CHAPTER 3

Handling Strings and Arrays

Two data types merit some special attention—strings and arrays. We've already seen strings at work, including single- and double-quoted strings (recall also that double-quoted strings allow variable interpolation). PHP also comes packed with more string power, and we're going to dig into that in this chapter—tons of functions are built into PHP that work with strings, from sorting strings to searching them, trimming extra spaces off of them, and getting their lengths. We'll get a handle on those functions in this chapter.

Besides strings, we're also going to get a handle on *arrays* in this chapter. We've seen how to store data in simple variables, but there's more to the story here. Arrays can hold multiple data items, assigning each one a numeric or text *index* (also called a *key*). For example, if you want to store some student test scores, you can store them in an array, and then you can access each score in the array via a numeric index. That's great as far as computers are concerned because you can work through all the elements in an array simply by steadily incrementing that index, as you might do with a loop. In that way, you can use your computer to iterate over all the elements in an array in order to print them out or find their average value, for example.

Arrays represent the first time we're associating data items together. Up to this point, we've only worked with simple variables, but working with arrays is fundamental to PHP for such tasks as reading the data that users enter in web pages. We'll get the details on strings and arrays in this chapter, and I'll start with the string functions.

Listing of String Functions

PHP has plenty of built-in string functions. Table 3-1 lists a selection of them.

TABLE 3-1 The String Functions

Function	Purpose
`chr`	Returns a specific character, given its ASCII code
`chunk_split`	Splits a string into smaller chunks
`crypt`	Supports one-way string encryption (hashing)
`echo`	Displays one or more strings
`explode`	Splits a string on a substring
`html_entity_decode`	Converts all HTML entities to their applicable characters
`htmlentities`	Converts all applicable characters to HTML entities
`htmlspecialchars`	Converts special characters to HTML entities
`implode`	Joins array elements with a string
`ltrim`	Strips whitespace from the beginning of a string
`number_format`	Formats a number with grouped thousand separators
`ord`	Returns the ASCII value of character
`parse_str`	Parses the string into variables
`print`	Displays a string
`printf`	Displays a formatted string
`rtrim`	Strips whitespace from the end of a string
`setlocale`	Sets locale information
`similar_text`	Calculates the similarity between two strings
`sprintf`	Returns a formatted string
`sscanf`	Parses input from a string according to a format
`str_ireplace`	Case-insensitive version of the `str_replace` function.
`str_pad`	Pads a string with another string
`str_repeat`	Repeats a string
`str_replace`	Replaces all occurrences of the search string with the replacement string

Function	Purpose
str_shuffle	Shuffles a string randomly
str_split	Converts a string to an array
str_word_count	Returns information about words used in a string
strcasecmp	Binary case-insensitive string comparison
strchr	Alias of the strstr function
strcmp	Binary-safe string comparison
strip_tags	Strips HTML and PHP tags from a string
stripos	Finds position of first occurrence of a case-insensitive string
stristr	Case-insensitive version of the strstr function
strlen	Gets a string's length
strnatcasecmp	Case-insensitive string comparisons
strnatcmp	String comparisons using a "natural order" algorithm
strncasecmp	Binary case-insensitive string comparison of the first n characters
strncmp	Binary-safe string comparison of the first n characters
strpos	Finds position of first occurrence of a string
strrchr	Finds the last occurrence of a character in a string
strrev	Reverses a string
strripos	Finds the position of last occurrence of a case-insensitive string
strrpos	Finds the position of last occurrence of a char in a string
strspn	Finds the length of initial segment matching mask
strstr	Finds the first occurrence of a string
strtolower	Converts a string to lowercase
strtoupper	Converts a string to uppercase
strtr	Translates certain characters
substr_compare	Binary-safe (optionally case-insensitive) comparison of two strings from an offset
substr_count	Counts the number of substring occurrences
substr_replace	Replaces text within part of a string
substr	Returns part of a string
trim	Strips whitespace from the beginning and end of a string

Using the String Functions

Here's an example that puts some of the useful string functions to work:

```
<?php
    echo trim("    No worries."), "\n";
    echo substr("No worries.", 3, 7), "\n";
    echo "\"worries\" starts at position ", strpos("No worries.", "worries"), "\n";
    echo ucfirst("no worries."), "\n";
    echo "\"No worries.\" is ", strlen("No worries."), " characters long.\n";
    echo substr_replace("No worries.", "problems.", 3, 8), "\n";
    echo chr(65), chr(66), chr(67), "\n";
    echo strtoupper("No worries."), "\n";
?>
```

In this example, we're using `trim` to trim leading spaces from a string, `substr` to extract a substring from a string, `strpos` to search a string for a substring, `ucfirst` to convert the first character of a string to uppercase, `strlen` to determine a string's length, `substr_replace` to replace a substring with another string, `chr` to convert an ASCII code to a letter (`ASCII 65 = "A", ASCII 66 = "B"`, and so on), and `strtoupper` to convert a string to uppercase.

Here are the results of this script, line by line:

```
No worries.
worries
"worries" starts at position 3
No worries.
"No worries." is 11 characters long.
No problems.
ABC
NO WORRIES.
```

This example shows some of the more powerful string functions at work. The list of string functions is a long one, but you'll usually find what you need in the table—and if not, you can often cobble together a solution using two or more of these functions.

Here's another tip: In PHP, you can also pick out the characters in a string by enclosing the place of the character you want in curly braces, like this:

```
$string = 'No worries.';
$first_character = $string{0};
```

Formatting Strings

There's a pair of string functions that are particularly useful when you want to format data for display (such as when you're formatting numbers in string form): `printf` and `sprintf`. The `printf` function echoes text directly, and you assign the return value of `sprintf` to a string. Here's how you use these functions (items in square brackets, [and], in function specifications like this one are optional):

```
printf (format [, args])
sprintf (format [, args])
```

The `format` string is composed of zero or more *directives*: characters that are copied directly to the result, and *conversion specifications*. Each conversion specification consists of a percent sign (%), followed by one or more of these elements, in order:

- An optional *padding specifier* that indicates which character should be used to pad the results to the correct string size. This may be a space character or a 0 (zero character). The default is to pad with spaces.
- An optional *alignment specifier* that indicates whether the results should be left-justified or right-justified. The default is right-justified (a - character here will make it left-justified).
- An optional number, the *width specifier*, specifying how many characters (minimum) this conversion should result in.
- An optional *precision specifier* that indicates how many decimal digits should be displayed for floating-point numbers. (There is no effect for types other than `float`.)
- A *type specifier* that says what type the argument data should be treated as.

Here are the possible type specifiers:

`%`	A literal percent character. No argument is required.
`b`	The argument is treated as an integer, and presented as a binary number.
`c`	The argument is treated as an integer, and presented as the character with that ASCII value.
`d`	The argument is treated as an integer, and presented as a (signed) decimal number.
`u`	The argument is treated as an integer, and presented as an unsigned decimal number.
`f`	The argument is treated as a float, and presented as a floating-point number.

- The argument is treated as an integer, and presented as an octal number.

s The argument is treated as and presented as a string.

x The argument is treated as an integer and presented as a hexadecimal number (with lowercase letters).

X The argument is treated as an integer and presented as a hexadecimal number (with uppercase letters).

These functions take a little getting used to, especially when you're formatting floating point values. For example, a format specifier of `%6.2` means that a floating point number will be given six places in the display, with two places behind the decimal point. Here's an example that puts `printf` and `sprintf` to work:

```
<?php
    printf("I have %s apples and %s oranges.\n", 4, 56);

    $year = 2005;
    $month = 4;
    $day = 28;
    printf("%04d-%02d-%02d\n", $year, $month, $day);

    $price = 5999.99;
    printf("\$%01.2f\n", $price);

    printf("%6.2f\n", 1.2);
    printf("%6.2f\n", 10.2);
    printf("%6.2f\n", 100.2);

    $string = sprintf("Now I have %s apples and %s oranges.\n", 3, 5);
    echo $string;
?>
```

In this example, we're formatting simple integers as strings, aligning floating-point numbers vertically so the decimal point lines up, and so on. Here's what you see when you run this script at the command line:

```
I have 4 apples and 56 oranges.
2005-04-28
$5999.99
  1.20
 10.20
100.20
Now I have 3 apples and 5 oranges.
```

NOTE

Another function useful for formatting numbers is `number_format()`.

Converting to and from Strings

Converting between string format and other formats is a common task on the Internet because the data passed from the browser to the server and back in text strings. To convert to a string, you can use the `(string)` cast or the `strval` function; here's what this might look like:

```
<?php
    $float = 1.2345;
    echo (string) $float, "\n";
    echo strval($float), "\n";
?>
```

A boolean `TRUE` value is converted to the string `"1"`, and the `FALSE` value is represented as `""` (empty string). An integer or floating point number (`float`) is converted to a string representing the number with its digits (including the exponent part for floating point numbers). The value `NULL` is always converted to an empty string.

You can also convert a string to a number. The string will be treated as a `float` if it contains any of the characters '`.`', '`e`', or '`E`'. Otherwise, it will be treated as an integer.

PHP determines the numeric value of a string from the *initial part* of the string. If the string starts with numeric data, it will use that. Otherwise, the value will be 0 (zero). Valid numeric data consists of an optional sign (`+` or `-`), followed by one or more digits (including a decimal point if you're using it) and an optional exponent (the exponent part is an '`e`' or '`E`', followed by one or more digits).

PHP will do the right thing if you start using a string in a numeric context, as when you start adding values together. Here are some examples to make all this clearer:

```
<?php
    $number = 1 + "14.5";
    echo "$number\n";
    $number = 1 + "-1.5e2";
    echo "$number\n";
    $text = "5.0";
    $number = (float) $text;
    echo $number / 2.0, "\n";
?>
```

And here's what you see when you run this script:

```
15.5
-149
2.5
```

Creating Arrays

It's time to take the next step up in PHP sophistication: arrays. Arrays are collections of values stored under a single name, and they're a big part of PHP work. You use arrays when you have a set of data to work with, such as the test scores of a set of students. Arrays are easy to handle in PHP because each data item, or *element*, can be accessed with an index value.

You can create arrays by assigning data to them, just as you do with other variables. You can give arrays the same names as you give to standard variables, and like variable names, array names begin with a $. PHP knows you're working with an array if you include [] after the name, like this:

```
$fruits[1] = "pineapple";
```

This creates an array named `$fruits` and sets the element at index `1` to `"pineapple"`. From now on, you can refer to this element as you would any simple variable—you just include the index value to make sure that you reference the data you want, like this:

```
echo $fruits[1];
```

This statement echoes `"pineapple"`. And you can add new values with different numeric indexes:

```
$fruits[2] = "pomegranate";
$fruits[3] = "tangerine";
```

Now you can refer to `$fruits[1]` (which is `"pineapple"`), `$fruits[2]` (which is `"pomegranate"`), and `$fruits[3]` (which is `"tangerine"`). In this case, we've stored strings using numeric indexes, but you can also use string indexes. Here's an example:

```
$apple_inventory["Pittsburgh"] = 2343;
$apple_inventory["Albany"] = 5778;
$apple_inventory["Houston"] = 18843;
```

You can refer to the values in this array by string, as `$apple_inventory["Pittsburgh"]` (which holds `2343`), `$apple_inventory["Albany"]` (which holds `5778`), and `$apple_inventory["Houston"]` (which holds `18843`). Note that in PHP, the same array can have *both* numeric and text indexes, if you want to set things up that way.

There's also a shortcut for creating arrays—you can simply use `[]` after the array's name. Here's an example:

```
$fruits[] = "pineapple";
$fruits[] = "pomegranate";
$fruits[] = "tangerine";
```

In this case, `$fruits[0]` will end up holding `"pineapple"`, `$fruits[1]` will hold `"pomegranate"`, and `$fruits[2]` will hold `"tangerine"`.

PHP starts numbering array elements with `0`. If you wanted to loop over all the elements in this array, you'd start with `0`, as in this `for` loop—note that you use the *`count`* function to find the number of elements in an array:

```
for ($index = 0; $index < count($fruits); $index++){
    echo $fruits[$index], "\n";}
```

Here's an even shorter shortcut for creating an array, using the PHP *`array`* function:

```
$fruits = array("pineapple", "pomegranate", "tangerine");
```

This also creates the same array, starting from an index value of `0`. What if you wanted to start with an index value of `1`? You could specify that with `=>` like this:

```
$fruits = array(1 => "pineapple", "pomegranate", "tangerine");
```

Now the array would look like this:

```
$fruits[1] = "pineapple";
$fruits[2] = "pomegranate";
$fruits[3] = "tangerine";
```

You can create arrays with text as index values in the same way:

```
$apple_inventory = array("Pittsburgh" => 2343,
"Albany" => 5778, "Houston"] => 18843);
```

This creates the following array:

```
$apple_inventory["Pittsburgh"] = 2343;
$apple_inventory["Albany"] = 5778;
$apple_inventory["Houston"] = 18843;
```

The `=>` operator lets you specify *key/value* pairs. For example, `"Pittsburgh"` is the key for the first element, and `2343` is the value.

NOTE

Here's another shortcut: if you have a well-defined range of data, you can automatically create array elements to match with the `range` function, such as the numbers 1 to 10 or characters "`a`" to "`z`" like this: `$values = range("a", "z");`.

Modifying Arrays

After you've created an array, what about modifying it? No problem—you can modify the values in arrays as easily as other variables. One way is to access an element in an array simply by referring to it by index. For example, say you have this array:

```
$fruits[1] = "pineapple";
$fruits[2] = "pomegranate";
$fruits[3] = "tangerine";
```

Now say you want to change the value of `$fruits[2]` to `"watermelon"`. No problem at all:

```
$fruits[1] = "pineapple";
$fruits[2] = "pomegranate";
$fruits[3] = "tangerine";

$fruits[2] = "watermelon";
```

Then say you wanted to add a new element, `"grapes"`, to the end of the array. You could do that by referring to `$fruits[]`, which is PHP's shortcut for adding a new element:

```
$fruits[0] = "pineapple";
$fruits[1] = "pomegranate";
$fruits[2] = "tangerine";

$fruits[2] = "watermelon";

$fruits[] = "grapes";
```

All that's left is to loop over the array and display the array contents, as shown in Example 3-1, phparray.php.

EXAMPLE 3-1 Modifying an array's contents, phparray.php

```
<HTML>
        <HEAD>
            <TITLE>
                Modifying an array
            </TITLE>
        </HEAD>

        <BODY>
            <H1>
                Modifying an array
            </H1>

        <?php
            $fruits[0] = "pineapple";
            $fruits[1] = "pomegranate";
```

```
            $fruits[2] = "tangerine";

            $fruits[2] = "watermelon";

            $fruits[] = "grapes";

            for ($index = 0; $index < count($fruits); $index++){
                echo $fruits[$index], "<BR>";             }
        ?>
    </BODY>
</HTML>
```

The results appear in Figure 3-1. As you can see, we not only were able to modify `$fruits[2]` successfully but were also able to add `"grapes"` to the end of the array.

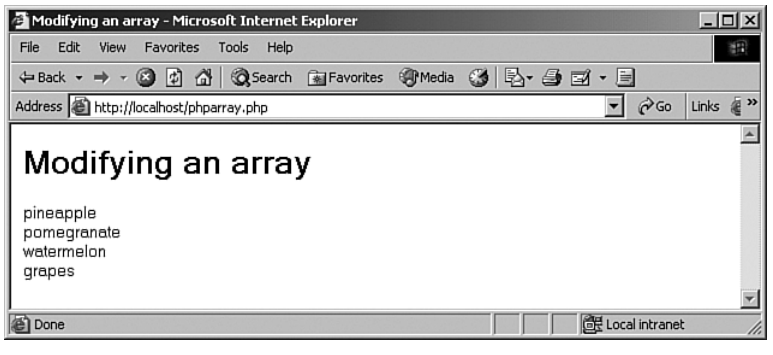

FIGURE 3-1 Modifying an array.

You can also copy a whole array at once if you just assign it to another array:

```
<?php
    $fruits[0] = "pineapple";
    $fruits[1] = "pomegranate";
    $fruits[2] = "tangerine";
    $fruits[2] = "watermelon";
    $fruits[] = "grapes";
    $produce = $fruits;
    echo $produce[2];
?>
```

This script gives you this output:

```
watermelon
```

Removing Array Elements

Another way of modifying arrays is to remove elements from them. To remove an element, you might try setting an array element to an empty string, `""`, like this:

```
<?php
    $fruits[0] = "pineapple";
    $fruits[1] = "pomegranate";
    $fruits[2] = "tangerine";

    $fruits[1] = "";

    for ($index = 0; $index < count($fruits); $index++){
        echo $fruits[$index], "\n";
    }
?>
```

But that doesn't remove the element; it only stores a blank in it:

```
pineapple

tangerine
```

To remove an element from an array, use the *`unset`* function:

```
unset($values[3]);
```

This actually removes the element `$values[3]`. Here's how that might work in our example:

```
<?php
    $fruits[0] = "pineapple";
    $fruits[1] = "pomegranate";
    $fruits[2] = "tangerine";

    unset($fruits[1]);

    for ($index = 0; $index < count($fruits); $index++){
        echo $fruits[$index], "\n";
    }
?>
```

Now when you try to display the element that's been unset, you'll get a warning:

```
pineapple
PHP Notice:  Undefined offset:  1 in C:\php\t.php on line 8
```

Looping Over Arrays

You already know you can loop over an array using a `for` loop and the `count` function, which determines how many elements an array contains:

```
<?php
    $fruits[0] = "pineapple";
    $fruits[1] = "pomegranate";
    $fruits[2] = "tangerine";
    for ($index = 0; $index < count($fruits); $index++){
        echo $fruits[$index], "\n";
    }
?>
```

Here's what you get:

```
pineapple
pomegranate
tangerine
```

There's also a function for easily displaying the contents of an array, `print_r`:

```
<?php
    $fruits[0] = "pineapple";
    $fruits[1] = "pomegranate";
    $fruits[2] = "tangerine";
    print_r($fruits);
?>
```

Here are the results:

```
Array
(
    [0] => pineapple
    [1] => pomegranate
    [2] => tangerine
)
```

The `foreach` statement was specially created to loop over collections such as arrays. This statement has two forms:

```
foreach (array_expression as $value) statement
foreach (array_expression as $key => $value) statement
```

The first form of this statement assigns a new element from the array to `$value` each time through the loop. The second form places the current element's key, another name for its index, in `$key` and its value in `$value` each time through the loop. For example, here's how you can display all the elements in an array using `foreach`:

```
<?php
    $fruits = array("pineapple", "pomegranate", "tangerine");
    foreach ($fruits as $value) {
        echo "Value: $value\n";
    }
?>
```

Here are the results:

```
Value: pineapple
Value: pomegranate
Value: tangerine
```

And here's how you can display both the keys and values of an array:

```
<?php
    $fruits = array("pineapple", "pomegranate", "tangerine");

    foreach ($fruits as $key => $value) {
        echo "Key: $key; Value: $value\n";
    }
?>
```

Here are the results:

```
Key: 0; Value: pineapple
Key: 1; Value: pomegranate
Key: 2; Value: tangerine
```

You can even use a `while` loop to loop over an array if you use a new function, *`each`*. The `each` function is meant to be used in loops over collections such as arrays; each time through the array, it returns the current element's key and value and then moves to the next element. To handle a multiple-item return value from an array, you can use the *`list`* function, which will assign the two return values from each to separate variables.

Here's what this looks like for our `$fruits` array:

```
<?php
    $fruits = array("pineapple", "pomegranate", "tangerine");

    while (list($key, $value) = each ($fruits)) {
        echo "Key: $key; Value: $value\n";
    }
?>
```

Here's what you get from this script:

```
Key: 0; Value: pineapple
Key: 1; Value: pomegranate
Key: 2; Value: tangerine
```

Listing of the Array Functions

Just as it has many string functions, PHP also has many array functions. You can see a sample of them in Table 3-2.

TABLE 3-2 The Array Functions

Function Name	Purpose
`array_chunk`	Splits an array into chunks
`array_combine`	Creates an array by using one array for the keys and another for the values
`array_count_values`	Counts the values in an array
`array_diff`	Computes the difference of arrays
`array_fill`	Fills an array with values
`array_intersect`	Computes the intersection of arrays
`array_key_exists`	Checks whether the given key or index exists in the array
`array_keys`	Returns the keys in an array
`array_merge`	Merges two or more arrays
`array_multisort`	Sorts multiple or multidimensional arrays
`array_pad`	Pads array to the specified length with a value
`array_pop`	Pops the element off the end of an array
`array_push`	Pushes one or more elements onto the end of array
`array_rand`	Picks one or more random elements out of an array
`array_reduce`	Reduces the array to a single value with a callback function
`array_reverse`	Returns an array with elements in reverse order
`array_search`	Searches the array for a given value and returns the corresponding key
`array_shift`	Shifts an element off the beginning of array
`array_slice`	Extracts a slice of the array
`array_sum`	Calculates the sum of values in an array
`array_unique`	Removes duplicate elements from an array
`array_unshift`	Adds one or more elements to the beginning of an array
`array_walk`	Calls a user-supplied function on every member of an array
`array`	Creates an array
`asort`	Sorts an array and maintains index association
`count`	Counts the elements in an array

continues

TABLE 3-2 continued

Function Name	Purpose
current	Returns the current element in an array
each	Returns the current key and value pair from an array and advances the array cursor
in_array	Checks whether a value exists in an array
key	Gets a key from an associative array
krsort	Sorts an array by key in reverse order
ksort	Sorts an array by key
list	Assigns variables as if they were an array
natcasesort	Sorts an array using a case-insensitive "natural order" algorithm
natsort	Sorts an array using a "natural order" algorithm
pos	Alias of the current function
reset	Sets the pointer of an array to its first element
rsort	Sorts an array in reverse order
shuffle	Shuffles an array's elements
sizeof	Alias of the count function
sort	Sorts an array
usort	Sorts an array by values with a user-defined comparison function

We'll see a number of the most important array functions in this chapter, such as those that let you sort the contents of an array, which are coming up next.

Sorting Arrays

PHP offers all kinds of ways to sort the data in arrays, starting with the simple `sort` function, which you use on arrays with numeric indexes. In the following example, we create an array, display it, sort it, and then display it again:

```
<?php
    $fruits[0] = "tangerine";
    $fruits[1] = "pineapple";
    $fruits[2] = "pomegranate";

    print_r($fruits);

    sort($fruits);

    print_r($fruits);
?>
```

Here are the results—as you can see, the new array is in sorted order; note also that the elements have been given new numeric indexes:

```
Array
(
    [0] => tangerine
    [1] => pineapple
    [2] => pomegranate
)
Array
(
    [0] => pineapple
    [1] => pomegranate
    [2] => tangerine
)
```

You can sort an array in reverse order if you use `rsort` instead:

```
<?php
    $fruits[0] = "tangerine";
    $fruits[1] = "pineapple";
    $fruits[2] = "pomegranate";

    print_r($fruits);
    rsort($fruits);
    print_r($fruits);
?>
```

Here's what you get:

```
Array
(
    [0] => tangerine
    [1] => pineapple
    [2] => pomegranate
)
Array
(
    [0] => tangerine
    [1] => pomegranate
    [2] => pineapple
)
```

What if you have an array that uses text keys? Unfortunately, if you use `sort` or `rsort`, the keys are replaced by numbers. If you want to retain the keys, use `asort` instead, as in this example:

```
<?php
    $fruits["good"] = "tangerine";
    $fruits["better"] = "pineapple";
    $fruits["best"] = "pomegranate";

    print_r($fruits);

    asort($fruits);

    print_r($fruits);
?>
```

Here are the results:

```
Array
(
    [good] => tangerine
    [better] => pineapple
    [best] => pomegranate
)
Array
(
    [better] => pineapple
    [best] => pomegranate
    [good] => tangerine
)
```

You can use `arsort` to sort arrays such as this in reverse order. What if you wanted to sort an array such as this one based on keys, not values? Just use `ksort` instead. To sort by reverse by keys, use `krsort`. You can even define your own sorting operations with a custom sorting function you use with PHP's `usort`.

Navigating through Arrays

PHP also includes a number of functions for navigating through arrays. That navigation is done with an array *pointer*, which holds the current location in an array. Here's how it works. Say you have this array:

```
$vegetables[0] = "corn";
$vegetables[1] = "broccoli";
$vegetables[2] = "zucchini";

print_r($vegetables);
echo "<BR>";
```

You can get the current element in this array with the `current` function:

```
echo "Current: ", current($vegetables), "<BR>";
```

And you can move the pointer to the next element with the `next` function:

```
echo "Next: ", next($vegetables), "<BR>";
```

The `prev` function moves the pointer back:

```
echo "Prev: ", prev($vegetables), "<BR>";
```

The `end` function moves the pointer to the last element:

```
echo "End: ", end($vegetables), "<BR>";
```

Want to move back to the beginning of the array? Use `reset`:

```
reset($vegetables);
```

And we can display the new current element, which will be the beginning of the array, like this:

```
echo "Current: ", current($vegetables), "<BR>";
```

Let's put all this together in a web page, as you can see in Example 3-2, phpnavigate.php.

EXAMPLE 3-2 Navigating through an array, phpnavigate.php

```
<HTML>
        <HEAD>
            <TITLE>
                Navigating through an array
            </TITLE>
        </HEAD>

        <BODY>
            <H1>
                Navigating through an array
            </H1>
        <?php

            $vegetables[0] = "corn";
            $vegetables[1] = "broccoli";
            $vegetables[2] = "zucchini";

            print_r($vegetables);
            echo "<BR>";

            echo "Current: ", current($vegetables), "<BR>";
            echo "Next: ", next($vegetables), "<BR>";
            echo "Prev: ", prev($vegetables), "<BR>";
            echo "End: ", end($vegetables), "<BR>";
            echo "Resetting the array.<BR>";
            reset($vegetables);
            echo "Current: ", current($vegetables), "<BR>";

        ?>
    </BODY>
</HTML>
```

This page appears in Figure 3-2, where we've used the array pointer to move through an array. Very nice.

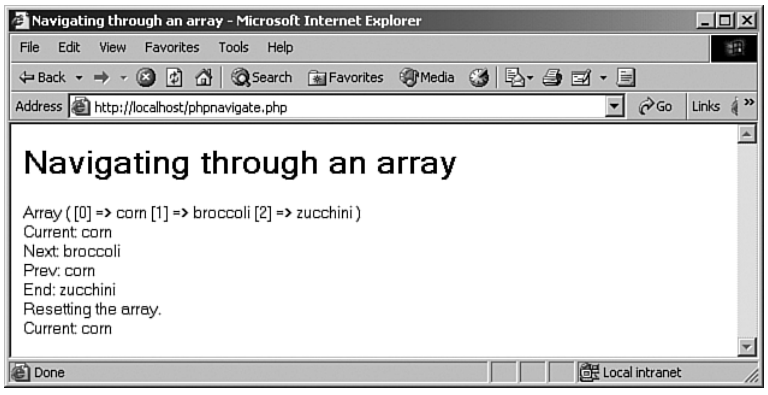

FIGURE 3-2 Navigating through an array.

Imploding and Exploding Arrays

You can also convert between strings and arrays by using the PHP *implode* and *explode* functions: `implode` implodes an array to a string, and `explode` explodes a string into an array.

For example, say you want to put an array's contents into a string. You can use `implode`, passing it the text you want to separate each element with in the output string (in this example, we use a comma) and the array to work on:

```
<?php
    $vegetables[0] = "corn";
    $vegetables[1] = "broccoli";
    $vegetables[2] = "zucchini";
    $text = implode(",", $vegetables);
    echo $text;
?>
```

This gives you:

```
corn,broccoli,zucchini
```

There are no spaces between the items in this string, however, so we change the separator string from "," to ", ":

```
$text = implode(", ", $vegetables);
```

The result is:

```
corn, broccoli, zucchini
```

What about exploding a string into an array? To do that, you indicate the text that you want to split the string on, such as ", ", and pass that to `explode`. Here's an example:

```
<?php
    $text = "corn, broccoli, zucchini";
    $vegetables = explode(", ", $text);
    print_r($vegetables);
?>
```

And here are the results. As you can see, we exploded the string into an array correctly:

```
Array
(
    [0] => corn
    [1] => broccoli
    [2] => zucchini
)
```

Extracting Variables from Arrays

The *extract* function is handy for copying the elements in arrays to variables if your array is set up with string index values. For example, take a look at this case, where we have an array with string indexes:

```
$fruits["good"] = "tangerine";
$fruits["better"] = "pineapple";
$fruits["best"] = "pomegranate";
    .
    .
    .
```

When you call the `extract` function on this array, it creates variables corresponding to the string indexes: `$good`, `$better`, and so on:

```
$fruits["good"] = "tangerine";
$fruits["better"] = "pineapple";
$fruits["best"] = "pomegranate";

extract($fruits);
    .
    .
    .
```

Take a look at how this works in Example 3-3, phpextract.php.

EXAMPLE 3-3 Extracting variables from an array, phpextract.php

```
<HTML>
    <HEAD>
        <TITLE>Extracting variables from an array</TITLE>
    </HEAD>

    <BODY>
        <H1>Extracting variables from an array</H1>
    <?php
        $fruits["good"] = "tangerine";
        $fruits["better"] = "pineapple";
        $fruits["best"] = "pomegranate";

        extract($fruits);

        echo "\$good = $good<BR>";
        echo "\$better = $better<BR>";
        echo "\$best = $best<BR>";
    ?>
  </BODY>
</HTML>
```

Now $good will hold "tangerine", $better will hold "pineapple", and $best will hold "pomegranate". You can see the results in Figure 3-3.

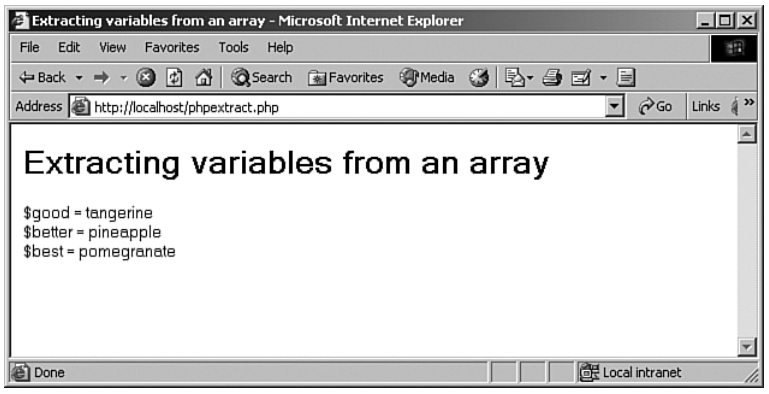

FIGURE 3-3 Filling variables from an array.

You can also use the PHP list function to get data from an array like this and store it in as many variables as you like. Here's an example:

```
<?php
    $vegetables[0] = "corn";
    $vegetables[1] = "broccoli";
    $vegetables[2] = "zucchini";
    list($first, $second) = $vegetables;
    echo $first, "\n";
    echo $second;
?>
```

And here is the result:

```
corn
broccoli
```

Can you go the opposite way and copy variables into an array? Sure, just use the *compact* function. You pass this function the *names* of variables (with the $), and compact finds those variables and stores them all in an array:

```
<?php
    $first_name  = "Cary";
    $last_name = "Grant";
    $role = "Actor";
    $subarray = array("first_name", "last_name");
    $resultarray = compact("role", $subarray);
?>
```

Merging and Splitting Arrays

You can also cut up and merge arrays when needed. For example, say you have a three-item array of various fruits and want to get a subarray consisting of the last two items. You can do this with the `array_slice` function, passing it the array you want to get a section of, the *offset* at which to start, and the *length* of the array you want to create:

```
<?php
    $fruits["good"] = "tangerine";
    $fruits["better"] = "pineapple";
    $fruits["best"] = "pomegranate";
    $subarray = array_slice($fruits, 1, 2);
    foreach ($subarray as $value) {
        echo "Fruit: $value\n";
    }
?>
```

Here are the results:

```
Fruit: pineapple
Fruit: pomegranate
```

If `offset` is negative, the sequence will be measured from the end of the array. If `length` is negative, the sequence will stop that many elements from the end of the array.

NOTE
If you don't give the length of the subarray you want, you'll get all the elements to the end (or the beginning, if you're going in the opposite direction) of the array.

You can also merge two or more arrays with `array_merge`:

```
<?php
    $fruits = array("pineapple", "pomegranate", "tangerine");
    $vegetables = array("corn", "broccoli", "zucchini");

    $produce = array_merge($fruits, $vegetables);

    foreach ($produce as $value) {
        echo "Produce item: $value\n";
    }
?>
```

And here's what you get (see also "Using the Array Operators" in this chapter):

```
Produce item: pineapple
Produce item: pomegranate
Produce item: tangerine
Produce item: corn
Produce item: broccoli
Produce item: zucchini
```

Comparing Arrays

PHP also includes support for comparing arrays and determining which elements are the same—or which are different. For example, say you have these two arrays, where only the second element is the same:

```
$local_fruits = array("apple", "pomegranate", "orange");
$tropical_fruits = array("pineapple", "pomegranate", "papaya");
```

You can use the `array_diff` function to create a new array, which we'll call `$difference`, that holds the elements that are different between the two arrays:

```
<?php
    $local_fruits = array("apple", "pomegranate", "orange");
    $tropical_fruits = array("pineapple", "pomegranate", "papaya");

    $difference = array_diff($local_fruits, $tropical_fruits);

    foreach ($difference as $key => $value) {
        echo "Key: $key; Value: $value\n";
    }
?>
```

Here's what this script displays:

```
Key: 0; Value: apple
Key: 2; Value: orange
```

Now say you're working with two arrays that use text indexes, and you want to see which elements have either different keys or values when comparing the arrays:

```
$local_fruits = array("fruit1" => "apple", "fruit2" => "pomegranate",
    "fruit3" => "orange");
$tropical_fruits = array("fruit1" => "pineapple", "fruit_two" => "pomegranate",
    "fruit3" => "papaya");
```

You can determine which array elements have either different keys or values by using the `array_diff_assoc` function (arrays with text indexes are also called *associative arrays*, hence the name `array_diff_assoc`) this way:

```
<?php
    $local_fruits = array("fruit1" => "apple", "fruit2" => "pomegranate",
        "fruit3" => "orange");
    $tropical_fruits = array("fruit1" => "pineapple", "fruit_two" => "pomegranate",
        "fruit3" => "papaya");

    $difference = array_diff_assoc($local_fruits, $tropical_fruits);
```

```
    foreach ($difference as $key => $value) {
        echo "Key: $key; Value: $value\n";
    }
?>
```

And here's what you get—note that we've been able to find all array elements that differ in either key or value:

```
Key: fruit1; Value: apple
Key: fruit2; Value: pomegranate
Key: fruit3; Value: orange
```

What if you want to find all array elements that the arrays have in common instead? In that case, use `array_intersect`. Here's an example, where we're finding the elements in common between our two arrays:

```
<?php
    $local_fruits = array("apple", "pomegranate", "orange");
    $tropical_fruits = array("pineapple", "pomegranate", "papaya");

    $common = array_intersect($local_fruits, $tropical_fruits);

    foreach ($common as $key => $value) {
        echo "Key: $key; Value: $value\n";
    }
?>
```

And this is what you get:

```
Key: 1; Value: pomegranate
```

You can also do the same with arrays that use text indexes if you use `array_intersect_assoc`:

```
<?php
    $local_fruits = array("fruit1" => "apple", "fruit2" => "pomegranate",
        "fruit3" => "orange");
    $tropical_fruits = array("fruit1" => "pineapple", "fruit2" => "pomegranate",
        "fruit3" => "papaya");

    $common = array_intersect_assoc($local_fruits, $tropical_fruits);

    foreach ($common as $key => $value) {
        echo "Key: $key; Value: $value\n";
    }
?>
```

And here's what this script gives you:

```
Key: fruit2; Value: pomegranate
```

Comparing arrays in PHP? No problem at all.

Manipulating the Data in Arrays

You can do even more with the data in arrays. For example, if you want to delete duplicate elements, you can use `array_unique`:

```
<?php
    $scores = array(65, 60, 70, 65, 65);
    print_r($scores);
    $scores = array_unique($scores);
    print_r($scores);
?>
```

Here's what this script looks like when you run it—note that the duplicate elements are removed:

```
Array
(
    [0] => 65
    [1] => 60
    [2] => 70
    [3] => 65
    [4] => 65
)
Array
(
    [0] => 65
    [1] => 60
    [2] => 70
)
```

Here's another useful array function—`array_sum`, which adds all the values in an array:

```
<?php
    $scores = array(65, 60, 70, 64, 66);

    echo "Average score = ", array_sum($scores) / count($scores);
?>
```

In this case, we're finding the average student score from the `$scores` array:

```
Average score = 65
```

And here's another one—the `array_flip` function will flip an array's keys and values. You can see that at work in Example 3-4, phpflip.php.

EXAMPLE 3-4 Flipping an array, phpflip.php

```
<HTML>
        <HEAD>
            <TITLE>
                Flipping an array
            </TITLE>
        </HEAD>

        <BODY>
            <H1>
                Flipping an array
            </H1>
            <?php
                $local_fruits = array("fruit1" => "apple", "fruit2" =>
                    "pomegranate", "fruit3" => "orange");
                foreach ($local_fruits as $key => $value) {
                    echo "Key: $key; Value: $value<BR>";
                }

                echo "<BR>";

                $local_fruits = array_flip($local_fruits);

                foreach ($local_fruits as $key => $value) {
                    echo "Key: $key; Value: $value<BR>";
                }
            ?>
    </BODY>
</HTML>
```

The results appear in Figure 3-4—note that the keys and values were indeed flipped.

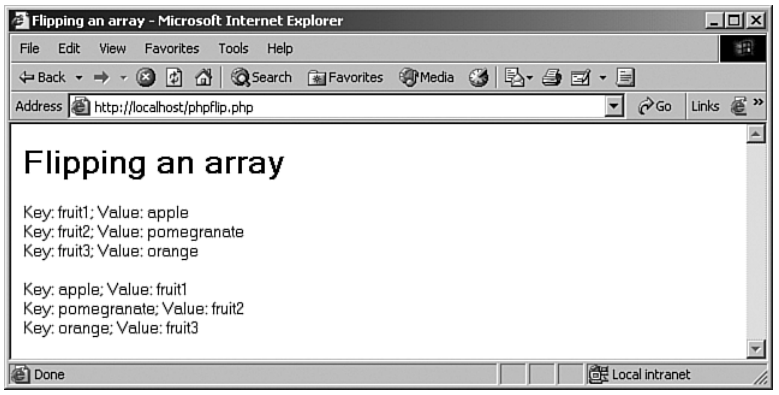

FIGURE 3-4 Flipping an array.

Creating Multidimensional Arrays

So far, we've been using only one-dimensional arrays, with only one set of keys. However, arrays with multiple sets of keys are also possible, and sometimes you need them. You might, for example, store test scores for various students like this:

```
$test_scores["Frank"] = 95;
$test_scores["Mary"] = 87;
```

But what if you gave a second test? You can add a second index to stand for the test number; here's what that might look like:

```
<?php
    $test_scores["Frank"][1] = 95;
    $test_scores["Frank"][2] = 85;
    $test_scores["Mary"][1] = 87;
    $test_scores["Mary"][2] = 93;
    print_r($test_scores);
?>
```

Now `$test_scores["Frank"][1]` is Frank's test score on the first test, `$test_scores["Frank"][2]` is his score on the second test, and so on. This script displays the new multidimensional array with `print_r`:

```
    [Frank] => Array
        (
            [1] => 95
            [2] => 85
        )
    [Mary] => Array
        (
            [1] => 87
            [2] => 93
        )
```

You can access individual elements using both indexes, like this:

```
echo "Frank's first test score is ", $test_scores["Frank"][1], "\n";
```

Want to interpolate an array item in double quotes? Enclose it in curly braces (and use single quotes for any text keys to avoid conflict with the double quotes):

```
echo "Frank's first test score is {$test_scores['Frank'][1]}\n";
```

You can also use the flowing syntax to create multidimensional arrays—but note that this will start the arrays off at an index value of 0:

```
<?php
    $test_scores["Frank"][] = 95;
    $test_scores["Frank"][] = 85;
    $test_scores["Mary"][] = 87;
    $test_scores["Mary"][] = 93;
    print_r($test_scores);
?>
```

In PHP, multidimensional arrays can be thought of as *arrays of arrays*. For example, a two-dimensional array may be considered as a single-dimensional array where each element is a single-dimensional array. Here's an example:

```
<?php
    $test_scores = array("Frank" => array(95, 85), "Mary" => array(87, 93));
    print_r($test_scores);
?>
```

This is what the results look like:

```
    [Frank] => Array
        (
            [0] => 95
            [1] => 85
        )

    [Mary] => Array
        (
            [0] => 87
            [1] => 93
        )
```

What if you wanted to start the array at index 1 instead of 0? You could do this:

```
<?php
    $test_scores = array("Frank" => array(1 => 95, 2 => 85),
        "Mary" => array(1 => 87, 2 => 93));
    print_r($test_scores);
?>
```

And here's what you would get:

```
    [Frank] => Array
        (
            [1] => 95
            [2] => 85
        )

    [Mary] => Array
        (
            [1] => 87
            [2] => 93
        )
```

Looping Over Multidimensional Arrays

So what about looping over multidimensional arrays? For example, what if your array contains two dimensions? Looping over it isn't a problem—just loop over the first index first and then the second index in an internal loop. You can do that by *nesting* a `for` loop inside another `for` loop:

```
<?php
    $test_scores[0][] = 95;
    $test_scores[0][] = 85;
    $test_scores[1][] = 87;
    $test_scores[1][] = 93;
    for ($outer_index = 0; $outer_index < count($test_scores);
        $outer_index++){
        for($inner_index = 0; $inner_index < count($test_scores[$outer_index]);
            $inner_index++){
            echo "\$test_scores[$outer_index][$inner_index] = ",
                $test_scores[$outer_index][$inner_index], "\n";
        }
    }
?>
```

In this way, we can set the first index in the array, print out all elements by incrementing the second index, and then increment the first index to move on. Here's what you get, just as you should:

```
$test_scores[0][0] = 95
$test_scores[0][1] = 85
$test_scores[1][0] = 87
$test_scores[1][1] = 93
```

You can also use `foreach` loops—and in fact they're a better idea than `for` loops if you're using text indexes (which can't be incremented for iterating through a loop). In the following example, each time through the outer loop, we extract a new single-dimensional array to iterate over:

```
<?php
    $test_scores["Frank"]["first"] = 95;
    $test_scores["Frank"]["second"] = 85;
    $test_scores["Mary"]["first"] = 87;
    $test_scores["Mary"]["second"] = 93;
     foreach ($test_scores as $outer_key => $single_array) {
        .
        .
        .
    }
?>
```

And we iterate over the single-dimensional array in the inner `foreach` loop, as you can see in phpmultidimensional.php, Example 3-5.

EXAMPLE 3-5 Looping over multidimensional arrays, phpmultidimensional.php

```
<HTML>
        <HEAD>
            <TITLE>
                Looping over multidimensional arrays
            </TITLE>
        </HEAD>

        <BODY>
            <H1>
                Looping over multidimensional arrays
            </H1>
            <?php
                $test_scores["Frank"]["first"] = 95;
                $test_scores["Frank"]["second"] = 85;
                $test_scores["Mary"]["first"] = 87;
                $test_scores["Mary"]["second"] = 93;
                foreach ($test_scores as $outer_key => $single_array) {
                    foreach ($single_array as $inner_key => $value) {
                        echo "\$test_scores[$outer_key][$inner_key] =
                              $value<BR>";
                    }
                }
            ?>
    </BODY>
</HTML>
```

The results appear in Figure 3-5. Very cool.

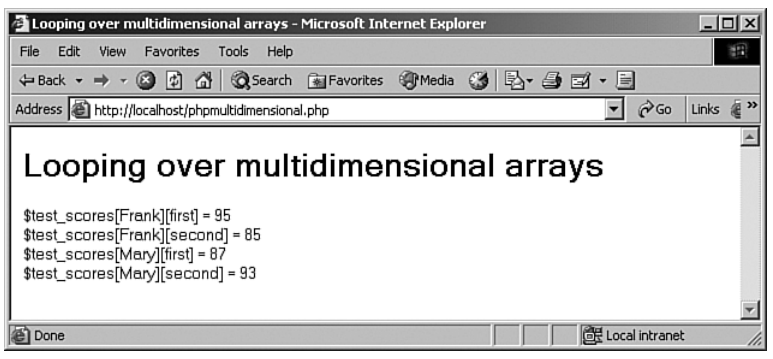

FIGURE 3-5 Looping over multidimensional arrays.

Using the Array Operators

Want more array power? Check out the array operators:

`$a + $b`	Union of `$a` and `$b`.
`$a == $b`	`TRUE` if `$a` and `$b` have the same elements.
`$a === $b`	`TRUE` if `$a` and `$b` have the same elements in the same order.
`$a != $b`	`TRUE` if `$a` is not equal to `$b`.
`$a <> $b`	`TRUE` if `$a` is not equal to `$b`.
`$a !== $b`	`TRUE` if `$a` is not identical to `$b`.

Most of these have to do with comparing arrays, but the + operator is designed to concatenate arrays. You can see an example in phparrayops.php, Example 3-6, where we put to work not only the + operator but also the == operator, checking to see if two arrays have the same elements (in this case, they don't).

EXAMPLE 3-6 Using array operators, phparrayops.php

```
<HTML>
        <HEAD>
            <TITLE>
                Using array operators
            </TITLE>
        </HEAD>

        <BODY>
            <H1>
                Using array operators
            </H1>
            <?php
                $fruits["apples"] = 3839;
                $fruits["oranges"] = 2289;
                $vegetables["broccoli"] = 1991;
                $vegetables["corn"] = 9195;
                echo "\$fruits: ";
                print_r($fruits);
                echo "<BR>";
                echo "\$vegetables: ";
                print_r($vegetables);
                echo "<BR>";
                $produce = $fruits + $vegetables;
                echo "\$produce: ";
                print_r($produce);
                echo "<BR>";
```

continues

EXAMPLE 3-6 continued

```
                if ($fruits == $vegetables){
                    echo "\$fruits has the same elements as \$vegetables<BR>";
                }
                else {
                    echo "\$fruits does not have the same elements as
                        \$vegetables<BR>";
                }
            ?>
    </BODY>
</HTML>
```

The results appear in Figure 3-6, where you can see that the + operator did indeed concatenate the arrays we wanted it to, and the == operator did indeed compare the two arrays properly.

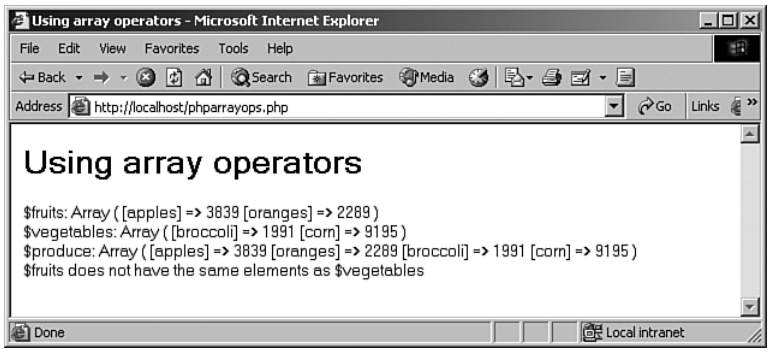

FIGURE 3-6 Using array operators.

Summary

PHP gives you some great ways of working with data in strings and arrays. Both have many different functions available. Here's a summary of the salient points in this chapter:

- The many string functions do everything from searching strings to formatting them for display. Take a look at Table 3-1 for a refresher.
- The `printf` and `sprintf` functions format strings for display.
- The `strstr` function searches a string for a `substring`.
- The `substr_compare` function lets you compare strings.
- You access the items in an array by using a numeric or string index.
- PHP knows you're working with an array if you include `[]` after a variable's name.
- The `unset` function removes items from arrays.
- The `foreach` statement provides a great way of looping over arrays.
- The array functions do everything from merging arrays to searching them.
- You can sort arrays with the `sort` function.
- You can convert between strings and arrays using the PHP `implode` and `explode` functions.
- You can use the `array_diff` function to create a new array that holds the elements that are different between two arrays.
- You can create multidimensional arrays simply by using two array indexes in square brackets, `[` and `]`.

That's it for our coverage of strings and arrays for the moment, both of which we'll see again throughout the book. Now it's time to turn to working with and creating functions in Chapter 4, "Breaking It Up: Functions."