OVERVIEW OF PHP DATA TYPES AND CONCEPTS:

Embedding PHP Code in Your Web Pages:

One of PHP's advantages is that you can embed PHP code directly alongside HTML. For the code to do anything, the page must be passed to the PHP engine for interpretation. But the Web server doesn't just pass every page; rather, it passes only those pages identified by a specific file extension (typically .php).

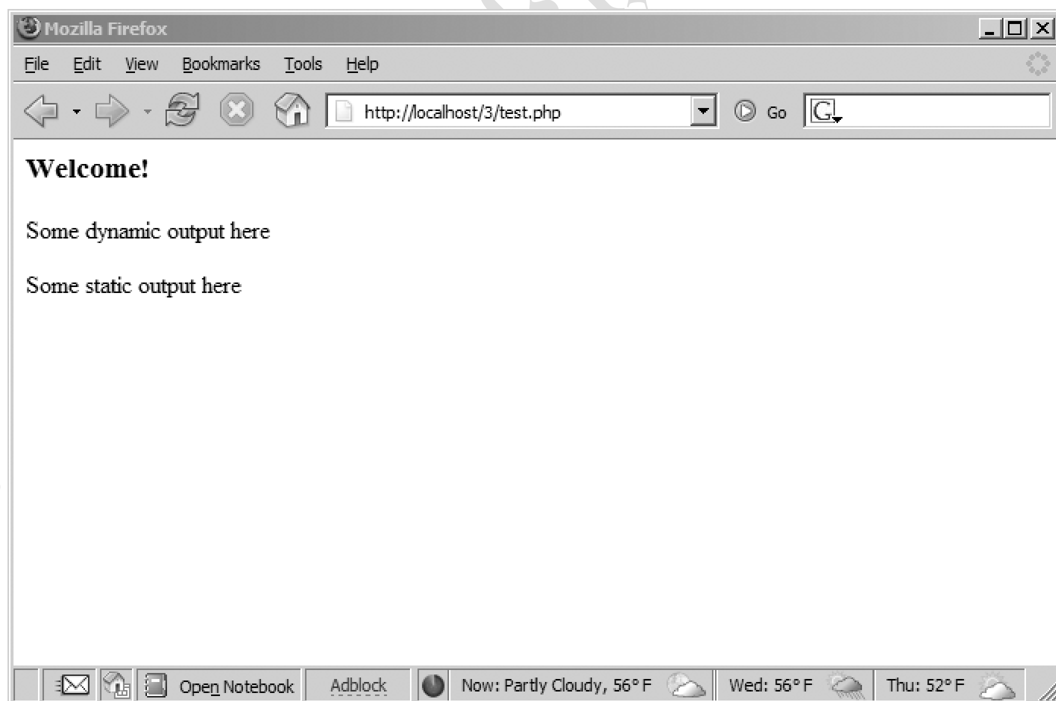
But even selectively passing only certain pages to the engine would nonetheless be highly inefficient for the engine to consider every line as a potential PHP command. Therefore, the engine needs some means to immediately determine which areas of the page are PHP-enabled. This is logically accomplished by delimiting the PHP code. There are four delimitation variants:

Default Syntax:

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

```
<h3>Welcome!</h3>
<?php
echo "<p>Some dynamic output here</p>";
?>
<p>Some static output here</p>
```

If you save this code as *test.php* and execute it from a PHP-enabled Web server, we will see the following output:



Short-Tags:

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

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

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

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

This is functionally equivalent to both of the following variations:

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

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

Script:

Historically, certain editors, Microsoft's FrontPage editor in particular, have had problems dealing with escape syntax such as that employed by PHP. Therefore, support for another mainstream delimiter variant, *<script>*, is offered:

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

Note: Microsoft's FrontPage editor also recognizes ASP-style delimiter syntax

ASP Style:

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

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

Note: ASP-style syntax was removed as of PHP 6

Embedding Multiple Code Blocks:

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

```
<html>
<head>
<title><?php echo "Welcome to my Web site!";?></title>
</head>
<body>
<?php
$date = "July 26, 2007";
?>
<p>Today's date is <?=$date;?></p>
</body>
</html>
```

As you can see, any variables declared in a prior code block are "remembered" for later blocks, as is the case with the $date variable in this example.

Commenting Your Code:

Single-Line C++ Syntax: Comments often require no more than a single line.

jkdirectory jkmaterials

Because of its brevity, there is no need to delimit the comment's conclusion because the newline (\n) character fills this need quite nicely. PHP supports C++ single-line comment syntax, which is prefaced with a double slash (//), like this:

```
<?php
// Title: My first PHP script
// Author: Jason
echo "This is a PHP program";
?>
```

Multiple-Line C Syntax:

It's often convenient to include somewhat more verbose functional descriptions or other explanatory notes within code, which logically warrants numerous lines. Although you could preface each line with C++ or shell-style delimiters, PHP also offers a multiple-line variant that can open and close the comment on different lines. Here's an example:

```
<?php
/*
Title: My PHP Program
Author: Jason
Date: July 26, 2007
*/
?>
```

Outputting Data to the Browser:

PHP offers several methods for doing so.

The print() Statement: It outputs data passed to it to the browser. Its prototype looks like this:

int print(argument)

All of the following are plausible print() statements:

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

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

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

All these statements produce identical output:

I love the summertime.

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

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

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

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

This code produces the following:

Lennox Lewis and Floyd Mayweather are great fighters.

The printf() statement:

If your intent is to output a blend of static text and dynamic information passed through variables or dynamic information stored within one or several variables, consider using printf() instead.

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

Its prototype looks like this: **boolean printf(string format [, mixed args])**

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

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

Executing this command produces the following:

Bar inventory: 100 bottles of tonic water.

In this example, %d is a placeholder known as a type specifier, and the d indicates an integer value will be placed in that position. When the printf() statement executes, the lone argument, 100, will be inserted into the placeholder.

Commonly Used Type Specifiers:

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

PHP's SUPPORTED DATATYPES:

A datatype is the generic name assigned to any data sharing a common set of characteristics. Common datatypes include Boolean, integer, float, string, and array. PHP has long offered a rich set of datatypes:

Scalar Datatypes: Scalar datatypes are capable of containing a single item of information. Several datatypes fall under this category, including Boolean, integer, float, and string.

jkdirectory jkmaterials

Boolean:

The Boolean datatype is named after George Boole (1815–1864), a mathematician who is considered to be one of the founding fathers of information theory. A Boolean variable represents truth, supporting only two values: TRUE and FALSE (case insensitive).

Alternatively, you can use zero to represent FALSE, and any nonzero value to represent TRUE. A few examples follow:

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

Integer:

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

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

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

Float:

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

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

String:

A string is a sequence of characters treated as a contiguous group. Strings are delimited by single or double quotes.

The following are all examples of valid strings:

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

Historically, PHP treated strings in the same fashion as arrays allowing for specific characters to be accessed via array offset notation. For example, consider the following string:

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

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

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

Compound Datatypes:

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

Array:

It's often useful to aggregate a series of similar items together, arranging and referencing them in some specific way. This data structure, known as an array, is formally defined as an indexed collection of data values. Each member of the array index also known as the ***key*** references a corresponding value and can be a simple numerical reference to the value's position in the series, or it could have some direct correlation to the value.

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

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

If the project required correlating U.S. states to their capitals then we might instead use an associative index, like this:

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

Object:

The other compound datatype supported by PHP is the object. The object is a central concept of the object-oriented programming paradigm.

Unlike the other datatypes contained in the PHP language, an object must be explicitly declared. This declaration of an object's characteristics and behavior takes place within something called a class. Here's a general example of a class definition and subsequent invocation:

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

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

jkdirectory jkmaterials

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

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

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

Converting Between Datatypes Using Type Casting:

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

Cast Operators	Conversion
(array)	Array
(bool) or (boolean)	Boolean
(int) or (integer)	Integer
(int64)	64-bit integer (introduced in PHP6)
(object)	Object
(real) or (double) or (float)	Float
(string)	String

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

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

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

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

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

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

The value can then be referenced as follows: print $ obj->**scalar**; // returns "Toyota"

Adapting Datatypes with Type Juggling:

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

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

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

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

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

Another interesting example: If a string used in a mathematical calculation includes ., e, or E (representing scientific notation), it will be evaluated as a float:

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

Type-Related Functions:

A few functions are available for both verifying and converting datatypes; they are:

Retrieving Types:

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

Converting Types:

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

Type Identifier Functions:

A number of functions are available for determining a variable's type, including is_*array*(), is_*bool*(), is_*float*(), is_*integer*(), is_*null*(), is_*numeric*(), is_*object*(), is_*resource*(), is_*scalar*(), and is_*string*(). Because all of these functions follow the same naming convention, arguments, and return values, their introduction is consolidated into a single example. The generalized prototype follows: **boolean is_*name*(mixed var)**

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

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

This code returns the following:

The variable $item is of type array: 0

The variable $item is of type integer: 1

The variable $item is numeric: 1

You might be wondering about the backslash preceding $item. Given the dollar sign's special purpose of identifying a variable, there must be a way to tell the interpreter to treat it as a normal character should you want to output it to the screen. Delimiting the dollar sign with a backslash will accomplish this.

Identifiers:

Identifier is a general term applied to variables, functions, and various other user defined objects. There are several properties that PHP identifiers must abide by:

1. An identifier can consist of one or more characters and must begin with a letter or an underscore. Furthermore, identifiers can consist of only letters, numbers; underscore characters, and other ASCII characters from 127 through 255.

   Valid identifiers are: my_function, Size, _someword

   Invalid identifiers are: This&that, !counter, 4ward

2. Identifiers are case sensitive. Therefore, a variable named $recipe is different from a variable named $Recipe, $rEciPe, or $recipE.

3. Identifiers can be any length.

4. An identifier name can't be identical to any of PHP's predefined keywords.

VARIABLES:

A variable is a symbol that can store different values at different times. For example, suppose you create a Web-based calculator capable of performing mathematical tasks. Of course, the user will want to plug in values of his choosing; therefore, the program must be able to dynamically store those values and perform calculations accordingly. At the same time, the programmer requires a user-friendly means for referring to these value-holders within the application. The variable accomplishes both tasks.

A variable is a named memory location that contains data and may be manipulated throughout the execution of the program.

Variable Declaration:

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

- *$color*
- *$operating_system*
- *$_some_variable*
- *$model*

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

- *$color*
- *$Color*

jkdirectory jkmaterials

- *$COLOR*

Interestingly, variables do not have to be explicitly declared in PHP as they do in Perl. Rather, variables can be declared and assigned values simultaneously. Good programming practice dictates that all variables should be declared prior to use, preferably with an accompanying comment.

Once you've declared your variables, you can begin assigning values to them. Two methodologies are available for variable assignment: *by value* and *by reference*.

Value Assignment:

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

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

Reference Assignment:

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

Let's consider an example:

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

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

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

Variable Scope:

However you declare your variables (by value or by reference), you can declare them anywhere in a PHP script. The location of the declaration greatly influences the realm in which a variable can be accessed, however. This accessibility domain is known as its scope.

PHP variables can be one of four scope types:

- **Local Variables:**

A variable declared in a function is considered local. That is, it can be referenced only in that function. Any assignment outside of that function will be considered to be an entirely different variable from the one contained in the function. Note that when you exit

the function in which a local variable has been declared, that variable and its corresponding value are destroyed.

Local variables are helpful because they eliminate the possibility of unexpected side effects, which can result from globally accessible variables that are modified, intentionally or not.

Example:

```
$x = 4;
function assignx () {
$x = 0;
printf("\$x inside function is %d <br />", $x);
}
assignx();
printf("\$x outside of function is %d <br />", $x);
```

Executing this listing results in the following:

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

- **Function Parameters:**

As in many other programming languages, in PHP, any function that accepts arguments must declare those arguments in the function header. Although those arguments accept values that come from outside of the function, they are no longer accessible once the function has exited.

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

```
// multiply a value by 10 and return it to the caller
function x10 ($value) {
$value = $value * 10;
return $value;
}
```

Keep in mind that although you can access and manipulate any function parameter in the function in which it is declared, it is destroyed when the function execution ends.

- **Global Variables:**

In contrast to local variables, a global variable can be accessed in any part of the program. To modify a global variable, however, it must be explicitly declared to be global in the function in which it is to be modified. This is accomplished, conveniently enough, by placing the keyword GLOBAL in front of the variable that should be recognized as global.

Placing this keyword in front of an already existing variable tells PHP to use the variable having that name. Consider an example:

```
$somevar = 15;
function addit() {
GLOBAL $somevar;
$somevar++;
echo "Somevar is $somevar";
}
```

```
addit();
```

The displayed value of $somevar would be 16. However, if you were to omit this line:

```
GLOBAL $somevar;
```

The variable $somevar would be assigned the value 1 because $somevar would then be considered local within the addit() function. This local declaration would be implicitly set to 0 and then incremented by 1 to display the value 1.

An alternative method for declaring a variable to be global is to use PHP's $GLOBALS array. Reconsidering the preceding example, you can use this array to declare the variable $somevar to be global:

```
$somevar = 15;
function addit() {
$GLOBALS["somevar"]++;
}
addit();
echo "Somevar is ".$GLOBALS["somevar"];
```

This returns the following:

Somevar is 16

- **Static Variables:**

In contrast to the variables declared as function parameters, which are destroyed on the function's exit, a static variable does not lose its value when the function exits and will still hold that value if the function is called again. You can declare a variable as static simply by placing the keyword STATIC in front of the variable name: **STATIC $somevar;**

Example:

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

As the variable $count was designated to be static the outcome would be as follows:

1
2
3

If the variable $count was not designated to be static then the outcome would be as follows:

1
2
3

Static scoping is particularly useful for recursive functions, which is a powerful programming concept in which a function repeatedly calls itself until a particular condition is met.

PHP's Superglobal Variables:

PHP offers a number of useful predefined variables that are accessible from anywhere within the executing script and provide you with a substantial amount of environment-specific information. You can sift through these variables to retrieve details about the current user session; the user's operating environment, the local operating environment, and more.

PHP creates some of the variables, while the availability and value of many of the other variables are specific to the operating system and Web server.

Therefore, rather than attempt to assemble a comprehensive list of all possible predefined variables and their possible values, the following code will output all predefined variables pertinent to any given Web server and the script's execution environment:

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

As we can get a bit of information where something is useful and something not so useful, we can display just one of these variables simply by treating it as a regular variable. For example, use this to display the user's IP address:

```
printf("Your IP address is: %s", $_SERVER['REMOTE_ADDR']);
```

This returns a numerical IP address, such as 192.0.34.166.

You can also gain information regarding the user's browser and operating system. Consider the following one-liner:

```
printf("Your browser is: %s", $_SERVER['HTTP_USER_AGENT']);
```

This returns information similar to the following:

Your browser is: Mozilla/5.0 (Windows; U; Windows NT 6.0; en-US; rv:1.8.0.10)Gecko/20070216 Firefox/1.5.0.10

Retrieving Variables Passed Using GET:

The $_GET Superglobal contains information pertinent to any parameters passed using GET method. If the URL http://www.example.com/index.html?cat=apache&id=157 is requested, you could access the following variables by using the $_GET Superglobal:

```
$_GET['cat'] = "apache"
$_GET['id'] = "157"
```

The $_GET Superglobal by default is the only way that you can access variables passed via the GET method. You cannot reference GET variables like this: $cat, $id.

Retrieving Variables Passed Using POST:

The $_POST Superglobal contains information pertinent to any parameters passed using the POST method. Consider the following form, used to solicit subscriber information:

```
<form action="subscribe.php" method="post">
<p>
Email address:<br />
```

```
<input type="text" name="email" size="20" maxlength="50" value="" />
</p>
<p>
Password:<br />
<input type="password" name="pswd" size="20" maxlength="15" value="" />
</p>
<p>
<input type="submit" name="subscribe" value="subscribe!" />
</p>
</form>
```

The following POST variables will be made available via the target subscribe.php script:

```
$_POST['email'] = "jason@example.com";
$_POST['pswd'] = "rainyday";
$_POST['subscribe'] = "subscribe!";
```

Like $_GET, the $_POST Superglobal is by default the only way to access POST variables. You cannot reference POST variables like this: $email, $pswd, and $subscribe.

Retrieving Information Stored Within Cookies:

The $_COOKIE Superglobal stores information passed into the script through HTTP cookies. Such cookies are typically set by a previously executed PHP script through the PHP function setcookie(). For example, suppose that you use setcookie() to store a cookie named example.com with the value ab2213. You could later retrieve that value by calling $_COOKIE["example.com"].

Retrieving Information About Files Uploaded Using POST:

The $_FILES Superglobal contains information regarding data uploaded to the server via the POST method. This Superglobal is a tad different from the others in that it is a two-dimensional array containing five elements. The first subscript refers to the name of the form's file-upload form element; the second is one of five predefined subscripts that describe a particular attribute of the uploaded file:

$_FILES['upload-name']['name']: The name of the file as uploaded from the client to the server.

$_FILES['upload-name']['type']: The MIME type of the uploaded file. Whether this variable is assigned depends on the browser capabilities.

$_FILES['upload-name']['size']: The byte size of the uploaded file.

$_FILES['upload-name']['tmp_name']: Once uploaded, the file will be assigned a temporary name before it is moved to its final location.

$_FILES['upload-name']['error']: An upload status code. Despite the name, this variable will be populated even in the case of success. There are five possible values:

UPLOAD_ERR_OK: The file was successfully uploaded.

UPLOAD_ERR_INI_SIZE: The file size exceeds the maximum size imposed by the upload_max_filesize directive.

UPLOAD_ERR_FORM_SIZE: The file size exceeds the maximum size imposed by an optional MAX_FILE_SIZE hidden form-field parameter.

UPLOAD_ERR_PARTIAL: The file was only partially uploaded.

UPLOAD_ERR_NO_FILE: A file was not specified in the upload form prompt.

jkmaterials jkdirectory

Learning More about the Operating System Environment:

The $_ENV Superglobal offers information regarding the PHP parser's underlying server environment. Some of the variables found in this array include the following:

$_ENV['HOSTNAME']: The server hostname

$_ENV['SHELL']: The system shell

Variable Variables:

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

$recipe = "spaghetti";

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

$$recipe = "& meatballs";

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

echo $recipe $spaghetti;

echo $recipe ${$recipe};

The result of both is the string spaghetti & meatballs.

CONSTANTS:

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

Defining a Constant:

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

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

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

define("PI", 3.141592);

The constant is subsequently used in the following listing:

printf("The value of pi is %f", PI);

*$pi2 = 2 * PI;*

printf("Pi doubled equals %f", $pi2);

This code produces the following results:

The value of pi is 3.141592.

Pi doubled equals 6.283184.

jkdirectory jkmaterials

EXPRESSIONS:

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

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

Operands:

Operands are the inputs of an expression. Some examples of operands follow:

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

OPERATORS:

An *operator* is a symbol that specifies a particular action in an expression. The precedence and associativity of operators are significant characteristics of a programming language.

Operator precedence, associativity and purpose is given in the following table:

<table>
<tr><th>Operator</th><th>Associativity</th><th>Purpose</th></tr>
<tr><td>New</td><td>NA</td><td>Object instantiation</td></tr>
<tr><td>( )</td><td>NA</td><td>Expression sub grouping</td></tr>
<tr><td>[ ]</td><td>Right</td><td>Index closure</td></tr>
<tr><td>! ~ ++ --</td><td>Right</td><td>Boolean NOT, bitwise NOT, increment, decrement</td></tr>
<tr><td>@`</td><td>Right</td><td>Suppression</td></tr>
<tr><td>/ * %</td><td>Left</td><td>Division, multiplication, modulus</td></tr>
<tr><td>+ - .</td><td>Left</td><td>Addition, subtraction, concatenation</td></tr>
<tr><td><< >></td><td>Left</td><td>Shift left, shift right (bitwise)</td></tr>
<tr><td>< <= > >=</td><td>NA</td><td>Less than, less than or equal to, greater than, greater than or equal to</td></tr>
<tr><td>== != === <></td><td>NA</td><td>Is equal to, is not equal to, is identical to, is not equal to</td></tr>
<tr><td>& ^ |</td><td>Left</td><td>Bitwise AND, bitwise XOR, bitwise OR</td></tr>
<tr><td>&& ||</td><td>Left</td><td>Boolean AND, Boolean OR</td></tr>
<tr><td>? :</td><td>Right</td><td>Ternary operator</td></tr>
<tr><td>= += *= /= .= %=&= |= ^= <<= >>=</td><td>Right</td><td>Assignment operators</td></tr>
<tr><td>AND XOR OR</td><td>Left</td><td>Boolean AND, Boolean XOR, Boolean OR</td></tr>
<tr><td>,</td><td>Left</td><td>Expression separation</td></tr>
</table>

Operator Precedence:

Operator precedence is a characteristic of operators that determines the order in which they evaluate the operands surrounding them.

Examples:

$total_cost = $cost + $cost * 0.06;

This is the same as writing *$total_cost = $cost + ($cost * 0.06);* because the multiplication operator has higher precedence than the addition operator.

Operator Associativity:

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

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

The preceding example is the same as the following: *$value = ((((3 * 4) * 5) * 7) * 2);*

This expression results in the value 840 because the multiplication (*) operator is left-to-right associative.

In contrast, right-to-left associativity evaluates operators of the same precedence from right to left:

```
$c = 5;
print $value = $a = $b = $c;
```

The preceding example is the same as the following:

```
$c = 5;
$value = ($a = ($b = $c));
```

When this expression is evaluated, variables $value, $a, $b, and $c will all contain the value 5 because the assignment operator (=) has right-to-left associativity.

Arithmetic Operators:

The arithmetic operators, listed in table below, perform various mathematical operations and will probably be used frequently in many of your PHP programs.

Example	Label	Outcome
$a + $b	Addition	Sum of $a and $b
$a - $b	Subtraction	Difference of $a and $b
$a * $b	Multiplication	Product of $a and $b
$a / $b	Division	Quotient of $a and $b
$a % $b	Modulus	Remainder of $a divided by $b

Assignment Operators:

The assignment operators assign a data value to a variable. The simplest form of assignment operator just assigns some value, while others (known as shortcut assignment operators) perform some other operation before making the assignment.

Example		Label		Outcome
$a = 5	→	Assignment	→	$a equals 5
$a += 5	→	Addition-assignment	→	$a equals $a plus 5
$a *= 5	→	Multiplication-assignment	→	$a equals $a multiplied by 5
$a /= 5	→	Division-assignment	→	$a equals $a divided by 5
$a .= 5	→	Concatenation-assignment	→	$a equals $a concatenated with 5

String Operators:

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

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

Here is an example involving string operators:

$a = "Spaghetti" . "& Meatballs";

// $a contains the string value "Spaghetti & Meatballs";

$a .= " are delicious."

// $a contains the value "Spaghetti & Meatballs are delicious."

Increment and Decrement Operators:

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

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

These operators can be placed on either side of a variable, and the side on which they are placed provides a slightly different effect. Consider the outcomes of the following examples:

$inv = 15; // Assign integer value 15 to $inv.
$oldInv = $inv--; // Assign $oldInv the value of $inv, then decrement $inv.
$origInv = ++$inv; // Increment $inv, then assign the new $inv value to $origInv

Logical Operators:

It provides a way to make decisions based on the values of multiple variables. Logical operators make it possible to direct the flow of a program and are used frequently with control structures, such as the if conditional and the while and for loops.

Logical operators are also commonly used to provide details about the outcome of other operations, particularly those that return a value:

file_exists("filename.txt") OR echo "File does not exist!";

One of two outcomes will occur:

The file filename.txt exists
The sentence "File does not exist!" will be output

Example	Label	Outcome
$a && $b	AND	True if both $a and $b are true
$a AND $b	AND	True if both $a and $b are true
$a || $b	OR	True if either $a or $b is true
$a OR $b	OR	True if either $a or $b is true
!$a	NOT	True if $a is not true
NOT $a	NOT	True if $a is not true
$a XOR $b	Exclusive OR	True if only $a or only $b is true

Equality Operators:

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

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

Comparison Operators:

Comparison operators like logical operators provide a method to direct program flow through an examination of comparative values of two or more variables.

Example	Label	Outcome
$a < $b	Less than	True if $a is less than $b
$a > $b	Greater than	True if $a is greater than $b
$a <= $b	Less than or equal to	True if $a is less than or equal to $b
$a >= $b	Greater than or equal to	True if $a is greater than or equal to $b
($a == 12) ? 5 **:** -1	Ternary	If $a equals 12, return value is 5; Otherwise, return value is –1

Bitwise Operators:

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

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

String Interpolation:

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

The $animal jumped over the wall.\n

You might assume that $animal is a variable and that \n is a newline character, and therefore both should be interpreted accordingly. However if you want to output the string exactly as it is written or perhaps you want the new line to be rendered but want the variable to display in it's literal form i.e. *$animal*, or vice versa, all these variations are possible in PHP.

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

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

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

jkmaterials jkdirectory

Escape sequences are also parsed. Consider this example:

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

This returns the following within the browser source:

This is one line.
And this is another line.

In addition to the newline character, PHP recognizes a number of special escape sequences, all of which are listed in table below:

Sequence	Description
\n	Newline character
\r	Carriage return
\t	Horizontal tab
\\	Backslash
\$	Dollar sign
\”	Double quote
\[0-7]{1,3}	Octal notation
\x[0-9A-Fa-f]{1,2}	Hexadecimal notation

Single Quotes: Enclosing a string within single quotes is useful when the string should be interpreted exactly as stated. This means that both variables and escape sequences will not be interpreted when the string is parsed. For example, consider the following single quoted string:

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

This produces the following:

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

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

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

?>
```

Several points are worth noting regarding this example:

1. The opening and closing identifiers, in the case of this example, EXCERPT, must be identical. You can choose any identifier as you like, but they must exactly match.

2. The only constraint is that the identifier must consist of solely alphanumeric characters and underscores and must not begin with a digit or an underscore.

3. The opening identifier must be preceded with three left-angle brackets, <<<.

4. Heredoc syntax follows the same parsing rules as strings enclosed in double quotes. That is, both variables and escape sequences are parsed. The only difference is that double quotes do not need to be escaped.
5. The closing identifier must begin at the very beginning of a line. It cannot be preceded with spaces or any other extraneous character.

Heredoc syntax is particularly useful when you need to manipulate a substantial amount of material but do not want to put up with the hassle of escaping quotes.

CONTROL STRUCTURES:

Control structures determine the flow of code within an application, defining execution characteristics such as whether and how many times a particular code statement will execute, as well as when a code block will relinquish execution control. These structures also offer a simple means to introduce entirely new sections of code (via file-inclusion statements) into a currently executing script.

Conditional Statements:

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

The if Statement: It is one of the most commonplace constructs of any mainstream programming language, offering a convenient means for conditional code execution. The following is the **syntax:**

```
if (expression) {
statement
}
```

Example:

```
<?php
$secretNumber = 453;
if ($_POST['guess'] == $secretNumber) {
echo "<p>Congratulations!</p>";
}
?>
```

The If statement brackets can be skipped if only a single code statement is enclosed

The else Statement: The problem with the previous example is that output is only offered for the user who correctly guesses the secret number. if you want to provide a tailored response no matter the outcome is then we have to use the function handily offered by way of the else statement.

Example:

```
<?php
$secretNumber = 453;
if ($_POST['guess'] == $secretNumber) {
echo "<p>Congratulations!!</p>";
} else {
echo "<p>Sorry!</p>";
}
?>
```

Like if, the else statement brackets can be skipped if only a single code statement is enclosed.

The elseif Statement:

The if-else combination works nicely in an “either-or” situation—that is, a situation in which only two possible outcomes are available. But what if several outcomes are possible? You would need a means for considering each possible outcome, which is accomplished with the elseif statement.

Example:

```
<?php
$secretNumber = 453;
$_POST['guess'] = 442;
if ($_POST['guess'] == $secretNumber) {
echo "<p>Congratulations!</p>";
} elseif (abs ($_POST['guess'] - $secretNumber) < 10) {
echo "<p>You're getting close!</p>";
} else {
echo "<p>Sorry!</p>";
}
?>
```

Like all conditionals, elseif supports the elimination of bracketing when only a single statement is enclosed.

The switch Statement:

You can think of the switch statement as a variant of the if-else combination, often used when you need to compare a variable against a large number of values:

Example:

```
<?php
switch($category) {
case "news":
echo "<p>What's happening around the world</p>";
break;
case "weather":
echo "<p>Your weekly forecast</p>";
break;
case "sports":
echo "<p>Latest sports highlights</p>";
break;
default:
echo "<p>Welcome to my Web site</p>";
}
?>
```

Note the presence of the break statement at the conclusion of each case block. If a break statement isn’t present, all subsequent case blocks will execute until a break statement is located.

Looping Statements:

Looping mechanisms offer a simple means for accomplishing a commonplace task in programming: repeating a sequence of instructions until a specific condition is satisfied. PHP offers several such mechanisms:

The while Statement:

The while statement specifies a condition that must be met before execution of its embedded code is terminated. Its **syntax** is the following:

```
while (expression) {
statements
}
```

In the following **example**, $count is initialized to the value 1. The value of $count is then squared and output. The $count variable is then incremented by 1, and the loop is repeated until the value of $count reaches 5.

```
<?php
$count = 1;
while ($count < 5) {
printf("%d squared = %d <br />", $count, pow($count, 2));
$count++;
}
?>
```

The output looks like this:

1 squared = 1
2 squared = 4
3 squared = 9
4 squared = 16

The do...while Statement:

It is a variant of while but it verifies the loop conditional at the conclusion of the block rather than at the beginning. The following is its **syntax**:

```
do {
statements
} while (expression);
```

Both while and do...while are similar in function. The only real difference is that the code embedded within a while statement possibly could never be executed, whereas the code embedded within a do...while statement will always execute at least once. Consider the following example:

```
<?php
$count = 11;
do {
printf("%d squared = %d <br />", $count, pow($count, 2));
} while ($count < 10);
?>
```

The following is the outcome:

11 squared = 121

jkdirectory jkmaterials

The for Statement:

The for statement offers a somewhat more complex looping mechanism than does while. The following is its **syntax**:

```
for (expression1; expression2; expression3) {
statements
}
```

There are a few rules to keep in mind when using PHP’s for loops:

1. The first expression, expression1, is evaluated by default at the first iteration of the loop.
2. The second expression, expression2, is evaluated at the beginning of each iteration; this expression determines whether looping will continue.
3. The third expression, expression3, is evaluated at the conclusion of each loop.
4. Any of the expressions can be empty, their purpose substituted by logic embedded within the for block.

Example-1:

```
for ($kilometers = 1; $kilometers <= 5; $kilometers++)
{
	printf("%d kilometers = %f miles <br />", $kilometers, $kilometers*0.62140);
}
```

Example-2:

```
for ($kilometers = 1; ; $kilometers++)
{
	if ($kilometers > 5) break;
		printf("%d kilometers = %f miles <br />", $kilometers, $kilometers*0.62140);
}
```

Example-3:

```
$kilometers = 1;
for (;;)
{
// if $kilometers > 5 break out of the for loop.
if ($kilometers > 5) break;
printf("%d kilometers = %f miles <br />", $kilometers, $kilometers*0.62140);
$kilometers++;
}
```

The results of all the three examples are as follows:

1 kilometers = 0.6214 miles

2 kilometers = 1.2428 miles

3 kilometers = 1.8642 miles

4 kilometers = 2.4856 miles

5 kilometers = 3.107 miles

jkdirectory jkmaterials

The foreach Statement:

The foreach looping construct syntax is adept at looping through arrays, pulling each key/value pair from the array until all items have been retrieved or some other internal conditional has been met. Two syntax variations are available.

The first syntax variant strips each value from the array, moving the pointer closer to the end with each iteration. The following is its **syntax**:

```
foreach (array_expr as $value)
{
statement
}
```

Example: Suppose you want to output an array of links:

```
<?php
$links = array("www.apress.com","www.php.net","www.apache.org");
echo "<b>Online Resources</b>:<br />";
foreach($links as $link) {
echo "<a href=\"http://$link\">$link</a><br />";
}
?>
```

This would result in the following:

```
Online Resources:<br />
<a href="http://www.apress.com">http://www.apress.com</a><br />
<a href="http://www.php.net">http://www.php.net</a><br />
<a href="http://www.apache.org"> http://www.apache.org </a><br />
```

The second variation is well-suited for working with both the key and value of an array. The **syntax** follows:

```
foreach (array_expr as $key => $value) {
statement
}
```

Revising the previous example, suppose that the $links array contains both a link and a corresponding link title:

```
$links = array("The Apache Web Server" => "www.apache.org", "Apress" => "www.apress.com", "The PHP Scripting Language" => "www.php.net");
```

Each array item consists of both a key and a corresponding value. The foreach statement can easily peel each key/value pair from the array, like this:

```
echo "<b>Online Resources</b>:<br />";
foreach($links as $title => $link) {
echo "<a href=\"http://$link\">$title</a><br />";
}
```

The result would be that each link is embedded under its respective title, like this:

```
Online Resources:<br />
<a href="http://www.apache.org">The Apache Web Server </a><br />
<a href="http://www.apress.com"> Apress </a><br />
<a href="http://www.php.net">The PHP Scripting Language </a><br />
```

The break and goto Statements

Encountering a break statement will immediately end execution of a do...while, for, foreach, switch, or while block. For example, the following for loop will terminate if a prime number is pseudo-randomly happened upon:

```
<?php
$primes = array(2,3,5,7,11,13,17,19,23,29,31,37,41,43,47);
for($count = 1; $count++; $count < 1000) {
$randomNumber = rand(1,50);
if (in_array($randomNumber,$primes)) {
break;
} else {
printf("Non-prime number found: %d <br />", $randomNumber);
}
}
?>
```

Sample output follows:

Non-prime number found: 48
Non-prime number found: 42
Prime number found: 17

Through the addition of the goto statement, this feature was extended in PHP 6 to support labels. This means you can suddenly jump to a specific location outside of a looping or conditional construct.

An example follows:

```
<?php
for ($count = 0; $count < 10; $count++)
{
$randomNumber = rand(1,50);
if ($randomNumber < 10)
goto less;
else
echo "Number greater than 10: $randomNumber<br />";
}
less:
echo "Number less than 10: $randomNumber<br />";
?>
```

It produces the following (your output will vary):

Number greater than 10: 22
Number greater than 10: 21
Number greater than 10: 35
Number less than 10: 8

The continue Statement

The continue statement causes execution of the current loop iteration to end and commence at the beginning of the next iteration.

Example:

```
<?php
$usernames = array("grace","doris","gary","nate","missing","tom");
for ($x=0; $x < count($usernames); $x++) {
if ($usernames[$x] == "missing") continue;
printf("Staff member: %s <br />", $usernames[$x]);
}
?>
```

This results in the following output:

Staff member: grace
Staff member: doris
Staff member: gary
Staff member: nate
Staff member: tom

File-Inclusion Statements:

Efficient programmers are always thinking in terms of ensuring reusability and modularity. The most prevalent means for ensuring such is by isolating functional components into separate files and then reassembling those files as needed. PHP offers four statements for including such files into applications:

The include() Statement

This statement will evaluate and include a file into the location where it is called. Including a file produces the same result as copying the data from the file specified into the location in which the statement appears. Its prototype follows:

include(/path/to/filename)

Like the print and echo statements, you have the option of omitting the parentheses when using include().

Example:

```
<?php
include "/usr/local/lib/php/wjgilmore/init.inc.php";
/* the script continues here */
?>
```

Note: You can also execute include() statements conditionally.

Example:

```
<?php
    if (expression)
        include ('filename');
?>
```

Ensuring a File Is Included Only Once:

The include_once() function has the same purpose as include() except that it first verifies whether the file has already been included. Its prototype follows:

include_once (filename)

If a file has already been included, include_once() will not execute.

Requiring a File:

For the most part, require() operates like include(), including a template into the file in which the require() call is located. Its prototype follows:

require (filename)

However, there are two important differences between require() and include(). First, the file will be included in the script in which the require() construct appears, regardless of where require() is located. For instance, if require() is placed within an if statement that evaluates to false, the file would be included anyway. The second important difference is that script execution will stop if a require() fails, whereas it may continue in the case of an include().

STRINGS:

PHP offers more than 100 functions collectively capable of manipulating practically every imaginable aspect of a string. The following are the some of the topics:

1. Determining string length
2. Comparing two strings
3. Manipulating string case
4. Converting strings to and from HTML
5. Counting characters and words

Determining string length:

Determining string length is a repeated action within countless applications. The PHP function strlen() accomplishes this task quite nicely. This function returns the length of a string, where each character in the string is equivalent to one unit. Its prototype follows:

int strlen(string str)

The following example verifies whether a user password is of acceptable length:

```
<?php
$pswd = "secretpswd";
if (strlen($pswd) < 10)
echo "Password is too short!";
else
echo "Password is valid!";
?>
```

In this case, the error message will not appear because the chosen password consists of ten characters, whereas the conditional expression validates whether the target string consists of less than ten characters.

Comparing Two Strings:

String comparison is arguably one of the most important features of the string-handling capabilities of any language. Although there are many ways in which two strings can be compared for equality, PHP provides four functions for performing this task: strcmp(), strcasecmp(), strspn(), and strcspn().

Comparing Two Strings Case Sensitively:

The strcmp() function performs a binary-safe, case-sensitive comparison of two strings. Its prototype follows:

int strcmp(string str1, string str2)

It will return one of three possible values based on the comparison outcome:

- ☑ 0 if str1 and str2 are equal
- ☑ -1 if str1 is less than str2
- ☑ 1 if str2 is less than str1

Web sites often require a registering user to enter and then confirm a password, lessening the possibility of an incorrectly entered password as a result of a typing error. strcmp() is a great function for comparing the two password entries because passwords are often case sensitive:

```
<?php
$pswd = "supersecret";
$pswd2 = "supersecret2";
if (strcmp($pswd,$pswd2) != 0)
echo "Passwords do not match!";
else
echo "Passwords match!";
?>
```

Comparing Two Strings Case Insensitively:

The strcasecmp() function operates exactly like strcmp(), except that its comparison is case insensitive. Its prototype follows:

int strcasecmp(string str1, string str2)

The following example compares two e-mail addresses, an ideal use for strcasecmp() because case does not determine an e-mail addresses uniqueness:

```
<?php
$email1 = "admin@example.com";
$email2 = "ADMIN@example.com";
if (! strcasecmp($email1, $email2))
echo "The email addresses are identical!";
?>
```

Calculating the Similarity between Two Strings:

The strspn() function returns the length of the first segment in a string containing characters also found in another string. Its prototype follows:

int strspn(string str1, string str2)

Here's how you might use strspn() to ensure that a password does not consist solely of numbers:

```
<?php
$password = "3312345";
if (strspn($password, "1234567890") == strlen($password))
echo "The password cannot consist solely of numbers!";
?>
```

In this case, the error message is returned because $password does indeed consist solely of digits.

jkdirectory jkmaterials

Calculating the Difference between Two Strings:

The strcspn() function returns the length of the first segment of a string containing characters not found in another string. Its prototype follows:

int strcspn(string str1, string str2)

Here's an example of password validation using strcspn():

```
<?php
$password = "a12345";
if (strcspn($password, "1234567890") == 0) {
echo "Password cannot consist solely of numbers!";
}
?>
```

In this case, the error message will not be displayed because $password does not consist solely of numbers.

Manipulating String Case

Four functions are available to aid you in manipulating the case of characters in a string: strtolower(), strtoupper(), ucfirst(), and ucwords().

Converting a String to All Lowercase:

The strtolower() function converts a string to all lowercase letters, returning the modified string. Nonalphabetical characters are not affected. Its prototype follows:

string strtolower(string str)

The following example uses strtolower() to convert a URL to all lowercase letters:

```
<?php
$url = "http://WWW.EXAMPLE.COM/";
echo strtolower($url);
?>
```

This returns the following: **http://www.example.com/**

Converting a String to All Uppercase:

This is accomplished with the function strtoupper(). Its prototype follows:

string strtoupper(string str)

Nonalphabetical characters are not affected. This example uses strtoupper() to convert a string to all uppercase letters:

```
<?php
$msg = "I annoy people by capitalizing e-mail text.";
echo strtoupper($msg);
?>
```

This returns the following:

I ANNOY PEOPLE BY CAPITALIZING E-MAIL TEXT.

Capitalizing the First Letter of a String:

The ucfirst() function capitalizes the first letter of the string str, if it is alphabetical. Its prototype follows:

jkdirectory jkmaterials

string ucfirst(string str)

Nonalphabetical characters will not be affected. Additionally, any capitalized characters found in the string will be left untouched. Consider this example:

```
<?php
$sentence = "the newest version of PHP was released today!";
echo ucfirst($sentence);
?>
```

This returns the following:

The newest version of PHP was released today!

Note that while the first letter is indeed capitalized, the capitalized word PHP was left untouched.

Capitalizing Each Word in a String:

The ucwords() function capitalizes the first letter of each word in a string. Its prototype follows:

string ucwords(string str)

Nonalphabetical characters are not affected. This example uses ucwords() to capitalize each word in a string:

```
<?php
$title = "O'Malley wins the heavyweight championship!";
echo ucwords($title);
?>
```

This returns the following:

O'Malley Wins The Heavyweight Championship!

Converting Strings to and from HTML

Converting a string or an entire file into a form suitable for viewing on the Web and vice versa) is easier than you would think. Several functions are suited for such tasks:

Converting Newline Characters to HTML Break Tags

The nl2br() function converts all newline (\n) characters in a string to their XHTML compliant equivalent, <br />. Its prototype follows:

string nl2br(string str)

The newline characters could be created via a carriage return, or explicitly written into the string. The following example translates a text string to HTML format:

```
<?php
$recipe = "3 tablespoons Dijon mustard
1/3 cup Caesar salad dressing
8 ounces grilled chicken breast
3 cups romaine lettuce";
// convert the newlines to <br />'s.
echo nl2br($recipe);
?>
```

Executing this example results in the following output:

3 tablespoons Dijon mustard

1/3 cup Caesar salad dressing

8 ounces grilled chicken breast

3 cups romaine lettuce

Converting Special Characters to Their HTML Equivalents:

During the general course of communication, you may come across many characters that are not included in a document's text encoding, or that are not readily available on the keyboard.

Examples of such characters include the copyright symbol (©), the cent sign (¢), and the grave accent (è). To facilitate such shortcomings, a set of universal key codes was devised, known as *character entity references.* When these entities are parsed by the browser, they will be converted into their recognizable counterparts.

For example, the three aforementioned characters would be presented as ©, ¢, and È, respectively. To perform these conversions, you can use the htmlentities() function. Its prototype follows:

string htmlentities(string str [, int quote_style [, int charset]])

Because of the special nature of quote marks within markup, the optional quote_style parameter offers the opportunity to choose how they will be handled. Three values are accepted:

ENT_COMPAT: Convert double quotes and ignore single quotes. *(This is default)*.

ENT_NOQUOTES: Ignore both double and single quotes.

ENT_QUOTES: Convert both double and single quotes.

A second optional parameter, charset, determines the character set used for the conversion. Table given below offers the list of supported character sets. If charset is omitted, it will default to ISO-8859-1.

CHARACTER SET	DESCRIPTION
BIG5	Traditional Chinese
BIG5-HKSCS	BIG5 with additional Hong Kong extensions, traditional Chinese
cp866	DOS-specific Cyrillic character set
cp1251	Windows-specific Cyrillic character set
cp1252	Windows-specific character set for Western Europe
EUC-JP	Japanese
GB2312	Simplified Chinese
ISO-8859-1	Western European, Latin-1
ISO-8859-15	Western European, Latin-9
KOI8-R	Russian
Shift-JIS	Japanese
UTF-8	ASCII-compatible multibyte 8 encode

The following example converts the necessary characters for Web display:

```
<?php
$advertisement = "Coffee at 'Cafè Française' costs $2.25.";
echo htmlentities($advertisement);
?>
```

This returns the following:

```
Coffee at 'Caf&egrave; Fran&ccedil;aise' costs $2.25.
```

Two characters are converted, the grave accent (è) and the cedilla (ç). The single quotes are ignored due to the default quote_style setting ENT_COMPAT.

Using Special HTML Characters for Other Purposes:

Several characters play a dual role in both markup languages and the human language. When used in the latter fashion, these characters must be converted into their displayable equivalents.

For example, an ampersand must be converted to &, whereas a greater-than character must be converted to >. The htmlspecialchars() function can do this for you, converting the following characters into their compatible equivalents. Its prototype follows:

string htmlspecialchars(string str [, int quote_style [, string charset]])

The list of characters that htmlspecialchars() can convert and their resulting formats follow:

- ☑ & becomes &
- ☑ " (double quote) becomes "
- ☑ ' (single quote) becomes '
- ☑ < becomes <
- ☑ > becomes >

This function is particularly useful in preventing users from entering HTML markup into an interactive Web application, such as a message board.

The following example converts potentially harmful characters using htmlspecialchars():

```
<?php
$input = "I just can't get <<enough>> of PHP!";
echo htmlspecialchars($input);
?>
```

Viewing the source, you’ll see the following:

I just can't get <<enough>> of PHP &!

If the translation isn’t necessary, perhaps a more efficient way to do this would be to use strip_tags(), which deletes the tags from the string altogether.

Converting Text into Its HTML Equivalent:

Using get_html_translation_table() is a convenient way to translate text to its HTML equivalent, returning one of the two translation tables (HTML_SPECIALCHARS or HTML_ENTITIES). Its prototype follows:

array get_html_translation_table(int table [, int quote_style])

This returned value can then be used in conjunction with another predefined function, strtr()to essentially translate the text into its corresponding HTML code. The following sample uses get_html_translation_table() to convert text to HTML:

```
<?php
$string = "La pasta é il piatto piú amato in Italia";
$translate = get_html_translation_table(HTML_ENTITIES);
echo strtr($string, $translate);       ?>
```

This returns the string formatted as necessary for browser rendering:

La pasta é il piatto piú amato in Italia

jkdirectory jkmaterials

Interestingly, array_flip() is capable of reversing the text-to-HTML translation and vice versa. Assume that instead of printing the result of strtr() in the preceding code sample, you assign it to the variable $translated_string. The next example uses array_flip() to return a string back to its original value:

```
<?php
$entities = get_html_translation_table(HTML_ENTITIES);
$translate = array_flip($entities);
$string = "La pasta &eacute; il piatto pi&uacute; amato in Italia";
echo strtr($string, $translate);
?>
```

This returns the following:

La pasta é il piatto piú amato in italia

Converting HTML to Plain Text:

You may sometimes need to convert an HTML file to plain text. You can do so using the strip_tags() function, which removes all HTML and PHP tags from a string, leaving only the text entities. Its prototype follows:

string strip_tags(string str [, string allowable_tags])

The optional allowable_tags parameter allows you to specify which tags you would like to be skipped during this process. This example uses strip_tags() to delete all HTML tags from a string:

```
<?php
$input = "Email <a href='spammer@example.com'>spammer@example.com</a>";
echo strip_tags($input);
?>
```

This returns the following:

Email spammer@example.com

The following sample strips all tags except the <a> tag:

```
<?php
$input = "This <a href='http://www.example.com/'>example</a>
is <b>awesome</b>!";
echo strip_tags($input, "<a>");
?>
```

This returns the following:

This example is awesome!

Counting Characters and Words

It's often useful to determine the total number of characters or words in a given string.

Counting the Number of Characters in a String

The function count_chars() offers information regarding the characters found in a string. Its prototype follows:

mixed count_chars(string str [, mode])

Its behavior depends on how the optional parameter mode is defined:

0: Returns an array consisting of each found byte value as the key and the corresponding frequency as the value, even if the frequency is zero. This is the default.

1: Same as 0, but returns only those byte values with a frequency greater than zero.

2: Same as 0, but returns only those byte values with a frequency of zero.

3: Returns a string containing all located byte values.

4: Returns a string containing all unused byte values.

The following example counts the frequency of each character in $sentence:

```
<?php
$sentence = "The rain in Spain falls mainly on the plain";
// Retrieve located characters and their corresponding frequency.
$chart = count_chars($sentence, 1);
foreach($chart as $letter=>$frequency)
echo "Character ".chr($letter)." appears $frequency times<br />";
?>
```

This returns the following:

Character appears 8 times
Character S appears 1 times
Character T appears 1 times
Character a appears 5 times
Character e appears 2 times
Character f appears 1 times
Character h appears 2 times
Character i appears 5 times
Character l appears 4 times
Character m appears 1 times
Character n appears 6 times
Character o appears 1 times
Character p appears 2 times
Character r appears 1 times
Character s appears 1 times
Character t appears 1 times
Character y appears 1 times

Counting the Total Number of Words in a String

The function str_word_count() offers information regarding the total number of words found in a string. Its prototype follows:

mixed str_word_count(string str [, int format])

If the optional parameter format is not defined, it will simply return the total number of words. If format is defined, it modifies the function's behavior based on its value:

1. Returns an array consisting of all words located in str.
2. Returns an associative array, where the key is the numerical position of the word in str, and the value is the word itself.

Example:

```
<?php
$summary = <<< summary
In the latest installment of the ongoing Developer.com PHP series,
I discuss the many improvements and additions to PHP 5's
object-oriented architecture.
summary;
$words = str_word_count($summary);
printf("Total words in summary: %s", $words);
?>
```

This returns the following:

Total words in summary: 23

You can use this function in conjunction with array_count_values() to determine the frequency in which each word appears within the string:

```
<?php
$summary = <<< summary
In the latest installment of the ongoing Developer.com PHP series,
I discuss the many improvements and additions to PHP 5's
object-oriented architecture.
summary;
$words = str_word_count($summary,2);
$frequency = array_count_values($words);
print_r($frequency);
?>
```

This returns the following:

Array ([In] => 1 [the] => 3 [latest] => 1 [installment] => 1 [of] => 1 [ongoing] => 1 [Developer] => 1 [com] => 1 [PHP] => 2 [series] => 1 [I] => 1 [discuss] => 1 [many] => 1 [improvements] => 1 [and] => 1 [additions] => 1 [to] => 1 [s] => 1 [object-oriented] => 1 [architecture] => 1)

ARRAYS:

An *array* is traditionally defined as a group of items that share certain characteristics, such as similarity and type. Each item is distinguished by a special identifier known as a *key.* Each item consists of two components: the aforementioned key and a value. The key serves as the lookup facility for retrieving its counterpart, the *value*. Keys can be *numerical* or *associative*.

Numerical keys bear no real relation to the value other than the value's position in the array. As an example, the array could consist of an alphabetically sorted list of state names, with key 0 representing Alabama, and key 49 representing Wyoming. Using PHP syntax, this might look like the following:

```
$states = array(0 => "Alabama", "1" => "Alaska"..."49" => "Wyoming");
```

Using numerical indexing, you could reference the first state (Alabama) as $states[0].

Like many programming languages, PHP's numerically indexed arrays begin with position 0, not 1.

An associative key logically bears a direct relation to its corresponding value.

For instance, you might want to create an array that maps state abbreviations to their names, like this: OH/Ohio, PA/Pennsylvania, and NY/New York. Using PHP syntax, this might look like the following:

$states = array("OH" => "Ohio", "PA" => "Pennsylvania", "NY" => "New York")

You could then reference Ohio as $states["OH"].

MULTIDIMENSIONAL ARRAYS:

It's also possible to create arrays of arrays, known as *multidimensional arrays.* For example, you could use a multidimensional array to store U.S. state information. Using PHP syntax, it might look like this:

$states = array ("Ohio" => array("population" => "11,353,140", "capital" => "Columbus"), "Nebraska" => array("population" => "1,711,263", "capital" => "Omaha"));

You could then reference Ohio's population:

$states["Ohio"]["population"]

This would return the following:

11,353,140

CREATING AN ARRAY:

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

Individual elements of a PHP array are referenced by denoting the element between a pair of square brackets. Because there is no size limitation on the array, you can create the array simply by making reference to it, like this:

$state[0] = "Delaware";

You can then display the first element of the array $state like this:

echo $state[0];

Additional values can be added by mapping each new value to an array index, like this:

$state[1] = "Pennsylvania";

$state[2] = "New Jersey"; and so on.

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

$state[] = "Pennsylvania";

$state[] = "New Jersey";

Creating associative arrays in this fashion is equally trivial except that the key is always required.

Creating Arrays with array()

The array() construct takes as its input zero or more items and returns an array consisting of these input elements.

Its prototype looks like this:

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

jkdirectory.materials

Here is an example of using array() to create an indexed array:

```
$languages = array("English", "Gaelic", "Spanish");
// $languages[0] = "English", $languages[1] = "Gaelic", $languages[2] = "Spanish"
```

You can also use array() to create an associative array, like this:

```
$languages = array("Spain" => "Spanish",
"Ireland" => "Gaelic",
"United States" => "English");
// $languages["Spain"] = "Spanish"
// $languages["Ireland"] = "Gaelic"
// $languages["United States"] = "English"
```

Extracting Arrays with list():

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

void list(mixed...)

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

For example, suppose you wanted to format and output information read from a text file named users.txt. Each line of the file contains user information, including name, occupation, and favorite color with each item delimited by a vertical bar. A typical line would look similar to the following:

Nino Sanzi|professional golfer|green

Using list(), a simple loop could read each line, assign each piece of data to a variable, and format and display the data as needed.

```
// Open the users.txt file
$users = fopen("users.txt", "r");
// While the EOF hasn't been reached, get next line
while ($line = fgets($users, 4096)) {
// use explode() to separate each piece of data.
list($name, $occupation, $color) = explode("|", $line);

// format and output the data
printf("Name: %s <br />", $name);
printf("Occupation: %s <br />", $occupation);
printf("Favorite color: %s <br />", $color);
}
fclose($users);
```

Each line of the users.txt file will be read and formatted similarly to this:

```
Name: Nino Sanzi
Occupation: professional golfer
Favorite Color: green
```

jkdirectory jkmaterials

Populating Arrays with a Predefined Value Range:

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

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

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

```
$die = range(0,6);
// Same as specifying $die = array(0,1,2,3,4,5,6)
```

If you want a range consisting of solely even or odd values or a range consisting of values solely divisible by five, the optional step parameter offers a convenient means for doing so.

<u>Example</u>:

If you want to create an array consisting of all even values between 0 and 20, you could use a step value of 2:

```
$even = range(0,20,2);
// $even = array(0,2,4,6,8,10,12,14,16,18,20);
```

The range() function can also be used for character sequences. For example, suppose you want to create an array consisting of the letters A through F:

```
$letters = range("A","F");
// $letters = array("A,","B","C","D","E","F");
```

Testing for an Array:

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

boolean is_array(mixed *variable*)

The is_array() function determines whether variable is an array, returning TRUE if it is and FALSE otherwise. Note that even an array consisting of a single value will still be considered an array. An example follows:

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

Executing this example produces the following:

```
$states is an array: TRUE
$state is an array: FALSE
```

Adding and Removing Array Elements:

PHP provides a number of functions for both growing and shrinking an array. Some of these functions are provided as a convenience to programmers who wish to mimic various queue implementations.

1. **Adding a Value to the Front of an Array:**

The array_unshift() function adds elements onto the front of the array.

All preexisting numerical keys are modified to reflect their new position in the array, but associative keys aren't affected. Its prototype follows:

```
int array_unshift(array array, mixed variable [, mixed variable...])
```

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

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

2. Adding a Value onto the End of an Array:

The array_push() function adds a value onto the end of an array, returning TRUE on success and FALSE otherwise. You can push multiple variables onto the array simultaneously by passing these variables into the function as input parameters. Its prototype follows:

int array_push(array *array*, mixed *variable* [, mixed *variable*...])

The following example adds two more states onto the $states array:

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

3. Removing a Value from the Front of an Array:

The array_shift() function removes and returns the item found in an array. Resultingly, if numerical keys are used, all corresponding values will be shifted down, whereas arrays using associative keys will not be affected. Its prototype follows:

mixed array_shift(array *array*)

The following example removes the first state from the $states array:

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

4. Removing a Value from the End of an Array:

The array_pop() function removes and returns the last element from an array. Its prototype follows:

mixed array_pop(array *target_array*)

The following example removes the last state from the $states array:

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

LOCATING ARRAY ELEMENTS:

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

Searching an Array: The in_array() function searches an array for a specific value, returning TRUE if the value is found, and FALSE otherwise.

jkdirectory Materials

Its prototype follows:

boolean in_array(mixed *needle*, array *haystack* [, boolean *strict*])

In the following example, a message is output if a specified state (Ohio) is found in an array consisting of states having statewide smoking bans:

```
$state = "Ohio";
$states = array("California", "Hawaii", "Ohio", "New York");
if(in_array($state, $states)) echo "Not to worry, $state is smoke-free!";
```

The optional third parameter, strict, forces in_array() to also consider type.

Searching Associative Array Keys: The function array_key_exists() returns TRUE if a specified key is found in an array, and returns FALSE otherwise. Its prototype follows:

boolean array_key_exists(mixed *key*, array *array*)

The following example will search an array's keys for Ohio, and if found, will output information about its entrance into the Union:

```
$state["Delaware"] = "December 7, 1787";
$state["Pennsylvania"] = "December 12, 1787";
$state["Ohio"] = "March 1, 1803";
if (array_key_exists("Ohio", $state))
printf("Ohio joined the Union on %s", $state["Ohio"]);
```

The following is the result:

```
Ohio joined the Union on March 1, 1803
```

Searching Associative Array Values: The array_search() function searches an array for a specified value, returning its key if located, and FALSE otherwise. Its prototype follows:

mixed array_search(mixed *needle*, array *haystack* [, boolean *strict*])

The following example searches $state for a particular date (December 7), returning information about the corresponding state if located:

```
$state["Ohio"] = "March 1";
$state["Delaware"] = "December 7";
$state["Pennsylvania"] = "December 12";
$founded = array_search("December 7", $state);
if ($founded) printf("%s was founded on %s.", $founded, $state[$founded]);
```

The output follows:

```
Delaware was founded on December 7.
```

Retrieving Array Keys: The array_keys() function returns an array consisting of all keys located in an array. Its prototype follows:

array array_keys(array *array* [, mixed *search_value*])

If the optional search_value parameter is included, only keys matching that value will be returned. The following example outputs all of the key values found in the $state array:

```
$state["Delaware"] = "December 7, 1787";
$state["Pennsylvania"] = "December 12, 1787";
$state["New Jersey"] = "December 18, 1787";
```

jkdirectory jkmaterials

```
$keys = array_keys($state);
print_r($keys);
```

The output follows:

```
Array ( [0] => Delaware [1] => Pennsylvania [2] => New Jersey )
```

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

array array_values(array *array*)

The following example will retrieve the population numbers for all of the states found in $population:

```
$population = array("Ohio" => "11,421,267", "Iowa" => "2,936,760");
print_r(array_values($population));
```

This example will output the following:

```
Array ( [0] => 11,421,267 [1] => 2,936,760 )
```

TRAVERSING ARRAYS:

The need to travel across an array and retrieve various keys, values, or both is common, that PHP offers numerous functions, many of these functions do double duty: retrieving the key or value residing at the current pointer location, and moving the pointer to the next appropriate location.

Retrieving the Current Array Key:

The key() function returns the key located at the current pointer position of input_array. Its prototype follows:

mixed key(array array)

The following example will output the $capitals array keys by iterating over the array and moving the pointer:

```
$capitals = array("Ohio" => "Columbus", "Iowa" => "Des Moines");
echo "<p>Can you name the capitals of these states?</p>";
while($key = key($capitals)) {
printf("%s <br />", $key);
next($capitals);
}
```

This returns the following:

```
Can You name the capitals of these states?
Ohio
Iowa
```

Retrieving the Current Array Value:

The current() function returns the array value residing at the current pointer position of the array. Its prototype follows:

mixed current(array array)

The following example will output the $capitals array values by iterating over the array and moving the pointer:

```
$capitals = array("Ohio" => "Columbus", "Iowa" => "Des Moines");
echo "<p>Can you name the states belonging to these capitals?</p>";
while($capital = current($capitals)) {
printf("%s <br />", $capital);
next($capitals);
}
```

The output follows:

Can you name the states belonging to these capitals?

Columbus
Des Moines

Retrieving the Current Array Key and Value:

The each() function returns the current key/value pair from the array and advances the pointer one position. Its prototype follows:

array each(array array)

The returned array consists of four keys, with keys 0 and key containing the key name, and keys 1 and value containing the corresponding data. If the pointer is residing at the end of the array before executing each(), FALSE is returned.

Moving the Array Pointer:

Several functions are available for moving the array pointer, some of them are as follows:

Moving the Pointer to the Next Array Position:

The next() function returns the array value residing at the position immediately following that of the current array pointer. Its prototype follows:

mixed next(array *array*)

<u>**Example**</u>**:**

```
$fruits = array("apple", "orange", "banana");
$fruit = next($fruits); // returns "orange"
$fruit = next($fruits); // returns "banana"
```

Moving the Pointer to the Previous Array Position:

The prev() function returns the array value residing at the location preceding the current pointer location, or FALSE if the pointer resides at the first position in the array. Its prototype follows:

mixed prev(array array)

Moving the Pointer to the First Array Position:

The reset() function serves to set an array pointer back to the beginning of the array. Its prototype follows:

mixed reset(array array)

This function is commonly used when you need to review or manipulate an array multiple times within a script, or when sorting has completed.

Moving the Pointer to the Last Array Position:

The end() function moves the pointer to the last position of an array, returning the last element. Its prototype follows:

jkdirectorysjkmaterials

```
mixed end(array array)
```

The following example demonstrates retrieving the first and last array values:

```
$fruits = array("apple", "orange", "banana");
$fruit = current($fruits); // returns "apple"
$fruit = end($fruits); // returns "banana"
```

PASSING ARRAY VALUES TO A FUNCTION:

The array_walk() function will pass each element of an array to the user-defined function. This is useful when you need to perform a particular action based on each array element. If you intend to actually modify the array key/value pairs, you'll need to pass each key/value to the function as a reference. Its prototype follows:

boolean array_walk(array &*array*, callback *function* [, mixed *userdata*])

The user-defined function must take two parameters as input. The first represents the array's current value, and the second represents the current key. If the optional userdata parameter is present in the call to array_walk(), its value will be passed as a third parameter to the user-defined function.

<u>Example</u>: Using an Array in a Form

```
<form action="submitdata.php" method="post">
<p> Provide up to six keywords that you believe best describe the state in which you live: </p>
<p>Keyword 1:<br />
<input type="text" name="keyword[]" size="20" maxlength="20" value="" /></p>
<p>Keyword 2:<br />
<input type="text" name="keyword[]" size="20" maxlength="20" value="" /></p>
<p>Keyword 3:<br />
<input type="text" name="keyword[]" size="20" maxlength="20" value="" /></p>
<p>Keyword 4:<br />
<input type="text" name="keyword[]" size="20" maxlength="20" value="" /></p>
<p>Keyword 5:<br />
<input type="text" name="keyword[]" size="20" maxlength="20" value="" /></p>
<p>Keyword 6:<br />
<input type="text" name="keyword[]" size="20" maxlength="20" value="" /></p>
<p><input type="submit" value="Submit!"></p>
</form>
```

This form information is then sent to some script, referred to as submitdata.php in the form. This script should sanitize user data and then insert it into a database for later review. Using array_walk(), you can easily filter the keywords using a predefined function:

```
<?php
function sanitize_data(&$value, $key) {
$value = strip_tags($value);
}
array_walk($_POST['keyword'],"sanitize_data");
?>
```

The result is that each value in the array is run through the strip_tags() function, which results in any HTML and PHP tags being deleted from the value.

Determining the Size of an Array:

The count() function returns the total number of values found in an array. Its prototype follows:

integer count(array array [, int mode])

If the optional mode parameter is enabled (set to 1), the array will be recursively counted which is a feature useful when counting all elements of a multidimensional array. The first example counts the total number of vegetables found in the $garden array:

```
$garden = array("cabbage", "peppers", "turnips", "carrots");
echo count($garden);
```

This returns the following:

4

The next example counts both the scalar values and array values found in $locations:

```
$locations = array("Italy","Amsterdam",array("Boston","Des Moines"),"Miami");
echo count($locations,1);
```

This returns the following:

6

Counting Array Value Frequency:

The array_count_values() function returns an array consisting of associative key/value pairs. Its prototype follows:

array array_count_values(array array)

Each key represents a value found in the input_array, and its corresponding value denotes the frequency of that key's appearance (as a value) in the input_array.

Example:

```
$states = array("Ohio","Iowa","Arizona","Iowa","Ohio");
$stateFrequency = array_count_values($states);
print_r($stateFrequency);
```

This returns the following:

```
Array ( [Ohio] => 2 [Iowa] => 2 [Arizona] => 1 )
```

Determining Unique Array Values:

The array_unique() function removes all duplicate values found in an array, returning an array consisting of solely unique values. Its prototype follows:

array array_unique(array array)

Example:

```
$states = array("Ohio","Iowa","Arizona","Iowa","Ohio");
$uniqueStates = array_unique($states);
print_r($uniqueStates);
```

This returns the following:

```
Array ( [0] => Ohio [1] => Iowa [2] => Arizona )
```

jkdirectory jkmaterials

SORTING ARRAYS:

Reversing Array Element Order:

The array_reverse() function reverses an array's element order. Its prototype follows:

array array_reverse(array array [, boolean preserve_keys])

If the optional preserve_keys parameter is set to TRUE, the key mappings are maintained. Otherwise, each newly rearranged value will assume the key of the value previously presiding at that position:

```
$states = array("Delaware","Pennsylvania","New Jersey");
print_r(array_reverse($states));
// Array ( [0] => New Jersey [1] => Pennsylvania [2] => Delaware )
```

Contrast this behavior with that resulting from enabling preserve_keys:

```
$states = array("Delaware","Pennsylvania","New Jersey");
print_r(array_reverse($states,1));
// Array ( [2] => New Jersey [1] => Pennsylvania [0] => Delaware )
```

Arrays with associative keys are not affected by preserve_keys; key mappings are always preserved in this case.

Flipping Array Keys and Values:

The array_flip() function reverses the roles of the keys and their corresponding values in an array. Its prototype follows:

array array_flip(array array)

An example follows:

```
$state = array("Delaware","Pennsylvania","New Jersey");
$state = array_flip($state);
print_r($state);
```

This example returns the following:

```
Array ( [Delaware] => 0 [Pennsylvania] => 1 [New Jersey] => 2 )
```

Sorting an Array

The sort() function sorts an array, ordering elements from lowest to highest value. Its prototype follows:

void sort(array array [, int sort_flags])

The sort() function doesn't return the sorted array. Instead, it sorts the array "in place," returning nothing, regardless of outcome. The optional sort_flags parameter modifies the function's default behavior in accordance with its assigned value:

SORT_NUMERIC: Sorts items numerically. This is useful when sorting integers or floats.

SORT_REGULAR: Sorts items by their ASCII value.

SORT_STRING: Sorts items in a fashion that might better correspond with how a human might perceive the correct order.

Example:

```
$grades = array(42,98,100,100,43,12);
```

jkdirectory jkmaterials

```
sort($grades);
print_r($grades);
```

The outcome looks like this:

```
Array ( [0] => 12 [1] => 42 [2] => 43 [3] => 98 [4] => 100 [5] => 100 )
```

It's important to note that key/value associations are not maintained.

For Example:

```
$states = array("OH" => "Ohio", "CA" => "California", "MD" => "Maryland");
sort($states);
print_r($states);
```

Here's the output:

```
Array ( [0] => California [1] => Maryland [2] => Ohio )
```

Sorting an Array in Reverse Order:

The rsort() function is identical to sort(), except that it sorts array items in reverse (descending) order. Its prototype follows:

```
void rsort(array array [, int sort_flags])
```

An example follows:

```
$states = array("Ohio","Florida","Massachusetts","Montana");
rsort($states);
print_r($states);
```

It returns the following:

```
Array ( [0] => Ohio [1] => Montana [2] => Massachusetts [3] => Florida )
```

FUNCTIONS:

The concept of embodying the repetitive processes within a named section of code and then invoking this name when necessary is known as a function.

Invoking Function:

You can invoke the function you want simply by specifying the function name, assuming that the function has been made available either through the library's compilation into the installed distribution or via the include() or require() statement.

For example:

Suppose you want to raise five to the third power. You could invoke PHP's pow() function like this:

```
<?php
$value = pow(5,3); // returns 125
echo $value;
?>
```

Creating a Function:

Although PHP's vast assortment of function libraries is a tremendous benefit to anybody seeking to avoid reinventing the programmatic wheel, sooner or later you'll need to go beyond what is offered in the standard distribution, which means you'll need to create custom functions or even entire function libraries.

jkdirectory jkmaterials

To do so, you'll need to define a function using a predefined template, like so:

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

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

```
function generateFooter()
{
echo "Copyright 2007 W. Jason Gilmore";
}
```

Once defined, you can call this function like so:

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

This yields the following result:

Copyright 2007 W. Jason Gilmore

Passing Arguments by Value:

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

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

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

Once defined, you can then invoke the function as demonstrated in the previous section. For example, the calcSalesTax() function would be called like so:

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

Of course, you're not bound to passing static values into the function. You can also pass variables like this:

```
<?php
$pricetag = 15.00;
$salestax = .075;
calcSalesTax($pricetag, $salestax);
?>
```

When you pass an argument in this manner, it's called passing by value.

jkdirectory jkmaterials

This means that any changes made to those values within the scope of the function are ignored outside of the function.

Passing Arguments by Reference:

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

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

Here's the result:

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

Note the value of $tax remains the same, although $cost has changed.

Default Argument Values:

Default values can be assigned to input arguments, which will be automatically assigned to the argument if no other value is provided. To revise the sales tax example, suppose that the majority of your sales are to take place in Franklin County, Ohio. You could then assign $tax the default value of 6.75 percent, like this:

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

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

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

Default argument values must appear at the end of the parameter list and must be constant expressions; you cannot assign nonconstant values such as function calls or variables.

You can designate certain arguments as optional by placing them at the end of the list and assigning them a default value of nothing, like so:

jkdirectory jkmaterials

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

This allows you to call calcSalesTax() without the second parameter if there is no sales tax:

```
calcSalesTax(42.00);
```

This returns the following output:

```
Total cost: $42.00
```

If multiple optional arguments are specified, you can selectively choose which ones are passed along. Consider this example:

```
function calculate($price, $price2="", $price3="")
{
echo $price + $price2 + $price3;
}
```

You can then call calculate(), passing along just $price and $price3, like so:

```
calculate(10, "", 3);
```

This returns the following value:

```
13
```

Returning Values from a Function:

Often, simply relying on a function to do something is insufficient; a script's outcome might depend on a function's outcome, or on changes in data resulting from its execution.

Yet variable scoping prevents information from easily being passed from a function body back to its caller, we can accomplish this by passing data back to the caller by the way of the *return()* statement.

The return Statement:

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

Revising the calcSalestax() function again, suppose you don't want to immediately echo the sales total back to the user upon calculation, but rather want to return the value to the calling block:

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

Alternatively, you could return the calculation directly without even assigning it to $total, like this:

```
function calcSalesTax($price, $tax=.0675)
{
```

```
return $price + ($price * $tax);
}
```

Here’s an example of how you would call this function:

```
<?php
$price = 6.99;
$total = calcSalesTax($price);
?>
```

Returning Multiple Values

It’s often convenient to return multiple values from a function. For example, suppose that you’d like to create a function that retrieves user data from a database, say the user’s name, e-mail address, and phone number, and returns it to the caller.

Accomplishing this is much easier than you might think, with the help of a very useful language construct, list(). The list() construct offers a convenient means for retrieving values from an array, like so:

```
<?php
$colors = array("red","blue","green");
list($red, $blue, $green) = $colors;
?>
```

Once the list() construct executes, $red, $blue, and $green will be assigned red, blue, and green, respectively. Building on the concept demonstrated in the previous example, you can imagine how the three prerequisite values might be returned from a function using list():

```
<?php
function retrieveUserProfile()
{
$user[] = "Jason";
$user[] = "jason@example.com";
$user[] = "English";
return $user;
}
list($name, $email, $language) = retrieveUserProfile();
echo "Name: $name, email: $email, language: $language";
?>
```

Executing this script returns the following:

```
Name: Jason, email: jason@example.com, language: English
```

Recursive Functions:

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

Function Libraries:

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

jkdirectory jkmaterials

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

Save this library, preferably using a naming convention that will clearly denote its purpose, such as taxes.library.php. Do not however save this file within the server document root using an extension that would because the Web server to pass the file contents unparsed. You can insert this file into scripts using include(), include_once(), require(), or require_once().

For example, assuming that you titled this library taxation.library.php, you could include it into a script like this:

```
<?php
require_once("taxation.library.php");
...
?>
```

jkdirectory jkmaterials