UNIT 1

Deciding on a Web Application Platform----Why PHP should be used

- Cost
- Ease of Use
- HTML-embeddedness-PHP can be embedded within HTML. In other words, PHP pages are ordinary HTML pages that escape into PHP mode only when necessary
- Cross-platform compatibility
  PHP and MySQL run native on every popular flavor of Linux/Unix (including Mac OS X) and Microsoft Windows. A huge percentage of the world's Hypertext Transfer Protocol (HTTP) servers run on one of these two classes of operating systems.PHP is compatible with the leading web servers: Apache HTTP Server for Linux/Unix and Windows and Microsoft Internet Information Server. It also works with several lesser-known servers. Specific web server compatibility with MySQL is not required, since PHP will handle all the dirty work for you.
- Stability
  The word *stable* means two different things in this context:
  The server doesn't need to be rebooted or restarted often.
  The software doesn't change radically and incompatibly from release to release.
- Many extensions
  PHP makes it easy to communicate with other programs and protocols. The PHP development team seems committed to providing maximum flexibility to the largest number of users.
- Fast feature development
  Users of proprietary web development technologies can sometimes be frustrated by the glacial speed at which new features are added to the official product standard to support emerging technologies. With PHP, this is not a problem. All it takes is one developer, a C compiler, and a dream to add important new functionality.
- Not proprietary
  The history of the personal computer industry to date has largely been a chronicle of proprietary standards: attempts to establish them, clashes between them, their benefits and drawbacks for the consumer, and how they are eventually replaced with new standards.

Server-Side Scripting

Server-side web scripting is mostly about connecting web sites to backend servers, processing data
and controlling the behavior of higher layers such as HTML and CSS. This enables the following types of two-way communication:
Server to client: Web pages can be assembled from backend-server output.
Client to server: Customer-entered information can be acted upon.

Common examples of client-to-server interaction are online forms with some drop-down lists (usually
the ones that require you to click a button) that the script assembles dynamically on the server.
Server-side scripting products consist of two main parts: the scripting language and the scripting
engine (which may or may not be built into the web server). The engine parses and interprets pages
written in the language.
The following code shows a simple example of server-side scripting — a page assembled on the fly
from a database. We include database calls (which we don't get around to explaining until Part II of
this book) and leave out some of the included files, because we intend this example to show the final
product of PHP rather than serve as a piece of working code.

Server-side scripting languages such as PHP perfectly serve most of the truly useful aspects of the
web, such as the items in this list:
Content sites (both p ■■ roduction and display)
■■Community features (forums, bulletin boards, and so on)
■■E-mail (web mail, mail forwarding, and sending mail from a web application)
■■Customer-support and technical-support systems
■■Advertising networks
■■Web-delivered business applications
■■Directories and membership rolls
■■Surveys, polls, and tests
■■Filling out and submitting forms online
■■Personalization technologies
■■Groupware
■■ nd informational sites
■■Games (for example, chess) with lots of logic but simple/static graphics
■■Any other application that needs to connect a backend server (database, Lightweight
Directory Access Protocol [LDAP], and so on) to a web server

PHP's Syntax

1. PHP is whitespace insensitive--it almost never matters *how many* whitespace characters you have in a row — one whitespace character is the same as many such characters
2. PHP is sometimes case sensitive-In particular, all variables are case sensitive
3. Statements are expressions terminated by semicolon
4. A *statement* in PHP is any *expression* that is followed by a semicolon (;). If expressions correspond
   to phrases, statements correspond to entire sentences, and the semicolon is the full stop at the end.
   Any sequence of valid PHP statements that is enclosed by the PHP tags is a valid PHP program. Here

is a typical statement in PHP, which in this case assigns a string of characters to a variable called
$greeting:
$greeting = “Welcome to PHP!”;

5. Braces make blocks
   Although statements cannot be combined like expressions, you can always put a sequence of statements
   anywhere a statement can go by enclosing them in a set of curly braces

Variables

Important things to know about variables in PHP

1. All variables in PHP are denoted with a leading dollar sign ($).
2. The value of a variable is the value of its most recent assignment.
3. Variables are assigned with the = operator, with the variable on the left-hand side and the expression to be evaluated on the right.
4. Variables can, but do not need, to be declared before assignment.
5. Variables have no intrinsic type other than the type of their current value.
6. Variables used before they are assigned have default values.

All variables in PHP start with a leading $ sign. After the initial $, variable names must be composed of letters (uppercase or lowercase), digits (0–9),and underscore characters (_). Furthermore, the first character after the $ may not be a number.
In PHP, because types are associated with values rather than variables, no such declaration is necessary — the first step in using a variable is to assign it a value.
Variable assignment is simple — just write the variable name, and add a single equal sign (=); then add the expression that you want to assign to that variable:
$pi = 3 + 0.14159;
Comments
A *comment* is the portion of a program that exists only for the human reader. The very first thing that
a program executor does with program code is to strip out the comments, so they cannot have any
effect on what the program does. Comments are invaluable in helping the next person who reads your code figure out what you were thinking when you wrote it, even when that person is yourself a
week from now.
C-style multiline comments
The *multiline* style of commenting is the same as in C: A comment starts with the character pair /*
and terminates with the character pair */. For example:
/* This is
a comment in
PHP */
The most important thing to remember about multiline comments is that they cannot be nested.
Single-line comments: # and //

In addition to the /* ... */ multiple-line comments, PHP supports two different ways of commenting
to the end of a given line: one inherited from C++ and Java and the other from Perl and shell scripts. The shell-script-style comment starts with a pound sign, whereas the C++ style comment starts with two forward slashes. Both of them cause the rest of the current line to be treated as a comment, as in the following:

```
# This is a comment, and
# this is the second line of the comment
// This is a comment too. Each style comments only
// one line so the last word of this sentence will fail
horribly.
```

Types in PHP

PHP's type system is simple, streamlined, and flexible, and it insulates the programmer from low-level
details. PHP makes it easy not to worry too much about typing of variables and values, both because it
does not require variables to be typed and because it handles a lot of type conversions for you.
the type of a variable does not need to be declared in advance. Instead, the programmer can jump right ahead to assignment and let PHP take care of figuring out the type of the expression assigned: PHP does a good job of automatically converting types when necessary. Like most other modern programming languages, PHP will do the right thing when, for example, doing math with mixed
numerical types. The result of the expression

```
$pi = 3 + 0.14159;
```

is a floating-point (double) number, with the integer 3 implicitly converted into floating point before the addition is performed.
Type Summary
PHP has a total of eight types: integers, doubles, Booleans, strings, arrays, objects, NULL, and resources.
■■*Integers* are whole numbers, without a decimal point, like 495.
■■*Doubles* are floating-point numbers, like 3.14159 or 49.0.
■■*Booleans* have only two possible values: TRUE and FALSE.
■■*NULL* is a special type that only has one value: NULL.
■■*Strings* are sequences of characters, like 'PHP 4.0 supports string operations.'
■■*Arrays* are named and indexed collections of other values.
■■*Objects* are instances of programmer-defined classes, which can package up both other kinds of values and functions that are specific to the class.
■■*Resources* are special variables that hold references to resources external to PHP (such as database connections).

The Simple Types

The most of the simple types in PHP are integers, doubles, Booleans, NULL, and strings

Integers

Integers are the simplest type — they correspond to simple whole numbers, both positive and negative.
Integers can be assigned to variables, or they can be used in expressions, like this:

$int_var = 12345;
$another_int = -12345 + 12345;
The PHP constant PHP_INT_MAX will tell you the maximum integer for your implementation
Doubles
Doubles are floating-point numbers, such as:
$first_double = 123.456;
Booleans
Booleans are true-or-false values, which are used in control constructs like the testing portion of an if statement.
Boolean constants
PHP provides a couple of constants especially for use as Booleans: TRUE and FALSE, which can be
used like this:
if (TRUE)
print("This will always print<BR>");
else
print("This will never print<BR>");
Interpreting other types as Booleans
Here are the rules for determine the "truth" of any value not already of the Boolean type:

1. If the value is a number, it is false if the number is zero and true otherwise.
2. If the value is a string, it is false if the string is empty (has zero characters) *or* is the string "0", and is true otherwise.
3. Values of type NULL are always false.
4. If the value is a compound type (an array or an object), it is false if it contains no other values,
5. and it is true otherwise. For an object, *containing a value* means having a member variable

6. that has been assigned a value.

■■Valid resources are true (although some functions that return resources when they are successful
will return FALSE when unsuccessful)
NULL
The
NULL type, however, takes this to the logical extreme: The type NULL has only one possible value,
which is the value NULL. To give a variable the NULL value, simply assign it like this:
$my_var = NULL;
The special constant NULL is capitalized by convention, but actually it is case insensitive; you could
just as well have typed:
$my_var = null;
Strings
Strings are character sequences, as in the following:
$string_1 = "This is a string in double quotes.";
$string_2 = 'This is a somewhat longer, singly quoted string';
$string_39 = "This string has thirty-nine characters.";
$string_0 = ""; // a string with zero characters

Strings can be enclosed in either single or double quotation marks, with different behavior at read
time. Singly quoted strings are treated almost literally, whereas doubly quoted strings replace variables
with their values as well as specially interpreting certain character sequences.

Branching

The two main structures for branching are if and switch

If-else

The syntax for if is:
if (*test*)
statement-1
Or with an optional else branch:
if (*test)*
statement-1
else
statement-2
When an if statement is processed, the test expression is evaluated, and the result is interpreted as a Boolean value. If test is true, statement-1 is executed. If test is not true, and there is an else clause, statement-2 is executed. If test is false, and there is no else clause, execution simply proceeds with the next statement after the if construct.
It's very common to want to do a cascading sequence of tests, as in the following nested if statements:

```
if ($day == 5)
print("Five golden rings<BR>");
else
if ($day == 4)
print("Four calling birds<BR>");
else
if ($day == 3)
print("Three French hens<BR>");
else
if ($day == 2)
print("Two turtledoves<BR>");
else
if ($day == 1)
print("A partridge in a pear tree<BR>");
```

Switch

For a specific kind of multiway branching, the switch construct can be useful. Rather than branch on arbitrary logical expressions, switch takes different paths according to the value of a single expression. The syntax is as follows, with the optional parts enclosed in square brackets ([]):

```
switch(expression)
{
case value-1:
statement-1;
statement-2;
...
[break;]
case value-2:
statement-3;
statement-4;
...
[break;]
...
[default:
default-statement;]
}
```

The expression can be a variable or any other kind of expression, as long as it evaluates to a simple
value (that is, an integer, a double, or a string). The construct executes by evaluating the expression
and then testing the result for equality against each case value. As soon as a matching value is found,
subsequent statements are executed in sequence until the special statement (break;) or until the end of the switch construct A special default tag can be used at the end, which
will match the expression if no other case has matched it so far.

For example, we can rewrite the if-else example as follows:

```
switch($day)
{
case 5:
print("Five golden rings<BR>");
break;
case 4:
print("Four calling birds<BR>");
break;
case 3:
print("Three French hens<BR>");
break;
case 2:
print("Two turtledoves<BR>");
break;
default:
print("A partridge in a pear tree<BR>");
}
```

This will print a single appropriate line for days 2–5; for any day other than those, it will print A partridge in a pear tree. Although switch will accept only a single argument, there’s no reason why that argument can’t be the value of expressions evaluated previously in your code.

Looping

While

The simplest PHP looping construct is while, which has the following syntax:

```
while (condition)
statement
```

The while loop evaluates the *condition* expression as a Boolean — if it is true, it executes
statement
and then starts again by evaluating *condition.* If the condition is false, the while loop terminates. Of
course, just as with if, *statement* may be a single statement or it may be a brace-enclosed block. The
body of a while loop may not execute even once, as in:

```
while (FALSE)
print("This will never print.<BR>");
```

Or it may execute forever, as in this code snippet:

```
while (TRUE)
print("All work and no play makes
Jack a dull boy.<BR>");
```

Or it may execute a predictable number of times, as in:

```
$count = 1;
while ($count <= 10)
{
print("count is $count<BR>");
$count = $count + 1;
}
```

which will print exactly 10 lines.

Do-while

The do-while construct is similar to while, except that the test happens at the end of the loop.
The syntax is:

```
do statement
while (expression);
```

The statement is executed once, and then the expression is evaluated. If the expression is true, the
statement is repeated until the expression becomes false. The only practical difference between
while and do-while is that the latter will always execute its statement at least once. For example:

```
$count = 45;
do
{
print("count is $count<BR>");
$count = $count + 1;
```

}
while ($count <= 10);
prints the single line:
count is 45

For

The most complicated looping construct is for, which has the following syntax:
for (*initial-expression*;*termination-check*;*loop-end-expression*)
statement
In executing a for statement, first the *initial-expression* is evaluated just once, usually to initialize variables. Then *termination-check* is evaluated — if it is false, the for statement concludes, and if it is true, the statement executes. Finally, the *loop-end-expression* is executed and the cycle begins again with *termination-check*. As always, by *statement* we mean a single (semicolon-terminated) statement, a brace-enclosed block, or a conditional construct.
If we rewrote the preceding for loop as a while loop, it would look like this:
initial-expression;
while (*termination-check*)
{
statement
loop-end-expression;
}
Actually, although the typical use of for has exactly one initial-expression, one termination-check,
and one loop-end-expression, it is legal to omit any of them. The termination-check is taken to be always true if omitted, so:
for (;;)
statement

Break and continue

The standard way to get out of a looping structure is for the main test condition to become false. The
special commands break and continue offer an optional side exit from all the looping constructs, including while, do-while, and for:
■■The break command exits the innermost loop construct that contains it.
■■The continue command skips to the end of the current iteration of the innermost loop
that contains it.
For example, the following code:
for ($x = 1; $x < 10; $x++)
{
// if $x is odd, break out
if ($x % 2 != 0)
break;
print("$x ");
}
prints nothing, because 1 is odd, which terminates the for loop immediately. On the other hand,

the code:

```
for ($x = 1; $x < 10; $x++)
{
// if $x is odd, skip this loop
if ($x % 2 != 0)
continue;
print("$x ");
}
```

prints:

```
2 4 6 8
```

because the effect of the continue statement is to skip the printing of any odd numbers.
Using the break command, the programmer can choose to dispense with the main termination test
altogether.

Alternate Control Syntaxes

PHP offers another way to start and end the bodies of the if, switch, for, and while constructs. It amounts to replacing the initial brace of the enclosed block with a colon and the closing brace with a special ending statement for that construct (endif, endswitch, endfor, or endwhile). For example, the if syntax becomes:

if (*expression*):
statement1
statement2
..
endif;

Or:

if (*expression*):
statement1
statement2
..
elseif *(expression2)*:
statement3
..
else:
statement4
..
endif;

Note that the else and elseif bodies also begin with colons. The corresponding while syntax is:

while (*expression*):
statement
endwhile;

Which syntax you use is a matter of taste. The nonstandard syntax in PHP is largely used for historical
reasons and for the comfort of people who are familiar with it from the early versions of PHP.

Terminating Execution

The exit() construct takes either a string or a number as argument, prints out the argument, and then terminates execution of the script. Everything that PHP produces up to the point of invoking exit() is sent to the client browser as usual, and nothing in your script after that point will even be parsed — execution of the script stops immediately. If the argument given to exit is a number rather
than a string, the number will be the return value for the script's execution. Because exit is a construct,
not a function, it's also legal to give no argument and omit the parentheses.
The die() construct is an alias for exit() and so behaves exactly the same way.
functions to make a database connection and that we then use that database connection:

```
$connection = make_database_connection();
if (!$connection)
die("No database connection!");
use_database_connection($connection);
```

UNIT 2

Functions

The basic syntax for using (or *calling*) a function is:

```
function_name(expression_1, expression_2, ..., expression_n)
```

This includes the name of the function followed by a parenthesized and comma-separated list of input expressions (which are called the *arguments* to the function). Functions can be called with zero
or more arguments, depending on their definitions.
When PHP encounters a function call, it first evaluates each argument expression and then uses these values as inputs to the function. After the function executes, the returned value (if any) is the
result of the entire function expression.
All the following are valid calls to built-in PHP functions:

```
sqrt(9); // square root function, evaluates to 3
rand(10, 10 + 10); // random number between 10 and 20
strlen("This has 22 characters"); // returns the number 22
pi(); // returns the approximate value of pi
```

These functions are called with 1, 2, 1, and 0 arguments, respectively.

What is a function?

A *function* is a way of wrapping up a chunk of code and giving that chunk a name, so that you can
use that chunk later in just one line of code. Functions are most useful when you will be using the
code in more than one place, but they can be helpful even in one-use situations, because they can

make your code much more readable.

Function definition syntax
Function definitions have the following form:
function *function-name* (*$argument-1*, *$argument-2*, ..)
{
statement-1;
statement-2;
...
}
That is, function definitions have four parts:
■■The special word function
■■The name that you want to give your function
■■The function's parameter list — dollar-sign variables separated by commas
■■The function body — a brace-enclosed set of statements
Just as with variable names, the name of the function must be made up of letters, numbers, and underscores, and it must not start with a number. Unlike variable names, function names are converted to lowercase before they are stored internally by PHP, so a function is the same regardless
of capitalization.
The short version of what happens when a user-defined function is called is:
1. PHP looks up the function by its name (you will get an error if the function has not yet been defined).
2. PHP substitutes the values of the calling arguments (or the *actual parameters*) into the variables
in the definition's parameter list (or the *formal parameters*).
3. The statements in the body of the function are executed. If any of the executed statements are return statements, the function stops and returns the given value. Otherwise, the function completes after the last statement is executed, without returning a value

```
<html>
<head>
</heaD>
<body>
<form method="get" action="function.php">
enter no1<input type="text" name="no1">
enter no2<input type="text" name="no2">
<select name="number" multiple>
<option id="0">1</option>
<option id="1">2</option>
</select>
<input type="submit" value="add">
</form>
</body>
</html>
```

```
<?php
```

```
function add($no1,$no2)
{
$no3=$no1+$no2;
return $no3;
}
function sub($no1,$no2)
{
$no3=$no1-$no2;
return $no3;
}

$no1=$_GET['no1'];
$no2=$_GET['no2'];
$num=$_GET['number'];
switch($num)
{
case 1:
$sum=add($no1,$no2);
echo "the sum  is $sum";
break;
case 2:
$sum=sub($no1,$no2);
echo "the differenceis $sum";
}
?>
```

Passing Information with PHP

GET Arguments

The GET method passes arguments from one page to the next as part of the Uniform Resource Indicator (you may be more familiar with the term Uniform Resource Locator, or URL) query string. When used for form handling, GET appends the indicated variable name(s) and value(s) to the URL designated in the ACTION attribute with a question mark separator and submits the whole thing to the processing agent (in this case a web server).

When the user makes a selection and clicks the Submit button, the browser agglutinates these elements in this order, with no spaces between the elements:

■ The URL in quotes after the word ACTION (http://localhost/baseball.php)

■ A question mark (?) denoting that the following characters constitute a GET string.

■ A variable NAME, an equal sign, and the matching VALUE (Team=Cubbies)

■ An ampersand (&) and the next NAME-VALUE pair (Submit=Select); further name-value pairs separated by ampersands can be added as many times as the server query-stringlength limit allows.

It then forwards this URL into its own address space as a new request. The PHP script to which the preceding form is submitted (sports.php) will grab the GET variables from the end of the request string, stuff them into the $_GET superglobal array and do something useful with them — in this case, plug one of two values into a text string

POST Arguments

POST is the preferred method of form submission today, particularly in nonidempotent usages (those that will result in permanent changes), such as adding information to a database. The form data set is included in the body of the form when it is forwarded to the processing agent (in this case, PHP). No visible change to the URL will result according to the different data submitted. The POST method has one primary advantage:

■ There is a much larger limit on the amount of data that can be passed (a couple of megabytes rather than a couple of hundred characters).

Passing Information with PHP 6 POST has these disadvantages:

■ The results at a given moment cannot be bookmarked.

■ Browsers exhibit different behavior when the visitor uses their Back and Forward navigation buttons within the browser.

Formatting Form Variables

But because of this automatic variable assignment, you need to always use a good NAME attribute for each INPUT. NAME attributes are not strictly necessary in HTML proper — your form will render fine without them — but the data will be of little use because the HTML form-field NAME attribute will be the variable name in the form handler. In other words, in this form for exampe the text field named email will cause the creation of a PHP variable called $_POST['email'] when the form is submitted. Similarly, the submit button will lead to the creation of a variable called $_POST['submit'] on the next page. The name you use in the HTML form will be the name of your variable in the PHP form handler.

PHP Superglobal Arrays

A change that has been coming for a long time in PHP is the gradual phasing out of automatic global variables in favor of superglobal arrays, which were introduced in PHP4. Understanding superglobal arrays before you understand arrays may present difficulties; if so, we recommend that you read Chapter 8 and come back to this section later. In the good old days before PHP4.1, you could write a piece of code like this and expect it to work:

All GET, POST, COOKIE, ENVIRONMENT, and SERVER variables were made global by the register_ globals directive in php.ini and were directly accessible by their names by default. The PHP team decided to phase out the practice of registering globals, forcing everyone to call these variables as indices in an array (for example, $_POST['secretpassword']). This had already been possible in PHP4, via arrays named $HTTP_GET_VARS, $HTTP_POST_VARS, $HTTP_POST_VARS, and so on, but few developers had used this syntax; frankly, it was a lot of extra keystrokes for a small increase in security. So the PHP team also took this opportunity to rename these arrays with shorter names: $_GET, $_POST, $_COOKIE, $_ENV, and $_SERVER. These superglobal arrays also have one cool feature that may ameliorate some pain: They are automatically global everywhere. This means, for instance, that you no longer have to pass cookie values into a function or declare the $HTTP_COOKIE_VARS array global before you can access those values in a function. This will help those who functionalize to the max and will be a small amelioration for everyone else. As of PHP6, register_globals is officially gone

PHP String Handling

Strings in PHP

Strings are sequences of characters that can be treated as a unit — assigned to variables, given as input to functions, returned from functions, or sent as output to appear on your user's web page. The simplest way to specify a string in PHP code is to enclose it in quotation marks, whether single quotation marks (') or double quotation marks ("), like this:

```
$my_string = 'A literal string';
$another_string = "Another string";
```

The difference between single and double quotation marks lies in how much interpolation PHP does of the characters between the quote signs before creating the string itself. If you enclose a string in single quotation marks, almost no interpolation will be performed; if you enclose it indouble quotation marks, PHP will splice in the values of any variables you include, as well as make substitutions for certain special character sequences that begin with the backslash (\) character. For example, if you evaluate the following code in the middle of a web page:

```
$statement = 'everything I say';
 $question_1 = "Do you have to take $statement so literally?\n";
$question_2 = 'Do you have to take $statement so literally?\n
'; echo $question_1;
 echo $question_2;
```

you should expect to see the browser output:

```
 Do you have to take everything I say so literally?
 Do you have to take $statement so literally?\n
```

Interpolation with curly braces

In most situations, you can simply include a variable in a doubly quoted string, and the variable's value will be spliced into the string when it is interpreted. There are two situations where the string parser might very reasonably get confused and need more guidance from you. The first situation is when your notion of where the variable name should stop is not the same as the parser's, and the other occurs when the expression you want to have interpolated is not a simple variable. In these cases, you can clear things up by enclosing the value you want interpolated in curly braces: {}. For example, PHP has no difficulty with the following code:

```
 $sport = ‘volleyball’;
$plan = “I will play $sport in the summertime”;
```

The parser in this case encounters the $ symbol, and then begins collecting characters for a variable name until it runs into the space after $sport. Spaces cannot be part of a variable name, so it is clear that the variable in question is $sport, and PHP successfully finds a value for that variable (‘volleyball’), and splices the value in. Sometimes, though, it is not convenient to stop a variable name with a space. Take this example:

```
$sport1 = ‘volley’;
 $sport2 = ‘foot’;
$sport3 = ‘basket’;
$plan1 = “I will play $sport1ball in the summertime”; //wrong
$plan2 = “I will play $sport2ball in the fall”; //wrong
 $plan3 = “I will play $sport3ball in the winter”; //wrong
```

You will not get the desired effect here, because PHP interprets $sport1 as part of the variable name $sport1ball, which is probably unbound. Instead, you need something like:
$plan1 = “I will play {$sport1}ball in the summertime”; //right which asks PHP to evaluate only the variable expression within the braces before interpolating. For similar reasons, PHP has difficulty interpolating complex variable expressions, such as multidimensional arrays and object variables, unless curly braces are used. The general rule is that if you have a { immediately followed by a $, PHP will evaluate the variable expression up until the closing } and will interpolate the resulting value into the string. (If you need a literal {$ to appear in your string, you can accomplish it by escaping either character with a backslash (\)).

String operators

PHP offers two string operators: the dot (.) or concatenation operator and the .= concatenating assignment operator. The concatenating assignment operator is discussed in the next section. The concatenation operator, when placed between two string arguments, produces a new string that is the result of putting the two strings together in sequence. For example:

```
 $my_two_cents = “I want to give you a piece of my mind “;
$third_cent = “ And another thing”;
```

 print($my_two_cents . “...” . $third_cent); gives the output: I want to give you a piece of my mind ... And another thing Note that we are not passing multiple string arguments to the print statement — we are handing it one string argument, which was created by concatenating three strings together. The first and third strings are variables, but the middle one is a literal string enclosed in double quotation marks.

Concatenation and assignment

Just as with arithmetic operators, PHP has a shorthand operator (.=) that combines concatenation with assignment. The following statement: $my_string_var .= $new_addition; is exactly equivalent to: $my_string_var = $my_string_var . $new_addition; Note that, unlike commutative addition and multiplication, with this shorthand operator it matters that the new string is added to the right. If you want the new string tacked on to the left, there's no alternative shorter than: $my_string_var = $new_addition . $my_string_var; Note also that unassigned variables are treated as empty strings for the purposes of concatenation, so $my_string_var will end up unchanged if $new_addition has never been given a value.

String Functions

1. **Inspecting strings----------strlen()** function (the name is short for string length).gives length of string
2. **Finding characters and substrings--------------**
   The **strpos()** function finds the numerical position of a particular character in a string, if it exists.The strpos() function can also be used to search for a substring rather than a single character, simply by giving it a multicharacter string rather than a single-character string. You can also supply an extra integer argument specifying the position to begin searching forward from.

   Searching in reverse is also possible, using the **strrpos()** function. (Note the extra r, which you can think of as standing for reverse.) This function takes a string to search and a single-character string to locate, and it returns the last position of occurrence of the second argument in the first argument. (Unlike with strpos(), the string searched for must have only one character.)
3. **Comparison and searching-------------**
   The most basic workhorse string-comparison function is **strcmp().** It takes two strings as arguments and compares them byte by byte until it finds a difference. It returns a negative number if the first string is less than the second and a positive number if the second string is less. It returns 0 if they are identical.

   The **strcasecmp()** function works the same way, except that the equality comparison is case insensitive. The function call strcasecmp("hey!", "HEY!") should return 0.

   The comparison functions just described tell you whether one string is equal to another. To find out if one string is contained within another, use the **strpos()** function (covered earlier) or the strstr() function (or one of its relatives). The strstr() function takes a string to search in and a string to look for (in that order). If it succeeds, it returns the portion of the string that starts with (and includes) the first instance of the string it is looking for. If the string is not found, a false value is returned.

4. **Substring selection**---------------The most basic way to choose a portion of a string is the **substr()** function, which returns a new string that is a subsequence of the old one. As arguments, it takes a string (that the substring will be selected from), an integer (the position at which the desired substring starts), and an optional third integer argument that is the length of the desired substring. If no third argument is given, the substring is assumed to continue until the end.

   Although they are technically substring functions, just like the others in this chapter, the functions **chop(), ltrim(), and trim()** are really used for cleaning up untidy strings. They trim whitespace off the end, the beginning, and the beginning and end, respectively, of their single string argument
5. **String replacement** --------The substring functions we've seen so far are all about choosing a portion of the argument rather than building a genuinely new string. Enter the functions **str_replace() and substr_replace()**.
   The **str_replace() function** enables you to replace all instances of a particular substring with an alternate string. It takes three arguments: the string to be searched for, the string to replace it with when it is found, and the string to perform the replacement on.As you've seen, str_replace() picks out portions to replace by matching to a target string; by contrast, **substr_replace()** chooses a portion to replace by its absolute position. The function takes up to four arguments: the string to perform the replacement on, the string to replace it with, the starting position for the replacement, and (optionally) the length of the section to be replaced. The substr_replace() function also takes negative arguments for starting position and length,
   The **strrev()** function simply returns a new string with the characters of its input in reverse order. **The str_repeat()** function takes a string argument and an integer argument and returns a string that is the appropriate number of copies of the string argument tacked together.
6. **Case functions** -----------These functions change lowercase to uppercase and vice versa. The first two (de)capitalize entire strings, whereas the second two operate only on first letters of words. strtolower() The **strtolower()** function returns an all-lowercase string. It doesn't matter if the original is all uppercase or mixed
   strtoupper() The **strtoupper()** function returns an all-uppercase string, regardless of whether the original was all lowercase or mixed **ucfirst()** The ucfirst() function capitalizes only the first letter of a string **ucwords()** The ucwords() function capitalizes the first letter of each word in a string:
7. **Escaping functions**
   The **addslashes()** function escapes quotation marks, double quotation marks, backslashes, and NULLs with backslashes, because these are the characters that typically need to be escaped for database queries.

```
<?php
$escapedstring = addslashes(“He said, ‘I’m a dog.’“);
$query = “INSERT INTO test (quote) values (‘$escapedstring’)“;
$result = mysql_query($query) or die(mysql_error());
?>
```

This will prevent the SQL statement from thinking it’s finished right before the letter I. When you
pull the data back out, you’ll need to use stripslashes() to get rid of the slashes.

```
<?php
$query = “SELECT quote FROM test WHERE ID=1”;
$result = mysql_query($query) or die(mysql_error());
$new_row = mysql_fetch_array($result);
$quote = stripslashes($new_row[0]);
echo $quote;
```

The **quotemeta()** function escapes a wider variety of characters, all of which usually have a special
meaning in the Unix command line: ‘.’, ‘\‘ ‘+’, ‘*‘, ‘?’, ‘[‘, ‘^’, ‘]‘, ‘(‘, ‘$’, and ‘)‘.
For example, the code:

```
$literal_string =
 ‘These characters ($, *) are very special to me\n<BR>’;
$qm_string = quotemeta($literal_string);
echo $qm_string;
```

will print:

```
These characters \(\$, \*\) are very special to me\\n
```

8. **Printing and output**

PHP also offers printf() and sprintf(), which
are modeled on C functions of the same name. The two functions take identical arguments: a special format string (described later in this section) and then any number of other arguments, which will be spliced into the right places in the format string to make the result.

The only difference between printf() and sprintf() is that **printf()** sends the resulting string directly to output, whereas **sprintf()** returns the result string as its value.

The complicated bit about these functions is the format string. Every character that you put in the string will show up literally in the result, except the % character and characters that immediately follow it. The % character signals the beginning of a conversion specification, which indicates how to print one of the arguments that follow the format string. After the %, there are six elements that make up the conversion specification, some of which are optional: padding, alignment, minimum width, precision, and type.

■ An optional sign character used for numbers to indicate whether the number will be negative (-).

■ The single (optional) padding character is either a 0 or a space ( ). This character is used to fill any space that would otherwise be unused but that you have insisted (with the minimum width argument) be filled with something. If this padding character is not given, the default is to pad with spaces.

■ The optional alignment character (-) indicates whether the printed value should be left- or right-justified. If present, the value will be left-justified; if absent, it will be right-justified.

■ An optional minimum width number that indicates how many spaces this value should take up, at a minimum. (If more spaces are needed to print the value, it will overflow beyond itsbounds.)

■ An optional precision specifier is written as a dot (.) followed by a number. It indicates how many decimal points of precision a double should print with. (This has no effect on printing things other than doubles.)

■ A single character indicating how the type of the value should be interpreted. The f character indicates printing as a double, the s character indicates printing as a string, and then the rest of the possible characters (b, c, d, o, x, X) mean that the value should be interpreted as an integer and printed in various formats. Those formats are b for binary, c for printing the character with the corresponding ASCII values, o for octal, x for hexadecimal (with lowercase letters) and X for hexadecimal with uppercase letters.

Here's an example of printing the same double in several different ways:

```
<pre>
<?php
$value = 3.14159;
printf("%f,%10f,%-010f,%2.2f\n",
 $value, $value, $value, $value);
?>
</pre>
```

gives us:

```
3.141590, 3.141590,3.141590000000000, 3.14
```

Arrays

An array is a collection of variables indexed and bundled into a single, easily referenced supervariable that offers an easy way to pass multiple values between lines of code, functions, and even pages.

Many built-in PHP environment variables are in the form of arrays (for example, $_SESSION, which contains all the variable names and values being propagated from page to page via PHP's session mechanism). If you want access to them, you need to understand, at a minimum, how to reference Arrays.

What Are PHP Arrays?

PHP arrays are associative arrays with a little extra machinery thrown in. The associative part means that arrays store element values in association with key values rather than in a strict linear index order.
If you store an element in an array, in association with a key, all you need to retrieve it later from that array is the key value. For example, storage is as simple as this:
$state_location[‘San Mateo’] = ‘California’;
which stores the element ‘California’ in the array variable $state_location, in association with the lookup key ‘San Mateo’. After this has been stored, you can look up the stored value by using the key, like so:
$state = $state_location[‘San Mateo’]; // equals ‘California’
If all you want arrays for is to store key/value pairs, the preceding information is all you need to know. Similarly, if you want to associate a numerical ordering with a bunch of values, all you have to do is use integers as your key values, as in:
$my_array[1] = “The first thing”;
$my_array[2] = “The second thing”;
arrays can be multidimensional. They can store values in association with a sequence of key values rather than a single key.

Creating Arrays

There are three main ways to create an array in a PHP script: by assigning a value into one (and thereby implicitly creating it), by using the array() construct, and by calling a function that happens to return an array as its value.
Direct assignment The simplest way to create an array is to act as though a variable is already an array and assign a value into it, like this:
$my_array[1] = “The first thing in my array that I just made”;
If $my_array was an unbound variable (or bound to a nonarray variable) before this statement, it will now be a variable bound to an array with one element. If instead $my_array was already an array, the string will be stored in association with the integer key 1. If no value was associated with that number before, a new array slot will be created to hold it; if a value was associated with 1, the previous value will be overwritten

The array() construct

The other way to create an array is via the array() construct, which creates a new array from the specification of its elements and associated keys. In its simplest version, array() is called with no arguments, which creates a new empty array. In its next simplest version, array() takes a comma separated list of elements to be stored, without any specification of keys. The result is that the elements are stored in the array in the order specified and are assigned integer keys beginning with zero. For example, the statement:

$fruit_basket = array('apple', 'orange', 'banana', 'pear');
causes the variable $fruit_basket to be assigned to an array with four string elements ('apple','banana', 'orange', 'pear'), with the indices 0, 1, 2, and 3, respectively.
Specifying indices using array()
The simple example of array() in the preceding section assigns indices to our elements, but those indices will be the integers, counting upward from zero — we're not getting a lot of choice in the matter. As it turns out, array() offers us a special syntax for specifying what the indices should be. Instead of element values separated by commas, you supply key/value pairs separated by commas, where the key and value are separated by the special symbol =>.Consider the following statement:
$fruit_basket = array(0 => 'apple', 1 => 'orange',2 => 'banana', 3 => 'pear');
Evaluating it will have exactly the same effect as our earlier version — each string will be stored in the array in succession, with the indices 0, 1, 2, 3 in order. Instead, however, we can use exactly the same syntax to store these elements with different indices:
$fruit_basket = array('red' => 'apple', 'orange' => 'orange','yellow' => 'banana', 'green' => 'pear');
This gives us the same four elements, added to our new array in the same order, but indexed by color names rather than numbers. To recover the name of the yellow fruit, for example, we just evaluate the expression:
$fruit_basket['yellow'] // will be equal to 'banana'
Finally, as we said earlier, you can create an empty array by calling the array function with no arguments. For example:
$my_empty_array = array();
creates an array with no elements. This can be handy for passing to a function that expects an array as argument.

Functions returning arrays

The final way to create an array in a script is to call a function that returns an array. This may be a user defined function, or it may be a built-in function that makes an array via methods internal to PHP. Many database-interaction functions, for example, return their results in arrays that the functions create on the fly. Other functions exist simply to create arrays that are handy to have as grist for later array-manipulating functions. One such is range(), which takes two integers as arguments and returns an array filled with all the integers (inclusive) between the arguments. In other words:
$my_array = range(1,5);
is equivalent to:
$my_array = array(1, 2, 3, 4, 5);

Retrieving Values

Retrieving by index

The most direct way to retrieve a value is to use its index. If we have stored a value in $my_array at index 5, $my_array[5] should evaluate to the stored value. If $my_array has never been assigned, or if nothing has been stored in it with an index of 5, $my_array[5] will behave like an unbound variable.

The list() construct

There are a number of other ways to recover values from arrays without using keys, most of which exploit the fact that arrays are silently recording the order in which elements are stored. We cover this in more detail in this chapter's "Iteration" section, but one such example is list(), which is used to assign several array elements to variables in succession. Suppose that the following two statements are executed:

```
$fruit_basket = array('apple', 'orange', 'banana');
list($red_fruit, $orange_fruit) = $fruit_basket;
```

This will assign the string 'apple' to the variable $red_fruit and the string 'orange' to the variable $orange_fruit (with no assignment of 'banana', because we didn't supply enough variables). The variables in list() will be assigned to elements of the array in the order they were originally stored in the array. Notice the unusual behavior here — the list() construct is on the left-hand side of the assignment operator (=), where we normally find only variables.

Multidimensional Arrays

multidimensional arrays are simply arrays that have other arrays stored in them, it's easier to see how the array() creation construct generalizes. In fact, even this seemingly complicated assignment is not that complicated:

```
$cornucopia = array('fruit' => array('red' => 'apple','orange' => 'orange','yellow' =>
'banana','green' => 'pear'),'flower' =>array('red' => 'rose','yellow' =>
'sunflower','purple' => 'iris'));
```

It is simply an array with two values stored in association with keys. Each of these values is an arrayitself. After we have made the array, we can reference it like this:

```
$kind_wanted = 'flower';
$color_wanted = 'purple';
print("The $color_wanted $kind_wanted is " .
$cornucopia[$kind_wanted][$color_wanted]);
```

Inspecting Arrays

1. **is_array()---**Takes a single argument of any type and returns a true value if the argument is an array, and false otherwise.

2. **count()-----**Takes an array as argument and returns the number of nonempty elements in the array. (This will be 1 for strings and numbers.)
3. **sizeof()---**Takes an array as argument and returns the number of nonempty elements in the array. (This will be 1 for strings and numbers.)
4. **in_array()-----**Takes two arguments: an element (that might be a value in an array), and an array (that might contain the element). Returns true if the element is contained as a value in the array, false otherwise. (Note that this does not test for the presence of keys in the array.)
5. **isset($array[$key])-----**Takes an array[key] form and returns true if the key portion is a valid key for the array.

Deleting from Arrays

Deleting an element from an array is simple, exactly analogous to getting rid of an assigned variable.
Just call unset(), as in the following:

```
$my_array[0] = ‘wanted’;
$my_array[1] = ‘unwanted’;
$my_array[2] = ‘wanted again’;
unset($my_array[1]);
```

Assuming that $my_array was unbound when we started, at the end it has two values (‘wanted’,‘wanted again’), in association with two keys (0 and 2, respectively). It is as though we had skipped the original ‘unwanted’ assignment (except that the keys are numbered differently). Note that this is not the same as setting the contents to an empty value. If, instead of calling unset(), we had the following statement:

```
$my_array[1] = ‘’;
```

at the end we would have three stored values (‘wanted’, ‘’, ‘wanted again’) in association with three keys (0, 1, and 2, respectively).

Iteration

FOR each

```
foreach ($array_variable as $key_var => $value_var) {
// .. do something with $key_var and/or $value_var
}
```

Current and Next function

The current() function returns the stored value that the current pointer points to. When an array is newly created with elements, the element pointed to will always be the first element. The next() function first advances that pointer and then returns the current value pointed to. If the next() function is called when the current pointer is already pointing to the last stored value and, therefore, runs off the end of the array, the function returns a false value.

Reset Function

The reset() function gives us a way to “rewind” that pointer to the beginning — it sets the pointer to the first key/value pair and then returns the stored value.

Reverse order with end() and prev()

the functions prev() moves the pointer back by one, and end(), which jumps the pointer to the last entry in the list.

Extracting keys with key()

The keys are also retrievable from the internal linked list of an array by using the key() function — this acts just like current() except that it returns the key of a key/value pair, rather than the value.

Empty values and the each() function

We have written several functions that print the contents of arrays by iterating through them and,as we have pointed out, all but the foreach version have the same weakness. Each one of them tests for completion by seeing whether next() returns a false value. This will reliably happen when the array runs out of values, but it will also happen if and when we encounter a false value that we have actually stored. False values include the empty string (“”), the number 0, and the Boolean value FALSE, any or all of which we might reasonably store as a data value for some task or other.
To the rescue comes each(), which is somewhat similar to next() but has the virtue of returning false only after it has run out of array to traverse. Oddly enough, if it has not run out, each() returns an array itself, which holds both keys and values for the key/value pair it is pointing at. This characteristic makes each() confusing to talk about because you need to keep two arrays straight: the array that you are traversing and the array that each() returns every time that it is called. The array that each() returns has the following four key/value pairs:
■■Key: 0; Value: current-key
■■Key: 1; Value: current-value
■■Key: ‘key’; Value: current-key
■■Key: ‘value’; Value: current-value
The current-key and current-value are the key and value from the array being traversed. In other words, the returned array packages up the current key/value pair from the traversed array and offers both numerical and string indices to specify whether you are interested in the key or the value.

Walking with array_walk()

Our last iteration function lets you pass an arbitrary function of your own design over an array,doing whatever your function pleases with each key/value pair. The array_walk() function takes two arguments: an array to be traversed and the name of a function to apply to each key/value pair.The function that is passed in to array_walk() should take two (or three) arguments. The first argument will be the value of the array cell that is visited, and the second argument will be the key of that cell.

```
function print_value_length($array_value, $array_key_ignored)
{
$the_length = strlen($array_value);
print(“The length of $array_value is $the_length<BR>”);
}
```

function over our standard sample array using array_walk():

```
array_walk($major_city_info, ‘print_value_length’);
```

USE THIS PROGRAM SNIPPET AS EXAMPLE FOR ALL FUNCTIONS

```
<?php
//single dimensional array creation with index

$myarr=array(1,2,3,4,5,6);
$fruit=array("apple","mango","banana");

//single dimensional array creation with key value pair

$fruit_basket=array('red'=>'apple','green'=>'grapes','yellow'=>'mango');

//single dimensional array access with index

for($i=0;$i<=5;$i++)
echo "$myarr[$i]<br>";
for($i=0;$i<3;$i++)
echo "$fruit[$i]<br>";

//single dimensional array access with key value pair

echo $fruit_basket['red'];
echo $fruit_basket['green'];
echo $fruit_basket['yellow'];
```

```
$my_aray=range(10,15);
for($i=0;$i<=5;$i++)
echo "$my_aray[$i]<br>";

//Multidimensional array creation and access

$mult=array('fruit'=>array('red'=>'apple','green'=>'grapes','yellow'=>'mango'),'flower'=>a
rray('red'=>'rose','green'=>'lily','yellow'=>'sunflower'));
echo $mult['fruit']['green'];

//function to check if argument is array or not

$myarr1=10;
if(is_array($myarr1))
echo "<br>array present";
else
echo "<br>array absent";

//function to print total number of elements in array


echo "<br>total number of elements in array are ".count($myarr);


//function to print size of array

echo "<br>size of array is ".sizeof($fruit);

//function to check if element is present in array

if(in_array('mango',$fruit))
echo "<br>element present";
else
echo "<br>element absent";

//function to check if key is present in array

if(isset($fruit_basket['yellow']))
echo "<br>key present";
else
echo "<br>key absent";
```

```
//deleting from array

unset($myarr[2]);
for($i=0;$i<=5;$i++)
echo "$myarr[$i]<br>";

//iteration method foreach

foreach($fruit_basket as $col => $frt)
{
echo "<br>".$col;
echo "     ".$frt;
}

//current and next method

while(current($myarr))
{
echo "<br>".current($myarr);
next($myarr);
}

//Reseting the position of array to start

echo "<br>".current($myarr);
reset($myarr);
echo "<br>".current($myarr);


//using the end and prev methods

end($myarr);
do
{
echo "<br>".current($myarr);
}while(prev($myarr));

/using the key method to print keys of array


reset($fruit_basket);
while(key($fruit_basket))
```

```
{
echo "<br>".key($fruit_basket);
next($fruit_basket);
}

//using each method for printing key value pair

reset($fruit_basket);
while($a=each($fruit_basket))
{
echo "<br>".$a['value']."\t".$a['key'];
}

//using array walk function

function find($array_value,$array_key_ignored)
{
echo "<br>the length of $array_value is   ".strlen($array_value);
echo "<br>the length of $array_key_ignored is   ".strlen($array_key_ignored);
}

array_walk($fruit_basket,'find');
```

Number Handing

Numerical Types

PHP has only two numerical types: integer (also known as long), and double (aka float).PHP does automatic conversion of numerical types,

In situations where you want a value to be interpreted as a particular numerical type, you can force a typecast by prepending the type in parentheses, such as:

(double) $my_var

(integer) $my_var

Or you can use the functions intval() and doubleval(), which convert their arguments to integers and doubles, respectively.

Mathematical Operators

+,-,*,%,/

Arithmetic operators and types

With the first three arithmetic operators (+, -, *), you should expect type contagion from doubles to integers; that is, if both arguments are integers, the result will be an integer, but if either argument is a double, then the result will be a double. With the division operator, there is the same sort of contagion,and in addition the result will be a double if the division is not even.The modulus operator in PHP (%) expects integer arguments — if it is given doubles, they will simply be converted to integers (by truncation) first. The result is always an integer

Incrementing operators

The increment operator (++) adds one to the variable it is attached to, and the decrement operator (--) subtracts one from the variable. Each one comes in two flavors, postincrement (which is placed immediately after the affected variable), and preincrement (which comes immediately before). Both flavors have the same side effect of changing the variable's value, but they have different values as expressions. The postincrement operator acts as if it changs the variable's value after the expression's value is returned, whereas the preincrement operator acts as though it makes the change first and then returns the variable's new value

Assignment operators

all five arithmetic operators have corresponding assignment operators (+=, -=, *=, /=, and %=) that assign to a variable the result of an arithmetic operation on that variable in one fell swoop

Comparison operators

PHP includes the standard arithmetic comparison operators, which take simple values (numbers or strings) as arguments and evaluate to either TRUE or FALSE:
■The < (less than) operator is true if its left-hand argument is strictly less than its right-hand argument but false otherwise.
■■The > (greater than) operator is true if its left-hand argument is strictly greater than its right-hand argument but false otherwise.
■■The <= (less than or equal) operator is true if its left-hand argument is less than or equal to its right-hand argument but false otherwise.
■■The >= (greater than or equal) operator is true if its left-hand argument is greater than or equal to its right-hand argument but false otherwise.
■■The == (equal to) operator is true if its arguments are exactly equal but false otherwise.

■■The != (not equal) operator is false if its arguments are exactly equal and true otherwise. This operator is the same as <>.
■■The === operator (identical to) is true if its two arguments are exactly equal and of the same type.
■■The !== operator (not identical to) is true if the two arguments are not equal or not of the same type.

Precedence and parentheses

Operator precedence rules govern the relative stickiness of operators, deciding which operators in an expression get first claim on the arguments that surround them.
■■ Arithmetic operators have higher precedence (that is, bind more tightly) than comparison
operators.
■■Comparison operators have higher precedence than assignment operators.
■■The *, /, and % arithmetic operators have the same precedence.
■■The + and – arithmetic operators have the same precedence.
■■The *, /, and % operators have higher precedence than + and –.
■■When arithmetic operators are of the same precedence, associativity is from left to right
(that is, a number will associate with an operator to its left in preference to the operator on
its right).

Simple Mathematical Functions

floor() -----Takes a single argument (typically a double) and returns the largest integer that is lessthan or equal to that argument.
ceil() -----Short for ceiling — takes a single argument (typically a double) and returns the smallest integer that is greater than or equal to that argument.
round() ------------Takes a single argument (typically a double) and returns the nearest integer. If thefractional part is exactly 0.5, it returns the nearest even number.
abs()----------Short for absolute value — if the single numerical argument is negative, the corresponding positive number is returned; if the argument is positive, the argument itself is returned.
min()------------ Takes any number of numerical arguments (but at least one) and returns the smallest of the arguments.
max() --------Takes any number of numerical arguments (but at least one) and returns the largest of the arguments.

Randomness

There are two random number generators (invoked with rand() and mt_rand(), respectively),each with the same three associated functions: a seeding function, the random number functionitself, and a function that retrieves the largest integer that might be returned by the generator.

Function and Behavior

srand() ------Takes a single positive integer argument and seeds the random number generator with it.
rand()------- If called with no arguments, returns a "random" number between 0 and RAND_MAX (which can be retrieved with the function getrandmax()). The function can also be called with two integer arguments to restrict the range of the number returned — the first argument is the minimum and the second is the maximum (inclusive).
getrandmax() Returns the largest number that may be returned by rand(). This number is limited to 32768 on Windows platforms.
mt_srand()------- Like srand(), except that it seeds the "better" random number generator.
mt_rand() ----------Like rand(), except that it uses the "better" random number generator.
mt_getrandmax()--------Returns the largest number that may be returned by mt_rand().

Seeding the generator

The typical way to seed either of the PHP random number generators (using mt_srand() or srand()) looks like this:
mt_srand((double)microtime()*1000000);
This sets the seed of the generator to be the number of microseconds that have elapsed since the last whole second.

PHP Gotchas

Installation-Related Problems

Symptom: Text of file displayed in browser window

If you are seeing the text of your PHP script instead of the resulting HTML, the PHP engine is clearly not being invoked. Check that you are accessing the site through the web server and not via the filesystem.
Use this:
http://localhost/mysite/mypage.php
rather than this:
file://home/httpd/html/mysite/mypage.php

Symptom: PHP blocks showing up as text under HTTP or browser prompts you to save file
The second most common reason is that your php.ini file is in the wrong place or has a bad configuration directive.
Symptom: Server or host not found/Page cannot be displayed
If your browser can't find your server, you may have a DNS (Domain Name Service) or Web-server configuration issue.
If you can get to the site via IP address rather than domain name, your problem is probably DNS-related.
If you cannot get to the site via IP address for a new installation, it's likely you haven't successfully bound the IP address to your network interface or configured the web server to handle requests for a particular domain

Rendering Problems

This section covers problems where PHP does not report an error per se, but what you see is not what you thought you would get.
Symptom: Totally blank page
A blank page could be caused by any number of issues. Usually, it's caused by a fatal error in the PHP code from which the PHP interpreter cannot recover. Begin by debugging at the top of the PHP file that you're trying to visit by placing a die() after the opening <?php tag:
Of course, another possible answer in this case is that the PHP module is not working at all. Test by browsing a different page in the same directory that you've previously verified is being correctly handled by PHP.
Symptom: PHP code showing up in Web browser
If you are seeing literal PHP code in your browser, rather than a rendering of the HTML it should be producing, you may have omitted a PHP start tag somewhere.
Symptom: Page cannot be found
If you get this message when you have been loading other PHP files without incident, it's quite likely you are just misspelling the filename or path. Alternatively, you may be confused about where the web server document root is located.
Symptom: Failed opening [file] for inclusion
When including files from PHP files, we sometimes see errors like this (on a Unix platform, the file paths would be different):
Warning Failed opening 'C:\InetPub\wwwroot\asdf.php' for inclusion (include_path='') in [no active file] on line 0 It turns out that this is the included-file version of Page cannot be found — that is, PHP hasn't even gotten to loading the first line of the active file. There is no active file because no file by that name could be found.

Parse Errors

The most common category of error arises from mistyped or syntactically incorrect PHP code, which confuses the PHP parsing engine.

Symptom: Parse error message

The most common causes of parse errors, detailed in the subsections that follow, are all quite minor and easy to fix, especially with PHP lighting the way for you. However, every parse error returns the identical message (except for filenames and line numbers) regardless of cause. Any HTML that may be in the file, even if it appears before the error-causing PHP fragment, will not be displayed or appear in the source code.

The missing semicolon

If each PHP instruction is not duly finished off with a semicolon, a parse error will result. In this sample fragment, the first line lacks a semicolon, and therefore, the variable assignment is never completed.

What we have here is

```
<?php
$Problem = “a silly misunderstanding”
echo $Problem; ?>.
```

No dollar signs

Another very common problem is that a dollar sign prepending a variable name is missing. If the dollar sign is missing during the initial variable assignment, like this:

What we have here is

```
<?php
Problem = “a big ball of earwax”;
echo $Problem; ?>.
```

a parse error message will result. However, if instead the dollar sign is missing from a later output of the variable, like this:

What we have here is

```
<?php
$Problem = “a big ball of earwax”;
print(“Problem”); ?>.
```

PHP will not indicate a parse error

Mode issues

Another family of glitches arises from faulty transitions in and out of PHP mode.

A parse error will result if you fail to close off a PHP block properly, as in:

What we have here is

```
<?php
$Problem = “Bad Code!”;
echo $Problem; .
```

This particular mode issue is very common with short PHP blocks. Conversely, if you fail to begin the PHP block properly, the rest of the intended block will simply appear as HTML

Unescaped quotation marks

Another type of parse error is characteristic of maximal PHP: the unescaped quotation mark.

```
<?php
print(“She said, /“What we have here is “);
$Problem = “a difference of opinion\“”;
print(“$Problem”); ?>.
```

In this case, the double-quote just before the word What is incorrectly, and therefore ineffectively,escaped by a forward slash rather than a backslash. If you simply forgot the backslash, the effect would be the same.

Unterminated strings

Failing to close off a quoted string can cause parse errors that refer to line numbers far away from the source of the problem. For example, a code file like this:

```
print(“I am a guilty print statement!); // line 5
// 47 lines of PHP code omitted ...
print(“I am an innocent print statement!”); // line 53
```

might well produce a parse error that complains about line 53. This is because PHP is happy to include any text you might want in a quoted string, including many lines of your own code. This inclusion finishes happily with the first double-quote in line 53, and then the parser finds the symbol I, which it can’t figure out how to interpret as PHP code.
If the quotation mark symbol that begins the unterminated string happens to be the last one in the file, the line number in the complaint will be the last line in the file — again, probably far away from the scene of the crime.

Missing Includes

In addition to loading top-level source files, PHP needs to be able to load any files you bring in via include() or require().

Symptom: Include warning

The problem is that you call somewhere in the script for a file to be included, but PHP can’t find it.Check to see that the path is correct. You might also have a case sensitivity or other typographic issue. Note the important difference between include() and require(). If a file is included and PHP can’t locate the file, execution of the script will continue with a PHP warning. If a file is required and PHP can’t locate that file, execution will stop with an error

Unbound Variables

PHP is different from many programming languages in that variables do not have to be declared before being assigned, and (under its default settings) PHP will not complain if they are used before being assigned (or bound) either. As a result, forgetting to assign a variable will not result in direct errors —

Symptom: Variable not showing up in print string

If you embed a variable in a double-quoted string (“like $this”) and then print the string using print or echo, the variable’s value should show up in the string. If it seems to not be there at all in the output (“like “), the variable has probably never been assigned.
Symptom: Numerical variable unexpectedly zero

Although it’s possible to have a math error or misunderstanding result in this symptom, it’s much more likely that you believe that the variable has been assigned when it actually hasn’t been.
Causes of unbound variables
PHP automatically converts the types of variables depending on the context in which they are used, and this is also true of unbound variables. In general, unbound variables are interpreted as 0 in a numerical context, “” in a string context, FALSE in a Boolean context, and as an empty array in an array context
Case problems
Variables in PHP are case sensitive, so the same name with different capitalization results in a different variable. Even after a value is assigned to the variable $Mississippi, the variable $mississippi will still be unbound. (Capitalization aside, variables that are this difficult to spell are probably to be avoided for the same reason.)
Scoping problems
As long as no function definitions are involved, PHP variable scoping is simple: Assign a variable,and its value will be there for you from that point on in that script’s execution

Function Problems

Many problems having to do with function calls result in fatal errors, which means that PHP gives up on processing the rest of the script.
Symptom: Call to undefined function my_function()
PHP is trying to call the function my_function(), which has not been defined. This could becuse you misspelled the name of a function (built-in or user-defined) or because you have simply omitted the function definition.
Symptom: Call to undefined function ()
In this case, PHP is trying to call a function and doesn’t even know the function’s name. This is invariably because you have code of the form $my_function(), where the name of the function is itself a variable
Symptom: Call to undefined function array()
This problem has a cause that is similar to the cause of the previous problem, although it still baffled us completely the first time we ran into it. It can arise when you have code like the following:
$my_amendments = array();

```
$my_amendments(5) = “the fifth”;
```

Unless you look closely, this looks like an innocent pair of statements to create an array and then store something in that array, with the number 5 as a key. And yet PHP is telling us that array() is an unbound function, even though we know that it is a very standard built-in function. What’s going on? The fault is actually with Line 2 above, rather than with Line 1. If we want to access an element of $my_amendments, the correct syntax is $my_amendments[5], with square brackets. Instead, we used parentheses, which the parser interprets as an attempted function call. It takes what is immediately
before the left parenthesis to be a function. Instead, what comes before the parenthesis is an array, which is not a function; PHP gives up on us, with this obscure complaint.

Symptom: Cannot redeclare my_function()

This is a simple one — somewhere in your code you have two definitions of my_function(), which PHP will not stand for. Make sure that you are not using include to pull in the same file of function definitions more than once.

Symptom: Wrong parameter count

The function named in the error message is being called with either fewer or more arguments than it is supposed to handle. In the case of more parameters you’re okay, but if you use fewer parameters than is expected you will get an error.

Symptom: Division-by-zero warning

Somewhere in your code, you have a division operator where the denominator is zero. The most common cause of this is an unbound variable, as in:

```
$numerator = 5;
$ratio = $numerator / $denominator;
```

where $denominator is unbound

Symptom: Unexpected arithmetic result

Sometimes things just don’t add up (or multiply up, or subtract up). If you are having this experience,check any complex arithmetic expressions for unbound variables (which would act as zeros)and for precedence confusions. If you have any doubt about the precedence of operators, add (possiblyredundant) parentheses to make sure the grouping is as you intend.

Symptom: NaN (or NAN)

If you ever see this dreaded acronym, it means that some mathematical function you used has gone out of range or given up on its inputs. The value NAN stands for “Not a Number,” and it has some special properties. Here’s what happens if we try to take the arccosine of 45, even though arccosine is defined only when applied to numbers between –1.0 and 1.0:

Timeouts

Of course any download can occasionally time out before a complete page can be delivered.

UNIT 3

MYSQL

What Is a Database?

A database is a collection of data.Databases implemented through a computer are created within software. That software, commonly known as a database application, controls how the actual data is stored and retrieved. Some database applications include Microsoft Access and OpenOffice.org's Base. Sometimes, databases are stored in a central location and managed by a database server. A database server is a database application built with multiple users in mind. Most of the time when programming PHP you'll be accessing a database server. Some database servers include PostgreSQL, MySQL, Microsoft's SQL Server, and the Oracle suite of databases. You may also see database servers called RDBMS, which is an acronym for relational database management system.

Why a Database?

Maintainability and scalability

Having PHP assemble your pages on the fly from a template and a database is an addictive experience.Once you enjoy it, you'll never go back to managing a static HTML site of any size. For th effort of programming one page, you can produce an infinite number of uniform pages. Change one, and you've changed them all.

Portability

Because a database is an application rather than a part of the operating system, you can easily transfer its structure and contents from one machine to another or (in certain cases) even from one platform to another. This is especially valuable for contractors, who may develop a project without being able to control the environment in which it will eventually be deployed — they can deliver a package of PHP plus a MySQL database schema dump.

Avoiding awkward programming

Searching

Although it's possible to search multiple text files for strings (especially on Unix platforms), it's not something most web developers will want to do often. After you search a few hundred files, the task becomes slow and hard to manage. Databases exist to make searching easy. With a single command, you can find anything from one ID number to a large text block to a JPEG-format image.

Connecting to MySQL

The basic command to initiate a MySQL connection if you're using variables is
mysql_connect($hostname, $user, $password) or
mysql_connect('localhost', 'root', 'sesame'); if you're using literal strings.
The password is optional, depending on whether this particular database user requires one (it's a good idea). If not, just leave that variable off. You can also specify a port and socket for the server ($hostname:port:socket), but unless you've specifically chosen a nonstandard port and socket, there's little to gain by doing so. The corresponding mysqli function is mysqli_connect, which adds a fourth parameter allowing you to select a database in the same function you use to connect. The function mysqli_select_db exists, but you'll need it only if you want to use multiple databases on the same connection. You do not need to establish a new connection each time you want to query the database in the same script. You will need to run this function again, however, for each script that interacts with the database in some fashion.

Next, you'll want to choose a database to work on:
mysql_select_db($database); if you're using variables, or
mysql_select_db('phpbook'); if you're using a literal string.

Making MySQL Queries
A database query from PHP is basically a MySQL command wrapped up in a tiny PHP function called mysql_query().
The MySQL commands to CREATE or DROP a table can also be used with this PHP function if you do not wish to make your databases using the MySQL client.You could write a query in the simplest possible way, as follows:

```
mysql_query("SELECT Surname FROM personal_info WHERE ID < 10");
```

PHP would dutifully try to execute it. However, there are very good reasons to split up this and similar commands into two lines with extra variables, like this:

```
$query = "SELECT Surname FROM personal_info WHERE ID < 10";
$result = mysql_query($query);
```

The function mysql_query takes as arguments the query string (which should not have a semicolon within the double quotation marks) and optionally a link identifier. Unless you have multiple connections, you don't need the link identifier. It returns a TRUE (nonzero) integer value if the query was executed successfully *even if no rows were affected.* It returns a FALSE integer if the query was illegal or not properly executed for some other reason.
If your query was an INSERT, UPDATE, DELETE, CREATE TABLE, or DROP TABLE and returned TRUE, you can now use mysql_affected_rows to see how many rows were changed by the query. This function optionally takes a link identifier, which is only necessary if you are using multiple connections.
It *does not* take the result handle as an argument! You call the function like this, without a result handle:

```
$affected_rows = mysql_affected_rows();
```

If your query was a SELECT statement, you can use mysql_num_rows($result) to find out how

many rows were returned by a successful SELECT.
The mysqli_affected_rows and mysqli_num_rows behave exactly the same as their mysql_counterparts.

Fetching Data Sets

What actually happens is that a mysql_query() command pulls the data out of the database and sends a receipt back to PHP reporting on the status of the operation. At this point, the data exists in a purgatory that is immediately accessible from neither MySQL nor PHP — you can think of it as a staging area of sorts. The data is there, but it's waiting for the commanding officer to give the order to deploy. It requires one of the mysql_fetch functions to make the data fully available to PHP.
The fetching functions are as follows:
■■mysql_fetch_row: Returns row as an enumerated array
■■mysql_fetch_object: Returns row as an object
■■mysql_fetch_array: Returns row as an associative array
■■mysql_result: Returns one cell of data

The differences among the three main fetching functions is small. The most general one is mysql_fetch_row, which can be used something like this:

```
$query = "SELECT ID, LastName, FirstName FROM users WHERE Status = 1";
$result = mysql_query($query);
while ($name_row = mysql_fetch_row($result)) {
print("{$name_row[0]} {$name_row[1]} {$name_row[2]}<BR>\n");
}
```

This code will output the specified rows from the database.

The function mysql_fetch_object performs much the same task, except the row is returned as an object rather than an array. Obviously, this is helpful for those among the PHP brethren who utilize the object-oriented notation:

```
$query = "SELECT ID, LastName, FirstName FROM users WHERE Status = 1";
$result = mysql_query($query);
while ($row = mysql_fetch_object($result)) {
echo "{$row->ID}, {$row->LastName}, {$row->FirstName}<BR>\n";
        }
```

The most useful fetching function, mysql_fetch_array, offers the choice of results as an associative or an enumerated array — or both, which is the default. This means you can refer to outputs by database field name rather than number:

```
$query = "SELECT ID, LastName, FirstName FROM users WHERE Status = 1";
$result = mysql_query($query);
while ($row = mysql_fetch_array($result)) {
echo "{$row['ID']}, {$row['LastName']}, {$row['FirstName']}<BR>\n";
}
```

Remember that mysql_fetch_array can *also* be used exactly the same way as mysql_fetch_ row — with numerical identifiers rather than field names. By using this function, you leave yourself the option. If you want to specify offset or field name rather than making both available, you can do it like this:

```
$offset_row = mysql_fetch_array($result, MYSQL_NUM);
or
$associative_row = mysql_fetch_array($result, MYSQL_ASSOC);
```

Last and least of the fetching functions is mysql_result(). You should only even *consider* using this function in situations where you are positive you need only one piece of data to be returned from MySQL. An example of its usage is:

```
$query = “SELECT count(*) FROM personal_info”;
$db_result = mysql_query($query);
$datapoint = mysql_result($db_result, 0, 0);
```

The mysql_result function takes three arguments: *result identifier, row identifier,* and (optionally) *field*. Field can take the value of the field offset as above or its name as in an associative array (“Surname”) or its MySQL field-dot-table name (“personal_info.Surname”). Use the offset if at all possible, as it is substantially faster than the other two. Even better, don’t use this function with any frequency. A well-formed query will almost always return a specific result more efficiently.

A special MySQL function can be used with any of the fetching functions to more specifically designate the row number desired. This is mysql_data_seek, which takes as arguments the result identifier and a row number and moves the internal row pointer to that row of the data set. The most common use of this function is to reiterate through a result set from the beginning by resetting the row number to zero, similar to an array reset. This obviates another expensive database call to get data you already have sitting around on the PHP side. Here’s an example of using mysql_data_seek():

```
<?php
echo(“<TABLE>\n<TR><TH>Titles</TH></TR>\n<TR>”);
$query = “SELECT title, publisher FROM books”;
$result = mysql_query($query);
while ($book_row = mysql_fetch_array($result)) {
echo(“<TD>$book_row[0]</TD>\n”);
}
echo(“</TR></TABLE><BR>\n”);
echo(“<TABLE>\n<TR><TH>Publishers</TH></TR>\n<TR>”);
mysql_data_seek($result, 0);
while ($book_row = mysql_fetch_array($result)) {
echo(“<TD>{$book_row[1]}</TD>\n”);
}
echo(“</TR></TABLE><BR>\n”);
?>
```

Without using mysql_data_seek, the second usage of the result set would turn back no 0 rows because it has already iterated through to the end of the dataset and the pointer stays there until you explicitly move it. This handy function helps greatly when you are formatting data in a way that does not place fields in columns and records in rows.

Getting Data about Data

You only need four PHP functions to put data into or get data out of a preexisting MySQL database: mysql_connect, mysql_select_db, mysql_query, and mysql_fetch_array. Most of the rest of the functions in this section are about getting information about the data you put into or took out of the database or about the construction of the database itself. PHP offers extensive built-in functions to help you learn the name of the table in which your data resides, the data type handled by a particular column, or the number of the row into which you have just inserted data. With these functions, you can effectively work with a database about which you know very little.
The MySQL metadata functions fall into two major categories:

- Functions that return information about t he previous operation only
- Functions that return information about the database structure in general

A very commonly used example of the first type is mysql_insert_id(), which returns the auto incremented ID assigned to a row of data you just inserted.
A commonly used example of the second type is mysql_field_type(), which reveals whether a particular database field's data must be an integer, a varchar, text, or what have you.
Observe however, that this function is also deceptively named. Rather than returning the MySQL type, it returns the PHP data type. For example, an ENUM-type field will return 'string'. Use mysql_field_flags to return more specialized field information. This should be apparent when you consider that it works on a result rather than on an actual MySQL field. It would be useful to have a function that got the possible values for an ENUM field, but there isn't a canned version at this point. Instead, use a "describe table" query and parse the result using PHP's regex functions.
Most of the data-about-data functions are pretty self-explanatory. There are a couple of things to keep in mind when using them, though. First, most of these functions are only effective if used in the proper combination — don't try to use a mysql_affected_rows after a SELECT query and then wonder what went wrong. Second, be careful about security with the functions that return information about your database structure. Knowing the name and structure of each table is very valuable to a cracker. And finally, be aware that some of these functions are shopping baskets full of simplerfunctions. If you need several pieces of information about a particular result set or database, it could be faster to use mysql_fetch_field than all the mysql_field functions one after the other.
All of the MySQL metadata functions are fairly easy to use. However, their efficacy is directly related to intelligent database design rather than a mere marker of the PHP's strengths. Good database practices will make these functions useful over the long haul. The mysqli equivalent functions are perfect analogues in each of these cases.

Multiple Connections

The main time that you need to use different connections is when you're querying two or more completely separate databases. The most common situation in which you might do this is when you're using MySQL in a replicated situation. MySQL replication is accomplished through a master-slave setup, where you typically get reads from a slave and make writes to the master.
To use multiple connections, you simply open connections to each database as needed and make sure to hang on to the right result sets
In this example, we are using connections from three different databases on different servers:

```
<?php
$link1 = mysql_connect('host1', 'me', 'sesame');
mysql_select_db('userdb', $link1);
$query1 = "SELECT ID FROM usertable WHERE username = '$username'";
```

```
$result1 = mysql_query($query1, $link1);
$array1 = mysql_fetch_array($result1);
$usercount = mysql_num_rows($result1);
mysql_close($link1);
$today = ‘2002-05-01’;
$link2 = mysql_connect(‘host2’, ‘myself’, ‘benne’);
mysql_select_db(‘inventorydb’, $link2);
$query2 = “SELECT sku FROM widgets WHERE ship_date = ‘$today’“;
$result2 = mysql_query($query2, $link2);
$array2 = mysql_fetch_array($result2);
$widgetcount = mysql_num_rows($result2);
mysql_close($link2);
if ($usercount > 0 && $widgetcount > 0) {
$link3 = mysql_connect(‘host3’, ‘I’, ‘seed’);
mysql_select_db(‘salesdb’, $link3);
$query3 = “INSERT INTO saleslog (ID, date, userID, sku)
VALUES (NULL, ‘$today’, ‘$array1[0]‘, ‘$array2[0]‘)“;
$result3 = mysql_query($query3, $link3);
$insertID = mysql_insert_id($link3);
mysql_close($link3);
if ($insertID >= 1) {
print(“Perfect entry”);
}
else {
print(“Danger, danger, Will Robinson!”);
}
} else {
print(“Not enough information”);
}
?>
```
In this example, we have deliberately kept the connections as discrete as possible for clarity’s sake, even going to the trouble to close each link after we use it. Without the mysql_close() commands, we would be running multiple concurrent connections — which you may want to do. There’s nothing stopping you from doing so. Just remember to pass the link value carefully from one function to the next, and you should be fine.

Building in Error Checking

This section could have been titled “Die, die, die!” because the main error-checking function is actually
called die().

die() is not a MySQL-specific function — the PHP manual lists it in “Miscellaneous Functions.” It
simply terminates the script (or a delimited portion thereof) and returns a string of your choice.

```
mysql_query(“SELECT * FROM mutual_funds WHERE code = ‘$searchstring’“) or die(“Please
check your query and try again.”);
```

Notice the syntax: the word or (you could alternatively use ||, but that isn't as much fun as saying or die) and only one semicolon per pair of alternatives.

Other built-in means of error-checking are error messages. These are particularly helpful during the
development and debugging phase, and they can be easily commented out in the final edit before going live on a production server. As mentioned, MySQL error messages no longer appear by default.
If you want them, you have to ask for them by using the functions mysql_errno()

```
if (!mysql_select_db($bad_db)) {
print(mysql_error());
}
```

Creating MySQL Databases with PHP

For those times when you need to create databases programmatically, the relevant functions are:
■■mysql_create_db(): Creates a database on the designated host, with name specified in arguments
■■mysql_drop_db(): Deletes the specified database
■■mysql_query(): Passes table definitions and drops in this function
A bare-bones database-generation script might look like this:

```
<?php
$linkID = mysql_connect('localhost', 'root', 'sesame');
mysql_create_db('new_db', $linkID);
mysql_select_db('new_db');
$query = "CREATE TABLE new_table (
id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
new_col VARCHAR(25)
)";
$result = mysql_query($query);
$axe = mysql_drop_db('new_db');
?>
```

PHP -MySQL Functions

Function Name and its Usage

mysql_affected_rows([link_id])-------- Use after a nonzero INSERT, UPDATE, or DELETE query to check number of rows changed.

mysql_change_user(user, password[, database][, link_id])----Changes MySQL user on an open link.
mysql_close([link_id]) Closes the identified link (usually unnecessary).
mysql_connect([host][:port][:socket][, username][, password])---------------Opens a link on the

specified host, port,socket; as specified user with password.All arguments are optional.
mysql_create_db(db_name[, link_id]) -------------Creates a new MySQL database on the host associated with the nearest open link.
mysql_data_seek(result_id, row_num)---------------- Moves internal row pointer to specified row number. Use a fetching function to return data from that row.
mysql_drop_db(db_name[, link_id])------ Drops specified MySQL database.
mysql_errno([link_id])-------- Returns ID of error.
mysql_error([link_id]) ------------Returns text error message.
mysql_fetch_array(result_id[, result_type])------------ Fetches result set as associative array.
mysql_fetch_field(result_id[, field_offset]) -------------------Returns information about a field as an
object.
mysql_fetch_lengths(result_id) --------------Returns length of each field in a result set.
mysql_fetch_object(result_id[, result_type]) ----------------Fetches result set as an object. See mysql_fetch_array for result types.
mysql_fetch_row(result_id) ------------Fetches result set as an enumerated array.
mysql_field_name(result_id, field_index) ---------------Returns name of enumerated field.
mysql_field_seek(result_id, field_offset) ------------------Moves result pointer to specified field offset. Used with mysql_fetch_field.
mysql_field_table(result_id, field_offset) --------------Returns name of specified field's table.
mysql_field_type(result_id, field_offset) ------------Returns type of offset field (for example, TINYINT, BLOB, VARCHAR).
mysql_field_flags(result_id, field_offset) ---------------------Returns flags associated with enumerated
field (for example, NOT NULL, AUTO_INCREMENT, BINARY).
mysql_field_len(result_id, field_offset) --------------Returns length of enumerated field.
mysql_free_result(result_id)----------------- Frees memory used by result set (usually unnecessary).
mysql_insert_id([link_id]) Returns--------------------- AUTO_INCREMENTED ID of INSERT; or FALSE if insert failed or last query was not an insert.
mysql_list_fields(database, table[, link_id])----------- Returns result ID for use in mysql_field functions, without performing an actual query.
mysql_list_dbs([link_id]) -------------------Returns result pointer of databases on mysqld. Used with mysql_tablename.
mysql_list_tables(database[, link_id])----------- Returns result pointer of tables in database. Used with mysql_ tablename.
mysql_num_fields(result_id) ------------Returns number of fields in a result set.
mysql_num_rows(result_id) -----------------Returns number of rows in a result set.
mysql_pconnect([host][:port][:socket][,username][, password])-----------Opens persistent connection to database.All arguments are optional. Be careful —mysql_close and script termination will not close the connection.
mysql_query(query_string[, link_id]) -----------Sends query to database. Remember to put the semicolon outside the double quoted query string.
mysql_result(result_id, row_id, field_identifier)------------Returns single-field result. Field identifier can be field offset (0), field name (FirstName) or table-dot name (myfield.mytable).
mysql_select_db(database[, link_id]) -------------Selects database for queries.
mysql_tablename(result_id, table_id) --------------Used with any of the mysql_list functions to return the value referenced by a result pointer

Sessions

Why to use sessions

1. We want to customize our users' experiences as they move through the site, in a way that depends on which (or how many) pages they have already seen.
2. We want to display advertisements to the user, but we do not want to display a given ad more than once per session.
3. We want the session to accumulate information about users' actions as they progress — as in an adventure game's tracking of points and weapons accumulated or an e-commerce site's shopping cart.
4. We are interested in tracking how people navigate through our site in general — when they visit that interior page, is it because they bookmarked it, or did they get there all the way from the front page?

Other Alternatives for sessions

IP address

Web servers usually know either the Internet hostname or the IP address of the client that is requesting
a page. In many configurations of PHP, these show up for free as variables — $_SERVER['REMOTE_HOST'] and $_SERVER['REMOTE_ADDR'], respectively. Now you might think that the identity of the machine at the other end is a reasonable stand-in for the person at the other end, at least over the short term. If you get two requests in quick succession from the same IP address, your code can safely conclude that the same person followed a link or form from one of your site's pages to another. Unfortunately, the IP address your browser knows about may not belong to the machine your user is browsing from. In particular, AOL and other large operations employ proxy servers, which act as intermediaries. Your user's browser actually requests a URL from the proxy server, which in turn requests the page from your server and then forwards back the page to the user.

Hidden variables

Every HTTP request is dealt with independently, but each time your user moves from page to page
within your site, it is usually via either a link or a form submission. If the very first page a user visits
can somehow generate a unique label for that visit, every subsequent "handoff" of one page to another can pass that unique identifier along. However, this approach to sessions is a pain to maintain — you must make sure that *every* link and form submission propagates the information as described, or the session identifier will be dropped. Also, if you send the information as GET

arguments, your session-tracking machinery will be visible in the web-address box of your user's browser, and such arguments are easily edited by the user.

Cookie-based home-grown sessions

A cookie is a special kind of file, located in the filesystem of your user's browsing computer, that web servers can read from and write to. Rather than checking for a passed GET/POST variable (and
assigning a new identifier if none is found), your script checks the user's machine for a previously
written cookie file and stores a new identifier in a new cookie file if none is found or if the old cookie
has expired. This method has some benefits over using hidden variables: The mechanism works behind the scenes (typically, not showing any trace of its activity in the browser window), and the
code that checks or sets the cookie can be centralized (rather than affecting every form and link). What's the drawback? Some very old browsers do not support cookies at all, and more recent browsers allow users to deny cookie-setting privileges to web servers. So, although cookies make for a
smooth solution, we can't assume that they are always available.

How Sessions Work in PHP

The first step in a script that uses the session feature is to let PHP know that a session may already
be in progress so that it can hook up to it and recover any associated information. This is done by calling the function session_start(), which takes no arguments. Also, any call to session_register() causes an implicit initial call to session_start().
The effect of session_start() depends on whether PHP can locate a previous session identifier, as supplied either by HTTP arguments or in a cookie. If one is found, the values of any previously
registered session variables are recovered. If one is not found, then PHP assumes that we are in the
first page of a new session, and generates a new session ID.the $_SESSION superglobal array as your suitcase for storing anything that you want to retrieve again from a later page in the same session. Assume that any other variables will be left behind when you leave this page and that everything in that suitcase will be there when you arrive at the next page.
So, session code to propagate a single numerical variable can be as simple as this:

```
<?php
session_start();
$temporary_number = 45;
$save_this_one = 19;
```

```
$another_temporary = 33;
$_SESSION[‘save_this’] = $save_this_one;
?>
```
The receiving code can be as simple as the following example:
```
<?php
session_start();
$saved_from_prev_page = $_SESSION[‘save_this’];
[..]
$temporary_number = 45;
$another_temporary = 33;
[..]
?>
```
That’s all there is to it. Assignment into the $_SESSION superglobal array implicitly does any registration
necessary for the new value to be carried forward to the next page.

Sample Session Code

In this listing, we perform the following tasks:
Initiate a session (or pick up an existing one) by using session_start().check for the existence of a preexisting entry in $_SESSION. If one is not present, we assume that the session is new. Increment a counter that tracks how many times that the user has visited this page. Store the incremented counter back in $_SESSION.Provide a link back to the page itself, embedding the session ID as an argument if it is found.
```
<?php
session_start();
?>
<HTML><HEAD><TITLE>Greetings</TITLE></HEAD>
<BODY>
<H2>Welcome to the Center for Content-free Hospitality</H2>
<?php
if (!IsSet($_SESSION[‘visit_count’])) {
echo “Hello, you must have just arrived.
Welcome!<BR>”;
$_SESSION[‘visit_count’] = 1;
}
else {
$visit_count = $_SESSION[‘visit_count’] + 1;
echo “Back again are ya? That makes $visit_count times now “.
“(not that anyone’s counting)<BR>”;
$_SESSION[‘visit_count’] = $visit_count;
}
$self_url = $_SERVER[‘PHP_SELF’];
$session_id = SID;
if (IsSet($session_id) &&
$session_id) {
$href = “$self_url?$session_id”;
}
else {
$href = $self_url;
}
echo “<BR><A HREF=\“$href\“>Visit us again</A> sometime”;
?>
</BODY></HTML>
```

Session Functions

Function Behavior

session_start()--------------------------

Takes no arguments and causes PHP either to notice a session ID that has been passed to it (via a cookie or GET/POST) or to create a new session ID if none is found.If an old session ID is found, PHP retrieves the assignments of all variables that have been registered and makes those assigned variables available as regular global variables.

session_register()

Takes a string as argument and registers the variable named by the string — for example, session_register(‘username’). If session_start() has not yet been called, session_register will implicitly call it before executing.

session_unregister()------------------

Takes a string variable name as argument and unregisters the corresponding variable from the session. As a result, the variable binding will no longer be serialized and propagated to later pages. (The variable-name string should not include the leading $.)

session_destroy()--------------------

Calling this function gets rid of all session variable information that has been stored.It does not unset any variables in the current script or the session cookie.

session_unset()--------------

Takes no arguments, and frees all variables in the session. Dangerous if using $_SESSION or $HTTP_SESSION_VARS; use unset() instead.

session_write_close()-------------------

Manually close session and release write lock on data file. Useful with frames, some clustering situations, and if you do something that might cause PHP to not realize the session has terminated (such as redirection).

session_name()-----------------

When called with no arguments, returns the current session-name string. This is usually ‘PHPSESSID’ by default. When called with one string argument, session_name() sets the current sessionname to that string. This name is used as a key to find the session ID in cookies and GET/POST arguments — for successful retrieval,
session_module_name()---------------
If given no arguments, returns the name of the module that is responsible for handling session data. This name currently defaults to ‘files’, meaning that session bindings are serialized and then written to files in the directory named by the function
session_save_path().--------------------
If given a string argument, changes the module name to that string.
session_ save_path()------------------
Returns (or sets, if given an argument) the pathname of the directory to which session variable-binding files will be written (which typically defaults to /tmp on Unix systems).
session_id() -----------------
Takes no arguments and returns a string, which is the unique key corresponding to a particular session. If given a string argument, will set the session ID to that string.
session_regenerate_id()----------
Takes no arguments and sets a new session ID, setting a new cookie if necessary and returning TRUE on success or FALSE on failure. Unlike session_id(), it does not return a string with the actual new ID.
session_encode()----------------
Returns a string encoding of the state of the current session, suitable for use by string_decode(). This can be used for saving a session for revival at some later time, such as by writing the encoded string to a file or database.
session_decode()-----------
Takes a string encoding as produced by session_encode() and reestablishes the session state, turning session bindings into page bindings as session_start() does
session_ get_cookie_params() ------------
Returns an array with current session cookie data: lifetime (in seconds till expiration, or 0 for no expiration), path (for which the cookie is valid), domain (for which the cookie is valid), secure (whether or not the cookie will only be sent over SSL connections). These parameters are normally set in the php.ini file, but can be changed for a single script through the session_set_cookie_params() function.
session_set_cookie_params()-----------------
Takes four arguments: int lifetime (in seconds till expiration, or 0 for no expiration),string path (for which the cookie is valid), string domain (for which the cookieis valid), boolean secure (whether or not the cookie will only be sent over SSL connections).

Session Configuration Variables

Description	Typical Value	Php.ini Variable

Pathname for the server-side directory where session datafiles will be written. Must be changed for Windows systems!	tmp under Unixsystems	session.save_path
When 1, sessions will initialize automatically every time a script loads. When 0, no session data will be available unless there is an explicit call to either session_start() or ession_register().	0	session.auto_start
string that determines underlying method for saving session variable information. Changing this is not recommended for the casual user.		session.save_handler‘files’, ‘user’
Specifies how long session cookies take to expire and,consequently, the lifetime of a session. The default of 0 means that sessions last until the browser is closed any other value indicates the number of seconds the session is allowed to live.	0	session.cookie_lifetime
If 1, the session mechanism will attempt to propagate thesession ID by setting/checking a cookie. (If the browser refuses the cookie, then GET/POST vars may be used.) If this variable is 0, no attempt to use cookies is made.	1	session.use_cookies

Cookies

A *cookie* is a small piece of information that is retained on the client machine, either in the browser's application memory or as a small file written to the user's hard disk. It contains a name/value pair — *setting a cookie* means associating a value with a name and storing that pairing on the client side. *Getting* or *reading* a cookie means using the name to retrieve the value.

In PHP, cookies are set using the setcookie() function, and cookies are read nearly automatically.In PHP4.1 and later, names and values of cookie variables show up in the superglobal array _COOKIES, with the cookie name as an index, and the value as the value it indexes.

The setcookie() function
There is just one cookie-related function, called setcookie().

Arguments to setcookie()

The name of your cookie (analogous to the name of a variable).value string The value you want to store in the cookie (analogous to the value you would assign to a variable). If this argument is not supplied, the cookie named by the first argument is deleted.expire int Specifies when this cookie should expire. A value of 0 (the default) means that it should last until the browser is closed. Any other integer is interpreted as an absolute time (as returned by the function mktime()) when the cookie should expire.
path string In the default case, any page within the web root folder would see (and be able to set) this named cookie. Setting the path to a subdirectory (for example, "/forum/") allows distinguishing cookies
that have the same name but are set by different *sites* or subareas of the web server (in this example, the cookie will only be valid in the forum area). Be sure to include a trailing slash in the path. httponly boolean Cookies set with this flag are only sent through HTTP requests. Default is FALSE. domain string In the default case, no check is made against the domain requested by the client. If this argument is nonempty, then the domain must match. For example, If the same server serves www. mysteryguide.com and forum.mysteryguide.com, one site's code can ensure that the other site does not read (or set) its cookies by including this argument as "forum.mysteryguide.com." secure boolean
(TRUE (1) or FALSE (0)) Defaults to 0 (FALSE). If this argument is 1 or TRUE, the cookie will only be sent over a secure socket (aka SSL or HTTPS) connection. Note that a secure connection must already be running for such a cookie to be set in the first place.

Examples
This section provides some example calls to setcookie(), along with comments, such as the following:
setcookie('membername', 'timboy');
This sets a cookie called membername, with a value of timboy. Because there are no arguments

except for the first two, the cookie will persist only until the current browser program is closed, and it will be read on subsequent page requests from this browser to this server, regardless of the domain name in the request or from where in the web root file hierarchy the page is served. The cookie will also be read regardless of whether the web connection is secure. For example, consider
the following call:

```
setcookie('membername', 'troutgirl', time() + (60 * 60 * 24),
"/", "www.troutworks.com", 1);
```

Deleting cookies

Deleting a cookie is easy. Simply call setcookie(), with the exact same arguments as when you set it, except the value, which should be set to an empty string. This does not set the cookie's value
to an empty string — it actually removes the cookie. Remember: If you used the path or domain arguments to set the cookie, you need to use them to unset the cookie too. Another method to clear
cookies is to set the expiration time in the past.

Reading cookies

Cookies that have been successfully set in a browser or user's machine will automatically be read on
the next request from that browser.
This has the following effects:
In PHP4.1 and later, the cookie's name/value pair will be added to the superglobal array $_COOKIE, as though we had evaluated $_COOKIE['name'] = value.

If the register_globals directive is turned on (for versions earlier than PHP6), a regular page-level global variable will be set to the cookie's value, named the same as the cookie's name. Because register_globals is turned off by default starting with PHP4.2 this feature is not available in 4.2 or later, unless either you or your ISP's administrator has changed the configuration. So, for example, you can set a cookie as follows:

```
setcookie('membername', 'timboy');
```

This means that, on a *later* page access, you might be able to print the value again as easily as this:

```
$membername = $_COOKIE['membername'];
print("The member name is $membername<BR>");
```

And, if register_globals has been turned on, the later page's use of the cookie becomes even simpler:

```
print("The member name is $membername<BR>");
```

Sending HTTP Headers

The setcookie() call provides a wrapper around a particular usage of HTTP headers. In addition, PHP offers the header() function, which you can use to send raw, arbitrary HTTP headers. You can
use this function to roll your own cookie function if you like, but you can also use it to take advantage
of any other kind of header-controlled functionality. The syntax of header() is as simple as it can be: It takes a single string argument, which is the
header to be sent.

Example: Redirection
One useful kind of HTTP header is "Location:", which can act as a redirector. Simply put a fully qualified URL after the "Location:" string, and the browser will start over again with the new address instead. Here's an example:

```
<?php
if (IsSet($_GET['gender']) && ($_GET['gender'] == "female"))
{
header(
"Location: http://www.example.com/secret.php");
exit;
}
?>
<HTML><HEAD><TITLE>The inclusive page</TITLE></HEAD></HTML>
<BODY>
<H3>Welcome!</H3>
We welcome anyone to this page, even men! Talk amongst yourselves.
</BODY></HTML>
```

If we simply enter the URL for this page (www.example.com/inclusive.php), we will see the rendering of the HTML at the bottom of the script. On the other hand, if we include the right GET argument (www.example.com/inclusive.php?gender=female), we find ourselves redirected to a different page entirely. Note that this is significantly different from selectively importing contents with the include() statement — we actually end up browsing a different URL than the one we typed in, and that new web address is what shows up in the Location or Address bar of your browse

Sending E-Mail with PHP

Windows configuration
In Windows, you need to set two variables in the php.ini file:
■■SMTP: A string containing the DNS name or IP address of an
SMTP server that relays for the Windows machine on which PHP

is installed. If it is on the PHP server, specify localhost.
■■sendmail_from: A string containing the e-mail address of your
default PHP mail sender (for example, mailbot@example.com).

Linux configuration

You need to check and possibly change one variable in the php.ini file if you're using Unix: sendmail_path, a string containing the full path to your sendmail program (usually /usr/ sbin/sendmail or /usr/lib/sendmail), a replacement, or a wrapper (such as /var/qmail/ bin/sendmail).

The mail function

The mail() function is the primary function used to send mail with PHP. This function, which returns a Boolean, attempts to send one message using the data within the parentheses. The simplest
use of this function (keeping in mind that this is a dummy address and should not be used for testing
purposes) is:

```
<?php
mail('receiver@example.com', 'A Sample Subject Line',
"Body of e-mail\r\nwith lines separated by the newline
character.");
?>
```

This is the default and minimum format: address of recipient, subject line, and body. In this case, PHP will automatically add a From: me@sendhost line to each message header.
You can also, as always, use variables instead of hardcoded values:

```
<?php
$address = 'santa@example.com';
$subject = 'All I want for Christmas';
$body = "Is my two front teeth.\r\nSincerely, Joey";
$mailsend = mail($address, $subject, $body);
echo $mailsend;
?>
```

Multiple Recipients

Multiple recipients all go into the address field, with commas separating them (this feature is not supported by all MTAs; if you want to be sure, use cc: instead):

```
<?php
$address1 = 'receiver@receipthost';
$address2 = 'jane@example.com';
```

```
$address3 = ‘john@example.org’;
$all_addresses = “$address1, $address2, $address3”;
$subject = ‘A Sample Subject Line’;
$body = “Body of e-mail\r\nwith lines separated by the
newline character.”;
$mailsend = mail($addresses, $subject, $body);
echo $mailsend;
?>
```

Most people would like more control over the addresses, appearance, and format of their e-mails. You can do that by putting an additional header *after* the three default headers.

```
<?php
$address = ‘receiver@receipthost’;
$subject = ‘A Sample Subject Line’;
$body = “Body of e-mail\r\nwith lines separated by the newline
character.”;
$extra_header_str = “From: me@sendhost\n\nbcc:
phb@sendhost\r\nContent-type: text/plain\r\nX-mailer: PHP/“
. phpversion();
$mailsend = mail($address, $subject, $body, $extra_header_str);
echo $mailsend;
?>
```

This “additional header” field is somewhat odd because it crams in several types of information that
would normally be given their own fields. Ours is not to wonder why; ours is but to explain the kinds of things you might want to put in this field.

■■Your name
■■The To: address
■■Your e-mail address
■■A reply-to or bounce-to address
■■X-mailer and version number
■■MIME version
■■Content-type
■■Charset (which uses a = to assign a value and not a : like the other headers)
■■ transfer-encoding
■■Carbon-Copy (cc:) and blind carboncopy (bcc:) recipients

Passing values from database to forms

PHP is brilliant at putting variables into a database, but it really shines when taking data from a database, displaying it in a form to be edited, and then putting it back in the database. Its HTMLembeddedness, easy variable passing, and slick database connectivity are at their best in this kind of job. These techniques are extremely useful, because you will find a million occasions to edit data you’re storing in a database.

Consider the example where the user enters student name and views his information for updation in the database

```
<html>
<head>
</head>
<body>
<form name="f1" action="getdata.php" method="post">
Enter name <input type="text" name="n1">
<input type="submit" value="view profile">
</form>
</body>
</html>
```

getdata.php

```
<?php
$s=$_POST['n1'];
$dbcnx=mysql_connect("localhost","root","");
if(!$dbcnx)
{
	echo("unable to connect to database");
	exit();
}
mysql_select_db("student",$dbcnx);
$sql="select * from stud where sname='".$s."'";
$result=mysql_query($sql);
?>
<form method="post" action="update.php">
<?php
while($row1=mysql_fetch_array($result))
{
echo "Student name  <input type='text' name='n1' value='";
echo $row1['sname'];
echo "'> <br>";
echo "Marks<input type='text' name='n1' value='";
echo $row1['marks'];
echo "'><br>";
echo "class<input type='text' name='n1' value='";
echo $row1['class'];
echo "'><br>";
}
?>
<input type="submit" value="update">
</form>
```

Examining Regular Expressions

Tokenizing and Parsing Functions

The process of breaking up a long string into words is called tokenizing.
PHP offers a special function for this purpose, called strtok().
The **strtok() function** takes two arguments: the string to be broken up into tokens and a string containing all the delimiters. On the first call, both arguments are used, and the string value returned is the first token. To retrieve subsequent tokens, make the same call, but omit the source string argument. It will be remembered as the current string, and the function will remember where it left off. For example:

```
$token = strtok("This is a php regular expression example"," ");
while($token){
 echo $token ."<BR>";
 $token = strtok(" " );
}
```

will print

```
This
is
a
php
regular
expression
example
```

You can also use the **explode() function** to do something similar, except that it stores the tokens all at once in an array. After the tokens are in the array, you can do anything you like with them, including sort them. The explode() function takes two arguments: a separator string and the string to be separated. It returns an array where each element is a substring between instances of the separator in the string to be separated. For example:

```
$explode_result = explode("AND","one AND  two AND  three     ");

echo count($explode_result) ;
```

The output is 3

The explode() function has an inverse function, **implode(),** which takes two arguments: a “glue” string (analogous to the separator string in explode()) and an array of strings like that returned by explode(). It returns a string created by inserting the glue string between each string element in the array For example:

```
$a=implode("arial , times",$explode_result);

echo $a;
```

the output is

```
three arial , times two arial , timesone
```

Why Regular Expressions?

you want to test strings to see if they are a particular kind of web hostname: addresses that start with www. and end with .com, and have one lowercase alphabetic word in the middle. For example,
these are strings we want:
‘www.ibm.com’
‘www.zend.com’
And the following are not:
‘java.sun.com’
‘www.java.sun.com’
‘www.php.net’
‘www.IBM.com’
‘www.Web addresses can’t have spaces.com’
With a little thought, it’s obvious that there is no convenient way to simply use string and substring comparison to build the test that we want. We can test for the presence of www. and .com, but it is difficult to enforce what should be happening between them. This is what regular expressions are good for.

Regex in PHP

Regular expressions (or regex, pronounced with a soft g by your authors, but with no consensus pronunciation) are patterns for string matching, with special wildcards that can match entire portions of the target string. There are two broad classes of regular expression that PHP works with: POSIX (extended) regex and Perl-compatible regex. The differences mostly have to do with syntax, although there are some functional differences, too.
POSIX-style regular expressions are ultimately descended from the regex pattern-matching machinery used in Unix command-line shells;
Perl-compatible regex is a more direct imitation of regular
expressions in Perl.

An example of POSIX-style regex
Here are a few of the rules for POSIX-style regular expressions, simplified:
■ Characters that are not special are matched literally. The letter a in a pattern, for example,matches the same letter in a target string.
■ The special character ^ matches the beginning of a string only, and the special character $matches the end of a string only.
■ The special character . matches any character.
■ The special character * matches zero or more instances of the previous regular expression,and + matches one or more instances of the previous expression.
■ A set of characters enclosed in square brackets matches any of those characters — the pattern[ab] matches either a or b. You can also specify a range of characters in brackets by using a hyphen — the pattern [a-c] matches a, b, or c.
■ Special characters that are escaped with a backslash (\) lose their special meaning and are matched literally

Consider the example of .com

```
^www\.[a-z]+\.com$
```

In this expression we have the '^' symbol, which says that the www portion must start at the beginning of the string. Then comes a dot (.), preceded by a backslash that says we really want a dot, not the special . wildcard character. Then we have a bracket-enclosed range of all the lowercase alphabetic letters. The following + indicates that we are willing to match any number of these lowercase letters in a row, as long as we have at least one of them. Then another literal ., the com, and the special $ that says that com is the end of it.

Regular expression function

POSIX-style regular expression

This functions are included for legacy applications where you might find them still being used. These functions are no longer in PHP6 and have been replaced with preg functions,

ereg()----------- Takes two string arguments and an optional third-array argument. The first string is the POSIX-style regular expression pattern, and the second string is the target string that is being matched. The function returns TRUE if the match was successful and FALSE otherwise. In addition, if an array argument is supplied and portions of the pattern are enclosed in parentheses, the parts of the target string that match successive parenthesized portions will be copied into successive
elements of the array.
ereg_replace() -------------Takes three arguments: a POSIX regular expression pattern, a string to do replacement with, and a string to replace into. The function scans the third argument for portions that match the pattern and replaces them with the second argument. The modified string is returned. If there are parenthesized portions of the pattern (as with ereg()), the replacement string may contain special substrings of the form \\digit (that is, two backslashes followed by a single-digit number), which will themselves be replaced with the corresponding piece of the target string.
eregi()----------- Identical to ereg(), except that letters in regular expressions are matched in a case-independent way.
eregi_replace()---------- Identical to ereg_replace(), except that letters in regular expressions are matched in a case-independent way.
split() ----------Takes a pattern, a target string, and an optional limit on the number of portions to split the string into. Returns an array of strings created by splitting the target string into chunks delimited by substrings that match the regular expression. spliti() ------------Case-independent version of split().

Perl-Compatible Regular Expressions

Perl-compatible regex in PHP has a completely distinct set of functions and a slightly different set of rules for patterns. Perl-compatible regex patterns are always bookended by one particular character, which must be
the same at beginning and end, indicating the beginning and end of the pattern. By convention, this is most often the / character, although you can use a different character if you so desire. The Perlcompatible
pattern:
/pattern/

matches any string that has the string (or substring) pattern in it. To make things slightly more complicated, these patterns are typically strings, and PHP needs its own quotes to recognize such strings. So if you are putting a pattern into a variable for later use, you might well do this:
$my_pattern = '/pattern/';
This variable would now be suitable for passing off to a Perl-compatible regex function that expects a pattern as argument.

Common Perl-Compatible Pattern Constructs

Simple literal character matches-------If the character involved is not special, Perl will match characters in sequence. The example pattern /abc/ matches any string that has the substring 'abc' in it.
Character class matches: [<list of characters>]------ Will match a single instance of any of the characters between the brackets. For example, /[xyz]/ matches a single character, as long as that character is either x, y, or z. A sequence of characters (in ASCII order) is indicated by a hyphen, so that a class matching all digits is [0-9].
Predefined character class abbreviations--------The patterns \d will match a single digit (from the character class [0-9]), and the pattern \s matches any whitespace character.
Multiplier patterns -------------------Any pattern followed by * means: "Match this pattern 0 or more times." Any pattern followed by ? means: "Match this pattern exactly once." Any pattern followed by + means: "Match this pattern 1 or more times."
Anchoring characters------------ The caret character ^ at the beginning of a pattern means that the pattern must start at the beginning of the string; the $ character at the end of a pattern means that the pattern must end at the end of the string. The caret character at the beginning of a character class [^abc] means that the set is the complement of the
characters listed (that is, any character that is not in the list).
Escape character-------------- '\' Any character that has a special meaning to regex can be treated as a simple matching character by preceding it with a backslash. The special characters that might need this treatment are:
. \ + * ? [ ] ^ $ ( ) { } = ! < > | :
Parentheses ---------------A parenthesis grouping around a portion of any pattern means: "Add the substring that matches this pattern to the list of substring matches."

Take, as an example, the following pattern:
/phone number\s+(\d\d\d\d\d\d\d)/
It matches any string that contains the literal phrase phone number, followed by some number of spaces (but at least one), followed by exactly seven digits (no spaces, no dash). In addition, because of the parentheses, the seven-digit number is saved and returned in an array containing substring matches if it is called from a function that returns such things.

Perl-Compatible Regular Expression Functions

preg_match() -----------Takes a regex pattern as first argument, a string to match against as second argument, and an optional array variable for returned matches. Returns 0 if no matches are found, and 1 if a match is found. If a match is successful, the array variable contains the entire matching substring as its first element, and subsequent elements contain portions matching parenthesized portions of the pattern. As of PHP 4.3.0, an optional flag of PREG_OFFSET_CAPTURE is also available. This flag causes preg match to return into the specified array a two-element array for each match, consisting of the match itself and the offset where the match occurs.
preg_match_all()------- Like preg_match(), except that it makes all possible successive matches of the pattern in the string, rather than just the first. The return value is the number of matches successfully made. The array of matches is not optional.The structure of the array returned depends on the optional fourth argument (either the constant PREG_PATTERN_ORDER, or PREG_SET_ORDER, defaulting to the former). PREG_OFFSET_CAPTURE is also available with this function.
preg_split()------------- Takes a pattern as first argument and a string to match as second argument. Returns an array containing the string divided into substrings, split along boundary strings matching the pattern. An optional third argument (limit) controls how many elements to split before returning the list; -1 means no limit. An optional flag in the fourth position can be PREG_SPLIT_NO_EMPTY causing the function to return only nonempty pieces, PREG_SPLIT_DELIM_CAPTURE causing any parenthesized expression in the delimiter pattern to be returned, or PREG_SPLIT_OFFSET_CAPTURE, which does the same as PREG_OFFSET_CAPTURE.
preg_replace() -----------Takes a pattern, a replacement string, and a string to modify. Returns the result of replacing every matching portion of the modifiable string with the replacement string. An optional limit argument determines how many replacements will occur (as in preg_split()).
preg_replace_callback()----------Like preg_replace(), except that the second argument is the name of a callback function, rather than a replacement string. This function should return the string that is to be used as a replacement.
preg_grep() ---------------Takes a pattern and an array and returns an array of the elements of the input array that matched the pattern. Surviving values of the new array have the same keys as in the input array.
preg_quote()------------- A special-purpose function for inserting escape characters into strings that are intended for use as regex patterns. The only required argument is a string to escape; the return value is that string with every special regex character preceded by a backslash.

Advanced String Functions

HTML functions

PHP offers a number of web-specific functions for string manipulation
htmlspecialchars()-------- Takes a string as argument and returns the string with replacements for four characters that have special meaning in HTML. Each of these characters is replaced with the corresponding HTML entity, so that it will look like the original when rendered by a browser. The & character is replaced by &amp; “” (the double-quote character) is replaced by &quot;; < is replaced by &lt;; > is replaced by &gt;.
htmlentities() -----------Goes further than htmlspecialchars(), in that it replaces all characters that have a corresponding HTML entity with that HTML entity.
get_html_translation_table()----------Takes one of two special constants (HTML_SPECIAL_CHARS and HTML_ENTITIES), and returns the translation table used by htmlspecialchars()and htmlentities(), espectively. The translation table is an array where keys are the character strings and the corresponding values are their replacements.
nl2br() --------------------------Takes a string as argument and returns that string with <br /> inserted before all new lines (\n, \r or \r\n). This is helpful, for example, in maintaining the apparent line length of text paragraphs when they are displayed in a browser.
strip_tags()------- Takes a string as argument and does its best to return that string stripped of all HTML tags and all PHP tags.

Strings as character collections

PHP offers some pretty specialized functions that treat strings more as collections of characters than as sequences.
The first is strspn(), which you can use to see what portion of a string is composed only of a given set of characters. For example:
$twister = “Peter Piper picked a peck of pickled peppers”;
$charset = “Peter picked a”;
print(“The segment matching ‘$charset’ is “ . strspn($twister, $charset) . “ characters long”);
gives us:
The segment matching ‘Peter picked a’ is 26 characters long because the first character not found in $charset is the o in of, and there are 26 characters that precede it.

The strcspn() function (where that internal c stands for complement) does the same thing, except that it accepts characters that are not in the character set argument. For example, the statement:
echo(strcspn($twister, “abcd”));
prints the number 14, because it accepts a 14-character sequence with the last character being the c in picked.

The count_chars() function returns a report on the occurrences of characters in its string argument, packaged as an array where the keys are the ASCII values of characters, and the values are the frequencies of those characters in the string. The second argument to count_chars() is an integer that determines which of several modes the results should be returned in. In mode 0, an array of key/value pairs is returned, where the keys are every ASCII value from 0 to 255, and the corresponding values are the frequencies of each character in the string. Modes 1 and 2 are variants that include only ASCII values that occurred in the string (mode 1) or that did not occur (mode 2).Finally, modes 3 and 4 return a string instead of an array, where the string contains all characters that occur (mode 3) or do not occur (mode 4).

```
$twister = “Peter Piper picked a peck of pickled peppers”;
print(“$twister<BR>”);
$letter_array = count_chars($twister, 1);
while ($cell = each($letter_array)){
 $letter = chr($cell[‘key’]);
 $frequency = $cell[‘value’];
 print(“Character: ‘$letter’; frequency: $frequency<BR>”);
}
```

String similarity functions

If the kind of similarity you want is similarity of spelling, consider the Levenshtein metric. The levenshtein() function takes two strings and returns the minimum number of additions, deletions, and replacements of letters needed to transform one into the other. For example:

- levenshtein(‘Tim’, ‘Time’) returns 1.
- levenshtein(‘boy’, ‘chefboyardee’) returns 9.
- levenshtein(‘never’, ‘clever’) returns 2.

If the similarity you are interested in is phonetic, consider the functions soundex() and metaphone(). Both of them take an input string and return a key string representing the pronunciation category of the word (in English). If two input word strings map to exactly the same output key, they most likely have a similar pronunciation.