Apress™
Books for Professionals by Professionals

Chapter Two: "Variables and Data Types"

A Programmer's Introduction to PHP 4.0

by William Jason Gilmore
ISBN # 1-893115-85-2

CHAPTER 2

Variables and Data Types

Data types form the backbone of any programming language, providing the programmer with a means by which to represent various types of information. PHP provides support for six general data types:

- Integers

- Floating-point numbers

- Strings

- Arrays

- Objects

- Booleans

One of the pillars of any programming language is its support for numbers. PHP supports both integers and real (double) numbers. Each of these number formats is described in further detail later.

Integer Values

An *integer* is nothing more than a whole number. Integers are represented as a series of one or more digits. Some examples of integers are:

5
591
52

Octals and Hexadecimals

Integers in octal (base 8) and hexadecimal (base 16) formats are supported. Octal values begin with the digit 0, followed by a sequence of digits, each ranging from 0 to 7. Some examples of octal integers are:

```
0422
0534
```

Hexadecimal integers can consist of the digits 0 through 9 or the letters a (A) through f (F). All hexadecimal integers are preceded by 0x or 0X. Some examples of hexadecimal integers are:

```
0x3FF
0X22abc
```

Floating-Point Numbers

A *floating-point number* is essentially a real numbers, that is, a number denoted either wholly or in part by a fraction. Floating-point numbers are useful for representing values that call for a more accurate representation, such as temperature or monetary figures. PHP supports two floating-point formats: standard notation and scientific notation.

Standard Notation

Standard notation is a convenient representation typically used for real numbers, for example, monetary values. Some examples are:

```
12.45
98.6
```

Scientific Notation

Scientific notation is a more convenient representation for very large and very small numbers, such as interplanetary distances or atomic measurements. Some examples include:

```
3e8
5.9736e24
```

String Values

A *string* is a group of characters that are represented as a single entity but can also be examined on a character-by-character basis. Some examples of strings are:

```
thesaurus
49ers
abc
&%/$£
```

Note that PHP doesn't include support for the char data type. Rather, the string data type can be considered the all-encompassing type that represents both single and multiple character sets.

String Assignments

Strings can be delimited in two ways, using either double quotation marks ("") or single quotation marks (''). There are two fundamental differences between the two methods. First, variables in a double-quoted string will be replaced with their respective values, whereas the single-quoted strings will be interpreted exactly as is, even if variables are enclosed in the string.

The following two string declarations produce the same result:

```
$food = "meatloaf";
$food = 'meatloaf';
```

However, the following two declarations result in two drastically different outcomes:

```
$sentence = "My favorite food is $food";
$sentence2 = 'My favorite food is $food';
```

The following string is what exactly will be assigned to $sentence. Notice how the variable $food is automatically interpreted:

```
My favorite food is meatloaf.
```

Whereas $sentence2 will be assigned the string as follows:

```
My favorite food is $food.
```

In contrast with $sentence, the uninterpreted variable $food will appear in the string assigned to $sentence2. These differing outcomes are due to the usage of double and single quotation marks in assigning the corresponding strings to $sentence and $sentence2.

Before discussing the second fundamental difference between double-quoted and single-quoted strings, an introduction of PHP's supported string delimiters is in order. As with most other mainstream languages, a set of delimiters is used to represent special characters, such as the tab or newline characters. Table 2-1 lists the supported delimiters:

Table 2-1. Supported String Delimiters

CHARACTER	SEQUENCE	REPRESENTATION
	\n	Newline
	\r	Carriage return
	\t	Horizontal tab
	\\	Backslash
	\$	Dollar sign
	\"	Double-quotation mark
	\[0-7]{1,3}	Octal notation regular expression pattern
	\x[0-9A-Fa-f]{1,2}	Hexadecimal notation regular expression pattern

With this in mind, the second fundamental difference is that while a double-quoted string recognizes all available delimiters, a single-quoted string recognizes only the delimiters "\\" and "\". Consider an example of the contrasting outcomes when assigning strings enclosed in double and single quotation marks:

```
$double_list = "item1\nitem2\nitem3";
$single_list = 'item1\nitem2\nitem3';
```

If you print both strings to the browser, the double-quoted string will conceal the newline character, but the single-quoted string will print it just as if it were any other character. Although many of the delimited characters will be irrelevant in the browser, this will prove to be a major factor when formatting for various other media. Keep this difference in mind when using double- or single-quoted enclosures so as to ensure the intended outcome.

Here Doc Syntax

A second method with which to delimit strings, introduced in PHP4, is known as *Here doc syntax*. This syntax consists of beginning a string with <<< followed

immediately by some identifier of your choice, then the string in which you would like to assign the variable, and finally a second occurrence of the same chosen identifier. Consider this example:

```
$paragraph = <<<DELIM
This is a string that
Will be interpreted exactly
As it is written in the
variable assignment.
DELIM;
```

Be sure to choose an identifier that will not appear in the string being assigned. Furthermore, the first character of the closing identifier *must* appear in the first column of the line following the string.

Character Handling

Strings can be accessed on a character-by-character basis, much like a sequentially indexed array. (Arrays are discussed in the next section.) An example follows:

```
$sequence_number = "04efgh";
$letter = $sequence_number[4];
```

The variable $letter will hold the value g. As you will learn in the next section, PHP begins array position counts from 0. To illustrate this further, consider the fact that $sequence_number[1] would hold the value 4.

Arrays

An *array* is a list of elements each having the same type. There are two types of arrays: those that are accessed in accordance with the index position in which the element resides, and those that are associative in nature, accessed by a key value that bears some sort of association with its corresponding value. In practice, however, both are manipulated in much the same way. Arrays can also be single-dimensional or multidimensional in size.

Single-Dimension Indexed Arrays

Single-dimension indexed arrays are handled using an integer subscript to denote the position of the requested value.

The general syntax of a single-dimension array is:

```
$name[index1];
```

A single-dimension array can be created as follows:

```
$meat[0] = "chicken";
$meat[1] = "steak";
$meat[2] = "turkey";
```

If you execute this command:

```
print $meat[1];
```

The following will be output to the browser:

```
steak
```

Alternatively, arrays may be created using PHP's array() function. You can use this function to create the same $meat array as the one in the preceding example:

```
$meat = array("chicken", "steak", "turkey");
```

Executing the same print command yields the same results as in the previous example, again producing "steak".

You can also assign values to the end of the array simply by assigning values to an array variable using empty brackets. Therefore, another way to assign values to the $meat array is as follows:

```
$meat[] = "chicken";
$meat[] = "steak";
$meat[] = "turkey";
```

Single-Dimension Associative Arrays

Associative arrays are particularly convenient when it makes more sense to map an array using words rather than integers.

For example, assume that you wanted to keep track of all of the best food and wine pairings. It would be most convenient if you could simply assign the arrays using key-value pairings, for example, wine to dish. Use of an associative array to store this information would be the wise choice:

```
$pairings["zinfandel"] = "Broiled Veal Chops";
$pairings["merlot"] = "Baked Ham";
$pairings["sauvignon"] = "Prime Rib";
$pairings["sauternes"] = "Roasted Salmon";
```

Use of this associative array would greatly reduce the time and code required to display a particular value. Assume that you wanted to inform a reader of the best accompanying dish with merlot. A simple call to the pairings array would produce the necessary output:

```
print $pairings["merlot"];                 // outputs the value "Baked Ham"
```

An alternative method in which to create an array is via PHP's array() function:

```
 $pairings = array(
                    zinfandel => "Broiled Veal Chops",
                    merlot => "Baked Ham",
                    sauvignon => "Prime Rib",
                    sauternes => "Roasted Salmon";
```

This assignment method bears no difference in functionality from the previous $pairings array, other than the format in which it was created.

Multidimensional Indexed Arrays

Multidimensional indexed arrays function much like their single-dimension counterparts, except that more than one index array is used to specify an element. There is no limit as to the dimension size, although it is unlikely that anything beyond three dimensions would be used in most applications.

The general syntax of a multidimensional array is:

```
 $name[index1] [index2]..[indexN];
```

An element of a two-dimensional indexed array could be referenced as follows:

```
 $position = $chess_board[5][4];
```

Multidimensional Associative Arrays

Multidimensional associative arrays are also possible (and quite useful) in PHP. Assume you wanted to keep track of wine-food pairings, not only by wine type, but also by producer. You could do something similar to the following:

```
$pairings["Martinelli"] ["zinfandel"] = "Broiled Veal Chops";
$pairings["Beringer"] ["merlot"] = "Baked Ham";
$pairings["Jarvis"] ["sauvignon"] = "Prime Rib";
$pairings["Climens"] ["sauternes"] = "Roasted Salmon";
```

Mixing Indexed and Associative Array Indexes

It is also possible to mix indexed and associative arrays indexes. Expanding on the single-dimension associative array example, suppose you wanted to keep track of the first and second string players of the Ohio State Buckeyes football team. You could do something similar to the following:

```
$Buckeyes["quarterback"] [1] = "Bellisari";
$Buckeyes["quarterback"] [2] = "Moherman";
$Buckeyes["quarterback"] [3] = "Wiley";
```

PHP provides a vast assortment of functions for creating and manipulating arrays, so much so that the subject merits an entire chapter. Read Chapter 5, "Arrays," for a complete discussion of how PHP arrays are handled.

Objects

The fifth PHP data type is the object. You can think of an *object* as a variable that is instantiated from a kind of template otherwise known as a *class.* The concept of objects and classes is integral to the notion of object-oriented programming (OOP).

Contrary to the other data types contained in the PHP language, an object must be explicitly declared. It is important to realize that an object is nothing more than a particular instance of a class, which acts as a template for creating objects having specific characteristics and functionality. Therefore, a class must be defined before an object can be declared. A general example of class declaration and subsequent object instantiation follows:

```
class appliance {
            var power;
            function set_power($on_off) {
                            $this->power = $on_off;
            }
}
. . .
$blender = new appliance;
```

A class definition creates several characteristics and functions pertinent to a data structure, in this case a data structure named appliance. So far, the appliance isn't very functional. There is only one characteristic: power. This characteristic can be modified by using the method set_power.

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 preceding listing an object of class appliance named blender is created.

The blender power can then be set by making use of the method set_power:

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

Object-oriented programming is such an important strategy in today's application development standards that its use with PHP merits its own chapter. Chapter 6, "Object-Oriented PHP," introduces PHP's OOP implementation in further detail.

Boolean, or True/False, Values

The boolean data type is essentially capable of representing only two data types: true and false. Boolean values can be determined in two ways: as a comparison evaluation or from a variable value. Both are rather straightforward.

Comparisons can take place in many forms. Evaluation typically takes place by use of a double equal sign and an if conditional. Here is an example:

```
 if ($sum == 40) :
. . .
```

This could evaluate to only either true or false. Either $sum equals 40, or it does not. If $sum does equal 40, then the expression evaluates to true. Otherwise, the result is false.

Boolean values can also be determined via explicitly setting a variable to a true or false value. Here is an example:

```
$flag = TRUE;
 if ($flag == TRUE) :
   print "The flag is true!";
else :
   print "The flag is false!";
endif;
```

If the variable $flag has been set to true, then print the appropriate statement;
Otherwise, print an alternative statement.

An alternative way to represent true and false is by using the values 1 and 0,
respectively. Therefore, the previous example can be restated as follows:

```
$flag = 1;
 if ($flag == TRUE) :
   print "The flag is true!";
else :
   print "The flag is false !";
endif;
```

Yet another alternative way to represent the above example follows:

```
$flag = TRUE;
// this implicitly asks "if ($flag == TRUE)"
if ($flag) :
   print "The flag is true!";
else :
   print "The flag is false!";
endif;
```

Identifiers

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

- An identifier can consist of one or more characters and must begin with an
  alphabetical letter or an underscore. Furthermore, identifiers can only con-
  sist of letters, numbers, underscore characters, and other ASCII characters
  from 127 through 255. Consider a few examples:

VALID	INVALID
my_function	This&that
Size	!counter
_someword	4ward

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

- Identifiers can be any length. This is advantageous, as it enables a programmer to accurately describe the identifier's purpose via the identifier name.

- Finally, an identifier name can't be identical to any of PHP's predefined keywords.

Variables

As a byproduct of examining the examples up to this point, I've introduced you to how variables are assigned and manipulated. However, it would be wise to explicitly lay the groundwork as to how variables are declared and manipulated. The coming sections will examine these rules in detail.

Variable Declaration

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

A variable always begins with a dollar sign, $. The following are all valid variables:

```
$color
$operating_system
$_some_variable
$model
```

Variable names follow the same naming rules as those set for identifiers. That is, a variable name can begin with either an alphabetical letter or underscore and can consist of alphabetical letters, underscores, integers, or other ASCII characters ranging from 127 through 255.

Interestingly, variables do not have to be explicitly declared in PHP, much as is the case with the Perl language. Rather, variables can be declared and assigned values simultaneously. Furthermore, a variable's data type is implicitly determined by examining the kind of data that the variable holds. Consider the following example:

```
$sentence = "This is a sentence.";    // $sentence evaluates to string.
$price = 42.99;                        // $price evaluates to a floating-point
$weight = 185;                         // $weight evaluates to an integer.
```

You can declare variables anywhere in a PHP script. However, the location of the declaration greatly influences the realm in which a variable can be accessed. This access domain is known as its *scope*.

Variable Scope

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

- Local variables

- Function parameters

- Global variables

- Static variables

Local Variables

A variable declared in a function is considered *local;* that is, it can be referenced solely in that function. Any assignment outside of that function will be considered to be an entirely different variable from the one contained in the function. Note that when you exit the function in which the local variable has been declared, that variable and its corresponding value will be destroyed.

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

```
$x  = 4;

function assignx () {

            $x = 0;
            print "\$x inside function is $x. <br>";
    }

assignx();

 print "\$x outside of function is $x. <br>";
```

Execution of the above listing results in:

```
$x inside function is 0.
```

$x outside of function is 4.

As you can see, two different values for $x are output. This is because the $x located inside the assignx function is local in nature. Modification of its value has no bearing on any values located outside of the function. On the same note, modification of the $x located outside of the function has no bearing on any variables contained in assignx().

Function Parameters

As is the case with many other programming languages, in PHP any function that accepts arguments must declare these arguments in the function header. Although these 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;

}
```

It is important to realize 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. However, in order to be modified, a global variable must be explicitly declared to be global in the function in which it is to be modified. This is accomplished, conveniently enough, by placing the keyword GLOBAL in front of the variable that should be recognized as global. Placing this keyword in front of an already existing variable tells PHP to use the variable having that name. Consider an example:

```
$somevar = 15;

function addit() {
            GLOBAL $somevar;
            $somevar++;
            print "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, since $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 above example, I use this array to declare the variable $somevar to be global:

```
$somevar = 15;

function addit() {
            $GLOBALS["somevar"];
            $somevar++;
}

addit();
print "Somevar is $somevar";
```

Regardless of the method you choose to convert a variable to global scope, be aware that the global scope has long been a cause of grief among programmers due to unexpected results that may arise from their careless use. Therefore, although global variables can be extremely useful, be prudent when using them.

Static Variables

The final type of variable scoping that I discuss is known as *static.* In contrast to the variables declared as function parameters, which are destroyed on the function's exit, a static variable will not lose its value when the function exits and will still hold that value should the function be called again. You can declare a variable to be static simply by placing the keyword STATIC in front of the variable name.

```
STATIC $somevar;
```

Consider an example:

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

keep_track();
keep_track();
keep_track();
```

What would you expect the outcome of this script to be? If the variable $count were not designated to be static (thus making $count a local variable), the outcome would be:

```
1
1
1
```

However, since $count is static, it will retain its previous value each time the function is executed. Therefore, the outcome will be:

```
1
2
3
```

Static scoping is particularly useful for recursive functions. *Recursive functions* are a powerful programming concept in which a function repeatedly calls itself until a particular condition is met. Recursive functions are covered in detail in Chapter 4, "Functions."

Type Juggling

From time to time, it may be convenient to use a variable in ways that were not intended when it was first created. For example, you may wish to add the string value "15" to the integer value 12. Fortunately, PHP variable types may be modified without any explicit conversion process. This conversion process, whether explicit or implicit, is known as *type juggling* and is best illustrated with a few examples.

Consider a string and an integer value summed together. What do you think should take place? You may want different actions to take place depending on the contents of the string. For example, if an integer and a numerical string are added together, an integer will result:

```
$variable1 = 1;
$variable2 = "3";
$variable3 = $variable1 + $variable2;
//  $variable3 holds the value 4.
```

Another example of type juggling is the attempt to add an integer and a double. Most certainly the integer would be converted to a double, so as not to lose any degree of accuracy provided by the double:

```
$variable1 = 3;
$variable2 = 5.4;
$variable3 = $variable1 + $variable2;
// $variable3 converts to a double, and $variable3 is assigned 8.4
```

A few obscure features of type juggling should be brought to light. What if you attempted to add together an integer and a string *containing* an integer value, as-

suming the string was not solely numerical in nature? Consider the following example:

```
$variable1 = 5;
$variable2 = "100 bottles of beer on the wall";
$variable3 = $variable1 + $variable2;
// $variable3 holds the value 105.
```

This will result in $variable3 being set to 105. This is because the PHP parser determines the type by looking at only the initial part of a string. However, suppose we modified $variable2 to read "There are 100 bottles of beer on the wall". Since there is no easy way to convert an alphabetical character into an integer value, this string would evaluate to 0, and thus $variable3 would be assigned the value 5.

Although in most cases, PHP's type-juggling strategy will suffice and produce the intended result, it is also possible to explicitly modify a variable to a particular type. This is explained in further detail in the next section, "Type Casting."

Type Casting

Forcing a variable to behave as a type other than the one originally intended for it is a rather straightforward process known as *type casting*. Type modification can be either a one-time occurrence or permanent.

A variable can be evaluated once as a different type by casting it. 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 casts in front of the variable (see Table 2-2).

Table 2-2. Cast Operators for Variables

CAST OPERATORS	CONVERSION
(int) or (integer)	Integer
(real) or (double) or (float)	Double
(string)	String
(array)	Array
(object)	Object

A simple example of how type casting works is as follows:

```
$variable1 = 13;                  // $variable1 is assigned the integer value 13.
$variable2 = (double) $variable1;  // $variable2 is assigned the value 13.0
```

Although $variable1 originally held the integer value 13, the double cast temporarily converted the type to double (and in turn, 13 became 13.0). This value was then assigned to $variable2.

You know from the previous section that an attempt to add an integer and a double together will result in a double. This could be prevented by casting the double to be an integer, as follows:

```
$variable1 = 4.0;
$variable2 = 5;
$variable3 = (int) $variable1 + $variable2;        // $variable3 =  9
```

It is worth noting that type casting a double to an integer will always result in that double being rounded down:

```
$variable1 = 14.7;
$variable2 = (int) $variable1;                     // $variable2 = 14;
```

It is also possible to cast a string or other data type to be a member of an array. The variable being cast simply becomes the first element of the array:

```
$variable1 = 1114;
$array1 = (array) $variable1;
print $array1[0];                                  // The value 1114 is printed.
```

Finally, any data type can also 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";
$new_obj = (object) $model;
```

The value can then be referenced as:

```
 print $new_obj->scalar;
```

Variable Assignment

You've already been exposed to how values can be assigned to variables in a PHP script. However, several nuances to variable assignment are worth reviewing. You might be familiar with *assignment by value,* which simply assigns a particular value, such as the integer 1 or the string ciao, to a named variable. However, there is a second kind of variable assignment, known as *assignment by reference,* which provides a valuable service to the developer. The following sections will consider each of these assignment methods in further detail.

Assignment by Value

This is the most common type of assignment, which simply assigns a value to a memory location represented by a variable name. Some examples of assignments by value are:

```
$vehicle = "car";
$amount = 10.23;
```

These two assignments result in the memory address represented by $vehicle being assigned the string "car" and that represented by $amount receiving the value 10.23.

Assignments by value can also take place through a return call in a function:

```
 function simple () {

    return 5;
}

$return_value = simple();
```

The function simple() does nothing more than return the value 5 to the variable that called it. In this case, $return_value will be assigned the value 5.

Assignment by Reference

The other way to assign a value to a variable is by referencing another variable's memory location. Instead of copying an actual value into the destination variable, a pointer (or reference) to the memory location is assigned to the variable receiving the assignment, and therefore no actual copying takes place.

An assignment by reference is accomplished by placing an ampersand (&) in front of the source variable:

```
$dessert = "cake";
$dessert2 = &$dessert;
$dessert2 = "cookies";
print "$dessert2 <br>"; // prints cookies
print $dessert; // Again, prints cookies
```

As you can see by the previous listing, after $dessert2 has been assigned $dessert's memory location reference, any modifications to $dessert2 will result in modification of $dessert or any other variable pointing to that same memory location.

Variable Variables

On occasion it is useful to make use of variables whose contents can be treated dynamically as a variable in itself. Consider this typical variable assignment:

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

Interestingly, we can then 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:

```
print $recipe $spaghetti;
```

```
print $recipe $($recipe);
```

The result of both is the string `"spaghetti & meatballs"`.

Predefined Variables

PHP offers a number of predefined variables geared toward providing the developer with a substantial amount of internal configuration information. PHP itself creates some of the variables, while many of the other variables change depending in which operating system and Web server PHP is running.

Rather than attempt to compile a complete listing of available predefined variables, I will highlight a few of the available variables and functions that most users will find applicable and useful.

To view a comprehensive list of Web server, environment, and PHP variables offered on your particular system setup, simply execute the following code:

```
while (list($var,$value) = each ($GLOBALS)) :

                            echo "<BR>$var => $value";

endwhile;
```

This will return a list of variables similar to the following. Take a moment to peruse through the listing produced by the above code and then check out the examples that immediately follow.

```
GLOBALS =>
HTTP_GET_VARS => Array
HTTP_COOKIE_VARS => Array
HISTSIZE => 1000
HOSTNAME => server1.apress.com
LOGNAME => unstrung
HISTFILESIZE => 1000
REMOTEHOST => apress.com
MAIL => /var/spool/mail/apress
MACHTYPE => i386
TERM => vt100
HOSTTYPE => i386-linux
PATH =>
/usr/sbin:/sbin:/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:/usr/local/ja=va/bin
HOME => /root
INPUTRC => /etc/inputrc
SHELL  => /bin/csh
USER => nobody
VENDOR => intel
GROUP  => root
HOST => server1.apress.com
OSTYPE => linux
PWD => /www/bin
SHLVL => 3_ => /www/bin/httpd
DOCUMENT_ROOT  => /usr/local/apress/site.apress
HTTP_ACCEPT => */*
HTTP_ACCEPT_ENCODING => gzip, deflate
HTTP_ACCEPT_LANGUAGE => it,en-us;q=0.5
HTTP_CONNECTION => Keep-Alive
HTTP_HOST => www.apress.com
HTTP_USER_AGENT  => Mozilla/4.0 (compatible; MSIE 5.0; Windows 98;
CNETHomeBuild051099)
REMOTE_ADDR => 127.0.0.1
REMOTE_PORT => 3207
SCRIPT_FILENAME => /usr/local/apress/site.apress/j/environment_vars.php
SERVER_ADDR => 127.0.0.1
SERVER_ADMIN => admin@apress.com
SERVER_NAME => www.apress.com
SERVER_PORT  => 80
SERVER_SIGNATURE =>
Apache/1.3.12 Server at www.apress.com Port 80

SERVER_SOFTWARE => Apache/1.3.12 (Unix) PHP/4.0.1
GATEWAY_INTERFACE => CGI/1.1
```

```
SERVER_PROTOCOL => HTTP/1.1
REQUEST_METHOD => GET
QUERY_STRING =>
REQUEST_URI => /j/environment_vars.php
SCRIPT_NAME => /j/environment_vars.php
PATH_TRANSLATED => /usr/local/apress/site.apress/j/environment_vars.php
PHP_SELF  => /j/environment_vars.php
argv => Array
argc => 0
var => argc
value => argc
```

As you can see, quite a bit of information is available to you, some rather useful, some not so useful. It is possible to display just one of these variables simply by treating it as such; a variable. For example, use this to display the user's IP address:

```
 print "Hi! Your IP address is: $REMOTE_ADDR";
```

This returns a numerical IP address, such as 208.247.106.187.

It is also possible to gain information regarding the user's browser and operating system. The following one-liner:

```
print "Your browser is: $HTTP_USER_AGENT";
```

returns information similar to the following:

```
Your browser is: Mozilla/4.0 (compatible; MSIE 5.0; Windows 98;
CNETHomeBuild051099)
```

This is information regarding the browser and the operating system on which it is running. This data can prove quite useful when formatting applications to browser-specific formats.

> **NOTE** *To make use of the predefined variable arrays, track_vars must be turned on in the php.ini file. As of PHP 4.03, track_vars is always enabled.*

Constants

A *constant* is essentially a value that cannot be modified throughout the execution of a program. Constants are particularly useful when working with values that will definitely not require modification, such as pi (3.141592), or a specific distance such as the number of feet in a mile (5,280).

In PHP, constants are defined using the define() function. Once a constant has been defined, it cannot be changed (or redefined) at any other point of the program.

Pi could be defined in a PHP script as follows:

```
define("PI", "3.141592");
```

And subsequently used in the following listing:

```
print "The value of pi is". PI.".<br>";

$pi2 = 2 * PI;

print "Pi doubled equals $pi2.";
```

producing:

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

There are two points to note regarding the previous listing: The first is that use of a constant does not require a dollar sign. The second is that it's not possible to modify the constant once it has been defined (for example, 2*PI); if you need to produce a value based on the constant, the value must be stored in an alternative variable.

What's Next?

Quite a bit of material was covered in this chapter, which introduced many of the details you need to begin understanding and writing the most basic PHP programs. In particular, the following topics were discussed:

- Valid data types (integers, floating points, strings, arrays, objects, booleans)

- Identifiers

Chapter 2

- Variables (declaration, scope)

- Type juggling

- Type casting

- Variable assignment (value, reference)

- Constants

This material will serve as the foundation for creating more complicated scripts in the next chapter, which covers PHP's expressions, operators, and control structures in detail. At the conclusion of Chapter 3, you will possess enough knowledge to build your first useful PHP application; namely, a simple Web-based events calendar.