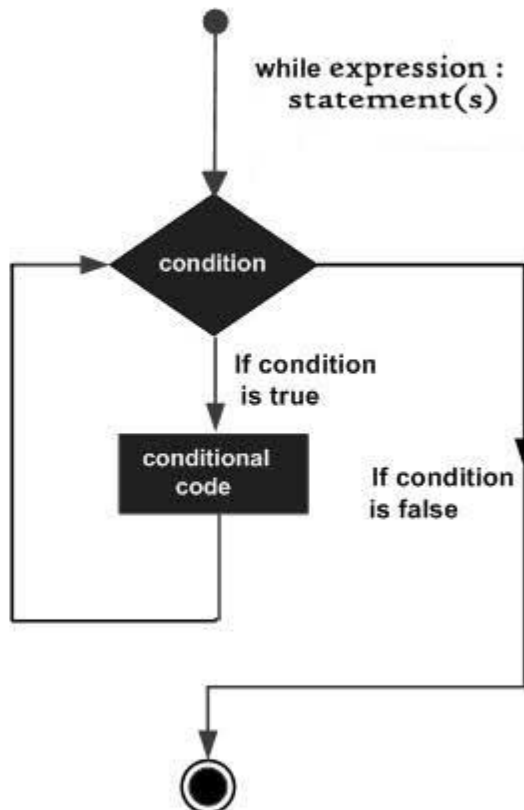
This will produce the following result –

```
At the end of the loop a = 50 and b = 25
```

The while loop statement

The while statement will execute a block of code if and as long as the test expression is true.

If the test expression is true then the code block will be executed. After the code has executed the test expression will again be evaluated and the loop will continue until the test expression is found to be false.



Syntax

```
while (condition) {
  code to be executed;
}
```

Example

This example decrements a variable value on each iteration of the loop and the counter increments until it reaches 10 when the evaluation is false and the loop ends.

```
<html>
  <body>

    <?php
      $i = 0;
      $num = 50;
```

```
      while( $i < 10) {
        $num--;
        $i++;
      }

      echo ("Loop stopped at i = $i and num = $num" );
    ?>

  </body>
</html>
```

This will produce the following result –

Loop stopped at i = 10 and num = 40

The do...while loop statement

The do...while statement will execute a block of code at least once - it then will repeat the loop as long as a condition is true.

Syntax

```
do {
  code to be executed;
}
while (condition);
```

Example

The following example will increment the value of i at least once, and it will continue incrementing the variable i as long as it has a value of less than 10 –

```
<html>
  <body>

    <?php
      $i = 0;
      $num = 0;

      do {
        $i++;
      }

      while( $i < 10 );
      echo ("Loop stopped at i = $i" );
    ?>

  </body>
</html>
```

This will produce the following result −

```
Loop stopped at i = 10
```

The foreach loop statement

The foreach statement is used to loop through arrays. For each pass the value of the current array element is assigned to $value and the array pointer is moved by one and in the next pass next element will be processed.

Syntax

```
foreach (array as value) {
  code to be executed;
}
```

Example

Try out following example to list out the values of an array.

```
<html>
  <body>

    <?php
      $array = array( 1, 2, 3, 4, 5);

      foreach( $array as $value ) {
        echo "Value is $value <br />";
      }
    ?>

  </body>
</html>
```

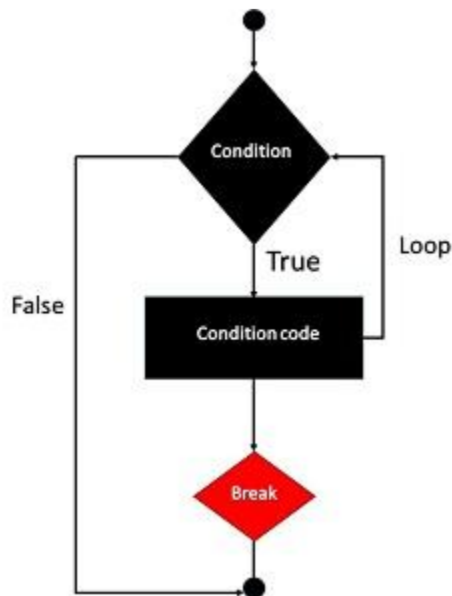
This will produce the following result –

```
Value is 1
Value is 2
Value is 3
Value is 4
Value is 5
```

The break statement

The PHP **break** keyword is used to terminate the execution of a loop prematurely.

The **break** statement is situated inside the statement block. If gives you full control and whenever you want to exit from the loop you can come out. After coming out of a loop immediate statement to the loop will be executed.



Example

In the following example condition test becomes true when the counter value reaches 3 and loop terminates.

```
<html>
  <body>

    <?php
      $i = 0;

      while( $i < 10) {
        $i++;
        if( $i == 3 )break;
      }
      echo ("Loop stopped at i = $i" );
    ?>

  </body>
</html>
```

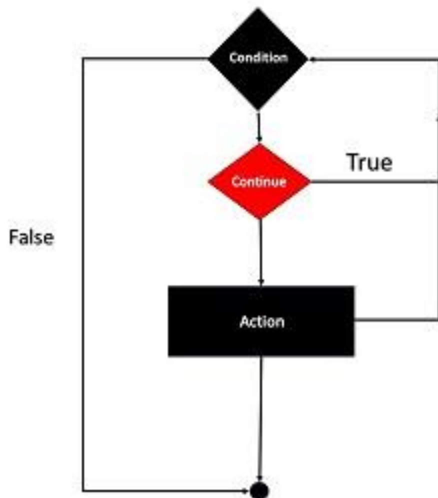
This will produce the following result –

```
Loop stopped at i = 3
```

The continue statement

The PHP **continue** keyword is used to halt the current iteration of a loop but it does not terminate the loop.

Just like the **break** statement the **continue** statement is situated inside the statement block containing the code that the loop executes, preceded by a conditional test. For the pass encountering **continue** statement, rest of the loop code is skipped and next pass starts.



Example

In the following example loop prints the value of array but for which condition becomes true it just skip the code and next value is printed.

```
<html>
  <body>

    <?php
      $array = array( 1, 2, 3, 4, 5);

      foreach( $array as $value ) {
        if( $value == 3 )continue;
        echo "Value is $value <br />";
      }
    ?>

  </body>
</html>
```

This will produce the following result –

Value is 1
Value is 2
Value is 4
Value is 5

3. File Inclusion statements

You can include the content of a PHP file into another PHP file before the server executes it. There are two PHP functions which can be used to included one PHP file into another PHP file.

- The include() Function
- The require() Function

This is a strong point of PHP which helps in creating functions, headers, footers, or elements that can be reused on multiple pages. This will help developers to make it easy to change the layout of complete website with minimal effort. If there is any change required then instead of changing thousand of files just change included file.

a.The include() Function

The include() function takes all the text in a specified file and copies it into the file that uses the include function. If there is any problem in loading a file then the **include()** function generates a warning but the script will continue execution.

Assume you want to create a common menu for your website. Then create a file menu.php with the following content.

```
<a href="http://www.tutorialspoint.com/index.htm">Home</a> -
<a href="http://www.tutorialspoint.com/ebxml">ebXML</a> -
<a href="http://www.tutorialspoint.com/ajax">AJAX</a> -
<a href="http://www.tutorialspoint.com/perl">PERL</a> <br />
```

Now create as many pages as you like and include this file to create header. For example now your test.php file can have following content.

```
<html>
  <body>

    <?php include("menu.php"); ?>
    <p>This is an example to show how to include PHP file!</p>

  </body>
</html>
```

It will produce the following result –

Home -

ebXML -

AJAX -

PERL

b. Ensuring a File is included only once:

The include_once() function verified whether the file has already been included.

include_once(filename);

If a file has already been included; include_once will not execute. Otherwise, it will inluce the file as necessary.

c. The require() Function

The require() function takes all the text in a specified file and copies it into the file that uses the include function. If there is any problem in loading a file then the **require()** function generates a fatal error and halt the execution of the script.

So there is no difference in require() and include() except they handle error conditions. It is recommended to use the require() function instead of include(), because scripts should not continue executing if files are missing or misnamed.

You can try using above example with require() function and it will generate same result. But if you will try following two examples where file does not exist then you will get different results.

```
<html>
  <body>

    <?php include("xxmenu.php"); ?>
    <p>This is an example to show how to include wrong PHP file!</p>

  </body>
</html>
```

This will produce the following result −

```
This is an example to show how to include wrong PHP file!
<html>
  <body>

     <?php require("xxmenu.php"); ?>
     <p>This is an example to show how to include wrong PHP file!</p>

  </body>
</html>
```

Ensuring a File Is Required Only Once

As your site grows, you may find yourself redundantly including certain files. Although this might not always be a problem, sometimes you will not want modified variables in the included file to be overwritten by a later inclusion of the same file. Another problem that arises is the

clashing of function names should they exist in the inclusion file. You can solve these problems with the require_once() function. Its prototype follows:
require_once (filename)

The require_once() function ensures that the inclusion file is included only once in your script. After require_once() is encountered, any subsequent attempts to include the same file will be ignored.

Functions:

Creating a Function

Although PHP's vast assortment of function libraries is a tremendous benefit to anybody seeking to avoid reinventing the programmatic wheel, sooner or later you'll need to go beyond what is offered in the standard distribution, which means you'll need to create custom functions or even entire function libraries. To do so, you'll need to define a function using PHP's supported syntax, which when written in pseudocode looks like this:

```
function functionName(parameters)
{
function-body
}
```

For example, consider the following function, generateFooter(), which outputs a page footer:

```
function generateFooter()
{
echo "Department of Computer Science and Engineering";
}
```

Once defined, you can call this function like so:

```
<?php
generateFooter();
?>
```

This yields the following result:
Department of Computer Science and Engineering

Passing Arguments by Value

You'll often find it useful to pass data into a function. As an example, let's create a function that calculates an item's total cost by determining its sales tax and then adding that amount to the price:

```
function calcSalesTax($price, $tax)
{
$total = $price + ($price * $tax);
echo "Total cost: $total";
```

```
}
```

This function accepts two parameters, aptly named $price and $tax, which are used in the calculation. Although these parameters are intended to be floating points, because of PHP's weak typing, nothing prevents you from passing in variables of any datatype, but the outcome might not be what you expect. In addition, you're allowed to define as few or as many parameters as you deem necessary; there are no language-imposed constraints in this regard.Once defined, you can then invoke the function as demonstrated

calcSalesTax() function would be called like so:

```
calcSalesTax(15.00, .075);
```

Passing Arguments by Reference

On occasion, you may want any changes made to an argument within a function to be reflected outside of the function's scope. Passing the argument by reference accomplishes this. Passing an argument by reference is done by appending an ampersand to the front of the argument. Here's an example:

```
<?php
$cost = 20.99;
$tax = 0.0575;
function calculateCost(&$cost, $tax)
{
// Modify the $cost variable
$cost = $cost + ($cost * $tax);
// Perform some random change to the $tax variable.
$tax += 4;
}
calculateCost($cost, $tax);
printf("Tax is %01.2f%% ", $tax*100);
printf("Cost is: $%01.2f", $cost);
?>
```

Here's the result:

```
Tax is 5.75%
Cost is $22.20
```

Default Argument Values

Default values can be assigned to input arguments, which will be automatically assigned to the argument if no other value is provided. You could then assign $tax the default value of 6.75 percent, like this:

```
function calcSalesTax($price, $tax=.0675)
{
$total = $price + ($price * $tax);
echo "Total cost: $total";
}
```

You can still pass $tax another taxation rate; 6.75 percent will be used only if calcSalesTax() is

invoked without the second parameter like this:

```
$price = 15.47;
calcSalesTax($price);
```

Using Type Hinting

PHP 5 introduced a new feature known as *type hinting*, which gives you the ability to force parameters to be objects of a certain class or to be arrays. Unfortunately, type hinting using scalar data types such as integers and strings is not supported. If the provided parameter is not of the desired type, a fatal error will occur.

Returning Values from a Function

Often, simply relying on a function to do something is insufficient; a script's outcome might depend on a function's outcome or on changes in data resulting from its execution. Yet variable scoping prevents information from easily being passed from a function body back to its caller, so how can we accomplish this? You can pass data back to the caller by way of the return() statement.

The return Statement

The return() statement returns any ensuing value back to the function caller, returning program control back to the caller's scope in the process. If return() is called from within the global scope, the script execution is terminated.

```
function calcSalesTax($price, $tax=.0675)
{
$total = $price + ($price * $tax);
return $total;
}
```

Returning Multiple Values

It's often convenient to return multiple values from a function. For example, suppose that you'd like to create a function that retrieves user data from a database (say the user's name, e-mail address, and phone number) and returns it to the caller. Accomplishing this is much easier than you might think, with the help of a very useful language construct, list(). The list() construct offers a convenient means for retrieving values from an array, like so:

```
<?php
function retrieveUserProfile()
{
$user[] = "Arifa Tehseen";
$user[] = "arifa@jbiet.edu.in";
$user[] = "English";
return $user;
}
list($name, $email, $language) = retrieveUserProfile();
```

```
echo "Name: $name, email: $email, language: $language";
?>
```

Executing this script returns the following:
Name: Arifa Tehseen, email:arifa@jbiet.edu.in, language: English

Recursive Functions
Recursive functions, or functions that call themselves, offer considerable practical value to the programmer and are used to divide an otherwise complex problem into a simple case, reiterating that case until the problem is resolved

Function Libraries
Great programmers are lazy, and lazy programmers think in terms of reusability. Functions offer a great way to reuse code and are often collectively assembled into libraries and subsequently repeatedly reused within similar applications. PHP libraries are created via the simple aggregation of function definitions in a single file, like this:

```
<?php
function localTax($grossIncome, $taxRate) {
// function body here
}
function stateTax($grossIncome, $taxRate, $age) {
// function body here
}
function medicare($grossIncome, $medicareRate) {
// function body here
}
?>
```

Arrays

What Is an Array?
PHP takes this definition a step further, forgoing the requirement that the items share the same data type. For example, an array could quite possibly contain items such as state names, ZIP codes, exam scores, or playing card suits.
Each item consists of two components: the aforementioned key and a value. The key serves as the lookup facility for retrieving its counterpart, the *value*. Keys can be *numerical* or *associative*. Numerical keys bear no real relation to the value other than the value's position in the array.

```
$states = array(0 => "Alabama", 1 => "Alaska"...49 => "Wyoming");
$states = array("OH" => "Ohio", "PA" => "Pennsylvania", "NY" => "New York")
```

Creating an Array
Unlike other languages, PHP doesn't require that you assign a size to an array at creation time. In fact, because it's a loosely typed language, PHP doesn't even require that you declare the array before using it.

$state[0] = "Delaware";

Interestingly, if you intend for the index value to be numerical and ascending, you can omit the index value at creation time:

```
$state[] = "Pennsylvania";
$state[] = "New Jersey";
...
$state[] = "Hawaii";
```

Creating Arrays with array()

The array() construct takes as its input zero or more items and returns an array consisting of these input elements. Its prototype looks like this:

array array([*item1* [,*item2* ... [,itemN]]])

Extracting Arrays with list()

The list() construct is similar to array(), though it's used to make simultaneous variable assignments from values extracted from an array in just one operation. Its prototype looks like this:

void list(mixed...)

This construct can be particularly useful when you're extracting information from a database or file.

Populating Arrays with a Predefined Value Range

The range() function provides an easy way to quickly create and fill an array consisting of a range of low and high integer values. An array containing all integer values in this range is returned. Its prototype looks like this:

array range(int *low*, int *high* [, int *step*])

For example, suppose you need an array consisting of all possible face values of a die:

```
$die = range(1, 6);
// Same as specifying $die = array(1, 2, 3, 4, 5, 6)
```

Testing for an Array

When you incorporate arrays into your application, you'll sometimes need to know whether a particular variable is an array. A built-in function, is_array(), is available for accomplishing this task. Its prototype follows:

boolean is_array(mixed *variable*)

The is_array() function determines whether variable is an array, returning TRUE if it is and FALSE otherwise.

An example follows:

```
$states = array("Florida");
$state = "Ohio";
printf("\$states is an array: %s <br />", (is_array($states) ? "TRUE" : "FALSE"));
```

```
printf("\$state is an array: %s <br />", (is_array($state) ? "TRUE" : "FALSE"));
```

Executing this example produces the following:
$states is an array: TRUE
$state is an array: FALSE

Outputting an Array
The most common way to output an array's contents is by iterating over each key and echoing the corresponding value. For instance, a foreach statement does the trick nicely:

```
$states = array("Ohio", "Florida", "Texas");
foreach ($states AS $state) {
echo "{$state}<br />";
}
```

Printing Arrays for Testing Purposes
The array contents in most of the previous examples have been displayed using comments. While this works great for instructional purposes, in the real world you'll need to know how to easily output their contents to the screen for testing purposes. This is most commonly done with the print_r() function. Its prototype follows:
boolean print_r(mixed *variable* [, boolean *return*])

The print_r() function accepts a variable and sends its contents to standard output, returning TRUE on success and FALSE otherwise.

Adding and Removing Array Elements
PHP provides a number of functions for both growing and shrinking an array. Some of these functions are provided as a convenience to programmers who wish to mimic various queue implementations (FIFO, LIFO, etc.), as reflected by their names (push, pop, shift, and unshift).

Adding a Value to the Front of an Array
The array_unshift() function adds elements to the front of the array. All preexisting numerical keys are modified to reflect their new position in the array, but associative keys aren't affected. Its prototype follows:
int array_unshift(array *array*, mixed *variable* [, mixed *variable*...])

The following example adds two states to the front of the $states array:

```
$states = array("Ohio", "New York");
array_unshift($states, "California", "Texas");
// $states = array("California", "Texas", "Ohio", "New York");
```

Adding a Value to the End of an Array
The array_push() function adds a value to the end of an array, returning the total count of elements in the array after the new value has been added. You can push multiple variables onto

the array simultaneously by passing these variables into the function as input parameters. Its prototype follows:
int array_push(array *array*, mixed *variable* [, mixed *variable*...])

The following example adds two more states onto the $states array:
```
$states = array("Ohio", "New York");
array_push($states, "California", "Texas");
// $states = array("Ohio", "New York", "California", "Texas");
```

Removing a Value from the Front of an Array
The array_shift() function removes and returns the first item found in an array. If numerical keys are used, all corresponding values will be shifted down, whereas arrays using associative keys will not be affected. Its prototype follows:
mixed array_shift(array *array*)
The following example removes the first state from the $states array:
```
$states = array("Ohio", "New York", "California", "Texas");
$state = array_shift($states);
// $states = array("New York", "California", "Texas")
// $state = "Ohio"
```

Removing a Value from the End of an Array
The array_pop() function removes and returns the last element from an array. Its prototype follows:
mixed array_pop(array *array*)
The following example removes the last state from the $states array:
```
$states = array("Ohio", "New York", "California", "Texas");
$state = array_pop($states);
// $states = array("Ohio", "New York", "California"
// $state = "Texas"
```

Locating Array Elements
The ability to efficiently sift through data is absolutely crucial in today's information-driven society.

Searching an Array	The in_array() function searches an array for a specific value, returning TRUE if the value is found and FALSE otherwise. Its prototype follows: **boolean in_array(mixed *needle*, array *haystack* [, boolean *strict*])**
Searching Associative Array Keys	The function array_key_exists() returns TRUE if a specified key is found in an array and FALSE otherwise. Its prototype follows: **boolean array_key_exists(mixed *key*, array *array*)**
Searching Associative Array Values	The array_search() function searches an array for a specified value, returning its key if located and FALSE otherwise. Its prototype follows: **mixed array_search(mixed *needle*, array *haystack* [, boolean *strict*])**
Retrieving Array Keys	The array_keys() function returns an array consisting of all keys located in an array. Its prototype follows: **array array_keys(array *array* [, mixed *search_value* [, boolean *preserve_keys*]])**

Retrieving Array Values	The array_values() function returns all values located in an array, automatically providing numeric indexes for the returned array. Its prototype follows: **array array_values(array *array*)**

Traversing Arrays

The need to travel across an array and retrieve various keys, values, or both is common, so it's not a surprise that PHP offers numerous functions suited to this need. Many of these functions do double duty: retrieving the key or value residing at the current pointer location, and moving the pointer to the next appropriate location.

Retrieving the Current Array Key	The key() function returns the key located at the current pointer position of the provided array. Its prototype follows: **mixed key(array *array*)**
Retrieving the Current Array Value	The current() function returns the array value residing at the current pointer position of the array. Its prototype follows: **mixed current(array *array*)**
Retrieving the Current Array Key and Value	The each() function returns the current key/value pair from the array and advances the pointer one position. Its prototype follows: **array each(array *array*)**

Moving the Array Pointer

Several functions are available for moving the array pointer.

Moving the Pointer to the Next Array Position	The next() function returns the array value residing at the position immediately following that of the current array pointer. Its prototype follows: **mixed next(array *array*)**
Moving the Pointer to the Previous Array Position	The prev() function returns the array value residing at the location preceding the current pointer location, or FALSE if the pointer resides at the first position in the array. Its prototype follows: **mixed prev(array *array*)**
Moving the Pointer to the First Array Position	The reset() function serves to set an array pointer back to the beginning of the array. Its prototype follows: **mixed reset(array *array*)**
Moving the Pointer to the Last Array Position	The end() function moves the pointer to the last position of an array, returning the last element. Its prototype follows: **mixed end(array *array*)**

Passing Array Values to a Function

The array_walk() function will pass each element of an array to the user-defined function. This is useful when you need to perform a particular action based on each array element. If you

intend to actually modify the array key/value pairs, you'll need to pass each key/value to the function as a reference. Its prototype follows:
boolean array_walk(array &*array*, callback *function* [, mixed *userdata*])

Determining Array Size and Uniqueness

A few functions are available for determining the number of total and unique array values.

Determining the Size of an Array	The count() function returns the total number of values found in an array. Its prototype follows: **integer count(array *array* [, int *mode*])**
Counting Array Value Frequency	The array_count_values() function returns an array consisting of associative key/value pairs. Its prototype follows: **array array_count_values(array *array*)**
Determining Unique Array Values	The array_unique() function removes all duplicate values found in an array, returning an array consisting of solely unique values. Its prototype follows: **array array_unique(array *array* [, int *sort_flags* = SORT_STRING])**

Sorting Arrays

Reversing Array Element Order	The array_reverse() function reverses an array's element order. Its prototype follows: **array array_reverse(array *array* [, boolean *preserve_keys*])**
Flipping Array Keys and Values	The array_flip() function reverses the roles of the keys and their corresponding values in an array. Its prototype follows: **array array_flip(array *array*)**
Sorting an Array	The sort() function sorts an array, ordering elements from lowest to highest value. Its prototype follows: **void sort(array *array* [, int *sort_flags*])**
Sorting an Array While Maintaining Key/Value Pairs	The asort() function is identical to sort(), sorting an array in ascending order, except that the key/value correspondence is maintained. Its prototype follows: **void asort(array *array* [, integer *sort_flags*])**
Sorting an Array in Reverse Order	The rsort() function is identical to sort(), except that it sorts array items in reverse (descending) order.Its prototype follows: **void rsort(array *array* [, int *sort_flags*])**
Sorting an Array in Reverse Order While Maintaining Key/Value Pairs	Like asort(), arsort() maintains key/value correlation. However, it sorts the array in reverse order. Its prototype follows: **void arsort(array *array* [, int *sort_flags*])**
Sorting an Array Naturally	The natsort() function is intended to offer a sorting mechanism comparable to the mechanisms that people normally use. Its prototype follows: **void natsort(array *array*)**
Case-Insensitive Natural Sorting	The function natcasesort() is functionally identical to natsort(), except that it is case insensitive: **void natcasesort(array *array*)**
Sorting an Array by Key Values	The ksort() function sorts an array by its keys, returning TRUE on success and FALSE otherwise. Its prototype follows: integer ksort(array *array* [, int *sort_flags*])
Sorting Array Keys in Reverse Order	The krsort() function operates identically to ksort(), sorting by key, except that it sorts in reverse (descending) order. Its prototype follows: **integer krsort(array *array* [, int *sort_flags*])**
Sorting According to User-Defined Criteria	The usort() function offers a means for sorting an array by using a user-defined comparison algorithm, embodied within a function. This is useful when you need to sort data in a fashion not offered by one of PHP's built-in sorting functions. Its prototype follows: **void usort(array *array*, callback *function_name*)**

Merging, Slicing, Splicing, and Dissecting Arrays

Merging Arrays	The array_merge() function merges arrays together, returning a single, unified array. The resulting array will begin with the first input array parameter, appending each subsequent array parameter in the order of appearance. Its prototype follows: **array array_merge(array *array1*, array *array2* [, array *arrayN*])**
Recursively Appending Arrays	The array_merge_recursive() function operates identically to array_merge(), joining two or more arrays together to form a single, unified array. The difference between the two functions lies in the way that this function behaves when a string key located in one of the input arrays already exists within the resulting array. Note that array_merge() will simply overwrite the preexisting key/value pair, replacing it with the one found in the current input array, while array_merge_recursive() will instead merge the values together, forming a new array with the preexisting key as its name. Its prototype follows: **array array_merge_recursive(array *array1*, array *array2* [, array arrayN])**
Combining Two Arrays	The array_combine() function produces a new array consisting of a submitted set of keys and corresponding values. Its prototype follows: **array array_combine(array *keys*, array *values*)** Both input arrays must be of equal size, and neither can be empty.
Slicing an Array	The array_slice() function returns a section of an array based on a starting and ending offset value. Its prototype follows: **array array_slice(array *array*, int *offset* [, int *length* [, boolean *preserve_keys*]])**
Splicing an Array	The array_splice() function removes all elements of an array found within a specified range, returning those removed elements in the form of an array. Its prototype follows: **array array_splice(array *array*, int *offset* [, int *length* [, array *replacement*]])**
Calculating an Array Intersection	The array_intersect() function returns a key-preserved array consisting only of those values present in the first array that are also present in each of the other input arrays. Its prototype follows: **array array_intersect(array *array1*, array *array2* [, *arrayN*])**
Calculating Associative Array Intersections	The function array_intersect_assoc() operates identically to array_intersect(), except that it also considers array keys in the comparison. Therefore, only key/value pairs located in the first array that arealso found in all other input arrays will be returned in the resulting array. Its prototype follows: **array array_intersect_assoc(array *array1*, array *array2* [, arrayN])**
Calculating Array Differences	Essentially the opposite of array_intersect(), the function array_diff() returns those values located in the first array that are not located in any of the subsequent arrays: **array array_diff(array *array1*, array *array2* [, arrayN])**
Calculating Associative Array Differences	The function array_diff_assoc() operates identically to array_diff(), except that it also considers array keys in the comparison. Therefore, only key/value pairs located in the first array but not appearing in any of the other input arrays will be returned in the result array. Its prototype follows: **array array_diff_assoc(array *array1*, array *array2* [, array *arrayN*])**

Strings and Regular Expressions

PHP has long supported two regular expression implementations known as Perl and POSIX.

Regular Expressions

Regular expressions provide the foundation for describing or matching data according to defined syntax rules. A regular expression is nothing more than a pattern of characters itself, matched against a certain parcel of text. This sequence may be a pattern with which you are already familiar, such as the word *dog,* or it may be a pattern with specific meaning in the context of the world of pattern matching, <(?)>.*<\ /.?>

Regular Expression Syntax (POSIX)

POSIX stands for *Portable Operating System Interface for Unix* and is representative of a set of standards originally intended for Unix-based operating systems. POSIX regular expression syntax is an attempt to standardize how regular expressions are implemented in many programming languages.

Brackets

Brackets ([]) are used to represent a list, or range, of characters to be matched. For instance, contrary to the regular expression php, which will locate strings containing the explicit string php, the regular expression [php] will find any string containing the character p or h. Several commonly used character ranges follow:

[0-9] matches any decimal digit from 0 through 9.

- [a-z] matches any character from lowercase a through lowercase z.
- [A-Z] matches any character from uppercase A through uppercase Z.
- [A-Za-z] matches any character from uppercase A through lowercase z.

Quantifiers

Sometimes you might want to create regular expressions that look for characters based on their frequency or position

- p+ matches any string containing at least one p.
- p* matches any string containing zero or more p's.
- p? matches any string containing zero or one p.
- p{2} matches any string containing a sequence of two p's.
- p{2,3} matches any string containing a sequence of two or three p's.
- p{2,} matches any string containing a sequence of at least two p's.
- p$ matches any string with p at the end of it.

Predefined Character Ranges (Character Classes)

For reasons of convenience, several predefined character ranges, also known as *character classes,* are available. Character classes specify an entire range of characters—for example, the alphabet or an integer set. Standard classes include the following:

[:alpha:]: Lowercase and uppercase alphabetical characters. This can also be specified as [A-Za-z].

[:alnum:]: Lowercase and uppercase alphabetical characters and numerical digits. This can also be specified as [A-Za-z0-9].

[:cntrl:]: Control characters such as tab, escape, or backspace.

[:digit:]: Numerical digits 0 through 9. This can also be specified as [0-9].

[:graph:]: Printable characters found in the range of ASCII 33 to 126.

[:lower:]: Lowercase alphabetical characters. This can also be specified as [a-z].

[:punct:]: Punctuation characters, including ~ ` ! @ # $ % ^ & * ( ) - _ + = { } [ ] : ; ' < > , . ? and /.

[:upper:]: Uppercase alphabetical characters. This can also be specified as [A-Z].

[:space:]: Whitespace characters, including the space, horizontal tab, vertical tab, new line, form feed, or carriage return.

[:xdigit:]: Hexadecimal characters. This can also be specified as [a-fA-F0-9].

PHP’s Regular Expression Functions (POSIX Extended)

PHP offers seven functions for searching strings using POSIX-style regular expressions: ereg(),ereg_replace(), eregi(), eregi_replace(), split(), spliti(), and sql_regcase().

Performing a Case-Sensitive Search	The ereg() function executes a case-sensitive search of a string for a defined pattern, returning the length of the matched string if the pattern is found and FALSE otherwise. Its prototype follows: **int ereg(string *pattern*, string *string* [, array *regs*])**
Performing a Case-Insensitive Search	The eregi() function searches a string for a defined pattern in a case-insensitive fashion. Its prototype follows: **int eregi(string *pattern*, string *string*, [array *regs*])** This function can be useful when checking the validity of strings, such as passwords.
Replacing Text in a Case-Sensitive Fashion	The ereg_replace() function operates much like ereg(), except that its power is extended to finding and replacing a pattern with a replacement string instead of simply locating it. Its prototype follows: string ereg_replace(string *pattern*, string *replacement*, string *string*)
Replacing Text in a Case-Insensitive Fashion	The eregi_replace() function operates exactly like ereg_replace(), except that the search for pattern in string is not case sensitive. Its prototype follows: string eregi_replace(string *pattern*, string *replacement*, string *string*)
Splitting a String into Various Elements Based on a Case-Sensitive Pattern	The split() function divides a string into various elements, with the boundaries of each element based on the occurrence of a defined pattern within the string. Its prototype follows: array split(string *pattern*, string *string* [, int *limit*])
Splitting a String into Various Elements Based on a Case-Insensitive Pattern	The spliti() function operates exactly in the same manner as its sibling, split(), except that its pattern is treated in a case-insensitive fashion. Its prototype follows: array spliti(string *pattern*, string *string* [, int *limit*])
Accommodating Products Supporting Solely Case-Sensitive Regular Expressions	The sql_regcase() function converts each character in a string into a bracketed expression containing two characters. If the character is alphabetical, the bracket will contain both forms; otherwise, the original character will be left unchanged. Its prototype follows: string sql_regcase(string *string*)

Regular Expression Syntax (Perl)
Modifiers

Often you’ll want to tweak the interpretation of a regular expression; for example, you may want to tell the regular expression to execute a case-insensitive search or to ignore comments embedded within its syntax. These tweaks are known as *modifiers*, and they go a long way toward helping you to write short and concise expressions.

Ex:

I: Perform a case-insensitive search.

G: Find all occurrences (perform a global search).

Metacharacters

Perl regular expressions also employ *meta character*s to further filter their searches. A meta character is simply an character or character sequence that symbolizes special meaning. A list of useful meta characters follows:
\A: Matches only at the beginning of the string.
\b: Matches a word boundary.
\B: Matches anything but a word boundary.
\d: Matches a digit character. This is the same as [0-9].
\D: Matches a nondigit character.
\s: Matches a whitespace character.
\S: Matches a nonwhitespace character.
[]: Encloses a character class.
(): Encloses a character grouping or defines a back reference.
$: Matches the end of a line.
^: Matches the beginning of a line.
^: Matches any character except for the newline.
\: Quotes the next metacharacter.
\w: Matches any string containing solely underscore and alphanumeric characters.
This is the same as [a-zA-Z0-9_].
\W: Matches a string, omitting the underscore and alphanumeric characters.

PHP's Regular Expression Functions (Perl Compatible)

PHP offers eight functions for searching and modifying strings using Perl-compatible regular expressions: preg_filter(), preg_grep(), preg_match(), preg_match_all(), preg_quote(), preg_replace(), preg_replace_callback(), and preg_split().

<table>
<tr><td>Searching an Array</td><td>The preg_grep() function searches all elements of an array, returning an array consisting of all elements matching a certain pattern. Its prototype follows:
array preg_grep(string pattern, array input [, int flags])</td></tr>
<tr><td>Searching for a Pattern</td><td>The preg_match() function searches a string for a specific pattern, returning TRUE if it exists and FALSE otherwise. Its prototype follows:
int preg_match(string pattern, string string [, array matches] [, int flags [, int offset]]])</td></tr>
<tr><td>Matching All Occurrences of a Pattern</td><td>The preg_match_all() function matches all occurrences of a pattern in a string, assigning each occurrence to an array in the order you specify via an optional input parameter. Its prototype follows:
int preg_match_all(string pattern, string string, array matches [, int flags] [, int offset]))</td></tr>
<tr><td>Delimiting Special Regular Expression Characters</td><td>The function preg_quote() inserts a backslash delimiter before every character of special significance to
regular expression syntax. These special characters include $ ^ * ( ) + = { } [ ] | \\ : < >. Its prototype follows:
string preg_quote(string str [, string delimiter])</td></tr>
<tr><td>Replacing All Occurrences of a Pattern</td><td>The preg_replace() function replaces all occurrences of pattern with replacement, and returns the modified result. Its prototype follows:
mixed preg_replace(mixed pattern, mixed replacement, mixed str [, int limit [, int count]])</td></tr>
<tr><td>Creating a Custom Replacement Function</td><td>Consider a situation where you want to scan some text for acronyms such as IRS and insert the complete name directly following the acronym. To do so, you need to create a custom function and then use the function</td></tr>
</table>

	preg_replace_callback() to temporarily tie it into the language. Its prototype follows: **mixed preg_replace_callback(mixed *pattern*, callback *callback*, mixed *str*** **[, int *limit* [, int *count*]])**
Splitting a String into Various Elements Based on a Case-Insensitive Pattern	The preg_split() function operates exactly like split(), except that pattern can also be defined in terms of a regular expression. Its prototype follows: **array preg_split(string *pattern*, string *string* [, int *limit* [, int *flags*]])**