7

SIMPLE ARRAYS

This chapter introduces *arrays*, a compound data type designed for storing and manipulating multiple data items under a single variable name. Arrays allow you to group related data and efficiently apply the same operations to each data item.

At its heart, an array is a mapping of values to keys. Each *value* is a piece of data you want to store in the array, and its *key* is a unique identifier associated with that value so that you can access it from within the array. In this chapter, we'll focus on simple arrays, which use integers as the keys. You'll learn how to create and manipulate simple arrays, and how to iterate over the items in an array by using a `foreach` loop. In the next chapter, we'll explore how to create more sophisticated arrays by using strings (and other data types) as keys, instead of integers.

Creating an Array and Accessing Its Values

Let's start our exploration of arrays by creating a simple array that stores the monthly rainfall totals for a location (Listing 7-1).

```
<?php
$rainfall = [10, 8, 12];
```

Listing 7-1: Declaring a simple array

We declare an array called `$rainfall` by using a sequence of comma-separated values inside square brackets: `[10, 8, 12]`. This is a three-element array, containing the values `10`, `8`, and `12`.

By default, PHP gives each array value an integer as a key. The keys are assigned in sequence, starting from zero: the first value (`10`) has a key of `0`, the second value (`8`) has a key of `1`, and the third value (`12`) has a key of `2`. Accepting this default mapping is what makes `$rainfall` a *simple array* (as opposed to the *sophisticated arrays* with custom key/value mappings that we'll explore in Chapter 8).

Now that we have an array, we can use its keys to access its values individually. In Listing 7-2, we concatenate each value from the `$rainfall` array into a string message and print it out.

```
<?php
$rainfall = [10, 8, 12];

print "Monthly rainfall\n";
❶ print "Jan: " . $rainfall[0] . "\n";
print "Feb: " . $rainfall[1] . "\n";
print "Mar: " . $rainfall[2] . "\n";
```

Listing 7-2: Accessing array elements with their integer keys

We access an item in an array by specifying its key in square brackets, after the array name. For example, `$rainfall[0]` gives us the first value in the `$rainfall` array (`10`), which we concatenate with the string `"Jan: "` ❶. Similarly, we access the second element of the array with `$rainfall[1]`. Since the integer keys start at zero, the last element of an n-member array has a key of $n - 1$. In this case, we access the last element of our three-element array with `$rainfall[2]`. Here's the output of running this script:

```
Monthly rainfall
Jan: 10
Feb: 8
Mar: 12
```

The values `10`, `8`, and `12` have been successfully read from the array and printed using their integer keys `0`, `1`, and `2`.

If you try to access an array element by using a key that hasn't been assigned, you'll get a PHP warning. For example, say we add the following `print` statement to the end of Listing 7-2:

```
print "Apr: " . $rainfall[3] . "\n";
```

This statement will trigger a warning that looks something like this:

```
PHP Warning:  Undefined array key 3 in /Users/matt/main.php on line 8
```

Our array has only three elements, with keys `0`, `1`, and `2`, so no element exists corresponding to `$rainfall[3]`. Later in the chapter, we'll discuss how to avoid warnings like this by first ensuring that an array element with a particular key exists before trying to access it.

THE ARRAY() FUNCTION

In Listing 7-1, we declared an array by writing it out as a literal, enclosing its values in square brackets. Another way to declare an array is to call the `array()` function, passing the array element values as a sequence of comma-separated arguments. Here's how to declare the same `$rainfall` array from Listing 7-1 by using the `array()` function:

```
$rainfall = array(10, 8, 12);
```

Understanding this alternative technique for declaring arrays is important, since you may find it in older programs and some of the PHP documentation pages (*https://www.php.net*). These days, however, the square-bracket notation is more common (and more succinct), so I'll stick to square-bracket notation in this book. Learn more about this function at *https://www.php.net/manual/en/function.array.php*.

Updating an Array

Often you'll need to update an array after you've created it, by adding or removing elements. For example, it's common to start with an empty array, created by assigning an empty set of square brackets `[]` to a variable, and then to add elements to it as a script progresses. In this section, we'll discuss common techniques for changing the contents of an array.

Appending an Element

If you're adding a new element to an array, you'll most often want to add it at the end, an operation known as *appending*. This is such a common task that PHP makes it very easy to do as part of a simple assignment statement. On the left side of the equal sign, you write the array name followed by an empty set of square brackets; on the right side of the equal sign, you write the value you want to append to the array. For example, Listing 7-3 shows a

script that creates an empty array of animals and then appends elements to the end of it.

```
<?php
❶ $animals = [];
$animals[] = 'cat';
$animals[] = 'dog';

print "animals[0] = $animals[0] \n";
print "animals[1] = $animals[1] \n";
```

Listing 7-3: Appending elements to the end of an array

We declare an empty array called `$animals` by writing an empty set of square brackets ❶. Then we add two elements to the end of the array, one at a time. For example, `$animals[] = 'cat'` adds the string value `'cat'` to the end of the array. PHP automatically gives the new element the next available integer as a key. In this case, since `$animals` is empty when `'cat'` is added, it receives a key of `0`. When we then use the same notation to add `'dog'` to the array, that element automatically gets a key of `1`. To confirm this, we print the individual values from the array at the end of the script, resulting in this output:

```
animals[0] = cat
animals[1] = dog
```

The output indicates that `'cat'` was successfully mapped to key `0` of the array, and `'dog'` to `1`. The PHP engine was able to find the array's highest integer key, add 1 to it, and use the result as the next unique integer key when appending to the array.

Adding an Element with a Specific Key

While it's more common to append elements to an array and let PHP do the work of automatically assigning the next available integer key, you can also manually specify an element's key when you're adding it to an array. For example, `$heights[22] = 101` would add the value `101` to the `$heights` array and give it the integer key `22`. If a value already exists at that key, that value will be overwritten. As such, this direct assignment technique is often used to update an existing value in an array rather than add a completely new value. Listing 7-4 expands our `$animals` array script to illustrate how this is done.

```
<?php
$animals = [];
$animals[] = 'cat';
$animals[] = 'dog';
❶ $animals[0] = 'hippo';

var_dump($animals);
```

Listing 7-4: Directly assigning an array element with a specified key

As before, we append `'cat'` and `'dog'` to the `$animals` array. Then we replace the value of the first array element with `'hippo'` by directly assigning this string to key `0` of the array ❶. Here's the output of running this script:

```
array(2) {
    [0]=>
    string(3) "hippo"
    [1]=>
    string(3) "dog"
}
```

Notice that `'hippo'` is now mapped to key `0`, indicating it has replaced the original `'cat'` value.

Be careful when adding a new array element with a specific key. This action can break the sequence of integer keys if an array element doesn't exist for the key you provide. This would happen if you used a key beyond the existing size of the array. Making an array with a break in the sequence of integer keys is permissible, but it can cause issues if you've written code elsewhere that relies on having a continuous sequence of keys. We'll explore nonsequential and non-integer keys when we look at sophisticated arrays in Chapter 8.

Appending Multiple Elements

So far we've been able to add only one element to an array at a time, but the built-in `array_push()` function can add several elements at once to the end of an array. The function takes a variable number of parameters. The first is the array you want to update, and the rest are the new values to be appended—as many as you want. For example, Listing 7-5 revisits the script from Listing 7-3, where we first added elements to the `$animals` array and then printed them, and uses `array_push()` to append two more animals to the end of the array.

```
<?php
$animals = [];
$animals[] = 'cat';
$animals[] = 'dog';
❶ array_push($animals, 'giraffe', 'elephant');

print "animals[0] = $animals[0] \n";
print "animals[1] = $animals[1] \n";
print "animals[2] = $animals[2] \n";
print "animals[3] = $animals[3] \n";
```

Listing 7-5: Using `array_push()` to append multiple values to the end of an array

We call `array_push()`, passing in the `$animals` array and the two string values we want to add, `'giraffe'` and `'elephant'` ❶. Since the new elements are added to the end of the array, they're automatically assigned the next available integer keys, `2` and `3`. We confirm this at the end of the script by

accessing the two additional elements by their keys and printing them out, along with the two original elements:

```
animals[0] = cat
animals[1] = dog
animals[2] = giraffe
animals[3] = elephant
```

The output indicates that `'giraffe'` was successfully mapped to key `2`, and `'elephant'` to `3`.

You may have noticed that when we called the `array_push()` function, we didn't do it as part of an assignment statement, with the function call on the right side of an equal sign and a variable name on the left to capture the function's return value. This is because `array_push()` directly modifies the array passed to it. In this sense, `array_push()` is quite different from the string manipulation functions we looked at in Chapter 3, which created and returned a new string rather than making changes directly to the original string passed to them.

The `array_push()` function can directly modify the provided array because its first parameter has been declared using a pass-by-reference approach. As we discussed in Chapter 5, this means the function is given a direct reference to the value of the argument passed in, as opposed to being given a copy of the argument's value via a pass-by-value approach. We can confirm this by looking at the function's signature in the PHP documentation:

```
array_push(array &$array, mixed ...$values): int
```

The ampersand (`&`) before the first parameter, `&$array`, indicates that this is a pass-by-reference parameter.

Since `array_push()` is directly modifying the array, there's no need for it to return a copy of the array, or for us to use an assignment statement to capture that return value when we call the function. In fact, `array_push()` *does* have a return value, an integer indicating the new length of the array. This can be useful if you need to keep track of the array's length as you're updating it; we didn't need this return value in Listing 7-5, so we simply made a stand-alone call to the function, without assigning the result to a variable.

Removing the Last Element

The built-in `array_pop()` function returns the last item of an array while also removing that item from the array. This is another example of a pass-by-reference function that changes the provided array. In Listing 7-6, we use `array_pop()` to retrieve and remove the last element of our `$animals` array.

```
<?php
$animals = [];
$animals[] = 'cat';
$animals[] = 'dog';
```

```
❶ $lastAnimal = array_pop($animals);
  print "lastAnimal = $lastAnimal\n";
  var_dump($animals);
```

Listing 7-6: Using `array_pop()` to retrieve and remove the last array element

We call `array_pop()`, passing the `$animals` array as an argument, and we store the function's return value in the `$lastAnimal` variable ❶. We then print out `$lastAnimal`, as well as the `$animals` array, to see which elements remain. Here's the result:

```
lastAnimal = dog
array(1) {
    [0]=>
    string(3) "cat"
}
```

The string in variable `$lastAnimal` is `'dog'`, since this was the last of the elements appended to the array. The `var_dump` of `$animals` shows that the array contains only `'cat'` after the call to `array_pop()`, demonstrating how this pass-by-reference function was able to change the array passed into it.

ARRAYS AS STACKS

One of the classic data structures for solving some types of computer tasks is the *stack*. It treats data like a stack of items, such as a pile of books or blocks. You can *push* an element onto the stack by adding it on top of the existing elements, or *pop* the last (topmost) element off the stack.

If you push A, then B, then C, for example, you have a stack with A on the bottom, B in the middle, and C on top. If you then start popping items, you first get C, then B, then A. The most recent item added to the stack is always the first item to be removed. The PHP functions `array_push()` and `array_pop()` mirror these operations, making it easy to create scripts that solve problems by using simple arrays as stacks.

Retrieving Information About an Array

We've considered some functions for modifying an array, but other functions can return useful information about an array without changing it at all. For example, `count( )` returns the number of elements in an array. This can be useful if you want to check whether an array contains anything at all (a count of zero might indicate that a shopping cart is empty or that no records were retrieved from a database), or whether it has more items than expected (perhaps a customer has more than one address on file).

Sometimes knowing the number of items in an array can be helpful in order to control a loop through that array. In Listing 7-7, we use `count()` to print the total number of items in the `$animals` array.

```
<?php
$animals = ['cat', 'dog', 'giraffe', 'elephant'];

print count($animals);
```

Listing 7-7: Counting the number of elements in an array

We call the `count()` function, passing the name of the array we want it to count up, and print the result. Since `$animals` has four elements, this script should output the integer `4`.

NOTE *The `sizeof()` function is an alias of `count()`. If you see a script that uses `sizeof()`, know that it works the same way as `count()`.*

Another analytical array function is `array_is_list()`. PHP distinguishes between arrays that are lists and arrays that aren't. To be considered a *list*, an array of length n must have consecutively numbered keys from 0 to $n - 1$. The `array_is_list()` function takes in an array and returns `true` or `false` based on whether the array meets that definition. All the arrays discussed in this chapter qualify as lists, since they rely on PHP's default behavior of assigning keys sequentially from 0. In the next chapter, however, we'll explore arrays with non-integer keys as well as the `unset()` function, which can remove an element of an array with a given key, potentially breaking the consecutive chain of numeric keys and disqualifying an array as a list. Thus, `array_is_list()` could be useful for evaluating an array before passing it along to code that expects the array to be structured as a list.

The `array_key_last()` function returns the key for the last element of the given array. Assuming the array is a proper list with consecutively numbered keys, the return value of `array_key_last()` should be one less than the return value of `count()`. For example, calling `array_key_last($animals)` at the end of Listing 7-7 would return the integer `3`, since that's the key of the fourth (and final) element of the array.

Earlier I mentioned that trying to access an array key that doesn't exist triggers a warning. To avoid this, use the `isset()` function to test whether an array key exists before trying to access it. Listing 7-8 shows the function in action.

```
<?php
$animals = ['cat', 'dog', 'giraffe', 'elephant'];

❶ if (isset($animals[3])) {
    print "element 3 = $animals[3]\n";
} else {
    print "sorry - there is no element 3 in this array\n";
}
```

```
print "(popping last element [3])\n";
❷ array_pop($animals);

if (isset($animals[3])) {
    print "element 3 = $animals[3]\n";
} else {
    print "sorry - there is no element 3 in this array\n";
}
```

Listing 7-8: Using `isset()` to test the existence of an array key

First, we create our four-element `$animals` array. Then we use an `if...else` statement with `isset()` to access only the element with key `3` if that element exists (at this point, it should) ❶. We next use `array_pop()` to remove the last element from `$animals` (the one at key `3`) ❷. Then we repeat the same `if...else` statement. Now no element has key `3`, but since we're testing for the element with `isset()` before attempting to access it, we shouldn't get a warning. Take a look at the output of the script:

```
element 3 = elephant
(popping last element [3])
sorry - there is no element 3 in this array
```

The first line of the output indicates the first call to `isset()` returned `true`, triggering the `if` branch of the conditional. The last line shows the second `isset()` call returned `false`, triggering the `else` branch and saving us from trying to access a nonexistent array element.

Looping Through an Array

It's common to have to access the elements of an array, one at a time, and do something with each one. Assuming the array is a list, you can do this with a `for` loop that uses a counter variable as the key for the current array element. By starting the counter at `0` and incrementing it up to the length of the array, you can access each element in turn. Listing 7-9 uses a `for` loop to print each element of our `$animals` array.

```
<?php
$animals = ['cat', 'dog', 'giraffe', 'elephant'];

$numElements = count($animals);
for ($i = 0; $i < $numElements; $i++) {
  ❶ $animal = $animals[$i];
    print "$animal, ";
}
```

Listing 7-9: Using a `for` loop to loop through an array

We use `count()` to look up the length of the array, storing the result into the `$numElements` variable. Then we declare a `for` loop that increments counter `$i` from `0` up to but not including the value of `$numElements`. (We could

hardcode the stopping condition as `$i < 4`, but using a variable makes the code more flexible in case the length of the array changes.) In the loop statement group, we use `$animals[$i]` to retrieve the element whose key is the current value of loop variable `$i`, storing it in `$animal` ❶. Then we print out this `$animal` string, followed by a comma and a space. The output when we run this script in a terminal is as follows:

```
cat, dog, giraffe, elephant,
```

Each of the array elements is printed out in sequence. (Don't worry, we'll fix that final comma shortly.)

Using a foreach Loop

This `for` loop approach works, but cycling through the elements of an array is such a common task that PHP provides another type of loop, the `foreach` loop, to do it more efficiently. At the core of a `foreach` loop is the `foreach` (`$`*`array`* `as $`*`value`*) syntax; `$`*`array`* is the name of an array to loop over, and `$`*`value`* is a temporary variable that will be assigned the value of each element in the array, one at a time. Listing 7-10 shows an updated version of Listing 7-9, using a `foreach` loop rather than a `for` loop.

```
<?php
$animals = ['cat', 'dog', 'giraffe', 'elephant'];

foreach ($animals as $animal) {
    print "$animal, ";
}
```

Listing 7-10: Using a `foreach` loop to elegantly loop through an array

We declare this loop by using `foreach ($animals as $animal)`. Here, `$animal` is a temporary variable that takes on the value of each array element in turn, which we then print in the body of the loop. Notice that we no longer have to worry about determining the length of the array to set the loop's stopping condition, nor do we need to manually access each array element, as we did in the `for` loop version with `$animals[$i]`. The `foreach` loop retrieves each element automatically. The result is the same as the `for` loop version, but the `foreach` loop's syntax is much more elegant and concise.

The `foreach` loop has the added benefit that we don't need to care whether the provided array is a true list. With the `for` loop version, we're relying on the consecutive integer numbering of the array keys; if a key is missing, we'll get a warning when we try to access that key. By contrast, the `foreach` loop simply accesses each element in the array, no matter what the keys are.

Accessing Keys and Values

An alternative syntax for `foreach` loops allows you to access both the key and the value of each array element, instead of just the value. For this, declare

the loop in the format `foreach ($array as $key => $value)`. Here, `$array` is the array you want to loop through, `$key` is a temporary variable that will hold the current element's key, and `$value` is a temporary variable that will hold the current element's value. The `=>` operator connects a key to a value. We'll use it more extensively in Chapter 8, when we work with sophisticated arrays whose keys can be strings and other data types.

Gaining access to keys as well as values allows us to eliminate that pesky final comma from the output after the last element in the `$animals` array. Recall that the `array_key_last()` function returns the key of the last element in an array. By comparing the value from this function with the current key in the `foreach` loop, we can decide whether to print a comma after each element. Listing 7-11 shows how.

```
<?php
$animals = ['cat', 'dog', 'giraffe', 'elephant'];

foreach ($animals as $key => $animal) {
    print "$animal";
  ❶ if ($key != array_key_last($animals)) {
        print ", ";
    }
}
```

Listing 7-11: A revised foreach loop that accesses the key and value of each array element

We declare a `foreach` loop by using `foreach ($animals as $key => $animal)`. Each cycle through the loop, `$key` will be the key and `$animal` will be the value of the current array element. Inside the loop, we first print out the string in `$animal`. Then we use an `if` statement ❶ to also print a comma and a space if the current element's key is *not* equal to the last key of the array (identified with the `array_key_last()` function). This should produce the following output:

```
cat, dog, giraffe, elephant
```

We've successfully eliminated the comma after the last element in the array.

Imploding an Array

The code in Listing 7-11 is essentially printing a string containing the elements in an array with a separator (in this case, a comma and a space) between them. This is a common task, so PHP provides a built-in function called `implode()` to do it automatically, without the need for any kind of loop.

The function takes two arguments: a string to use as a separator, and an array to implode into a string. The separator goes *between* elements, not *after each* element, so the code won't place an extra separator after the last array element. Listing 7-12 shows an updated script that uses `implode()` rather than a `foreach` loop.

```
<?php
$animals = ['cat', 'dog', 'giraffe', 'elephant'];

print implode(', ', $animals);
```

Listing 7-12: Using `implode()` to convert an array into a string

Here we print the result of calling `implode()` on the `$animals` array. We use the string `', '` as a separator to put a comma and a space between the array elements. The output should be exactly the same as that of Listing 7-11, but `implode()` makes the code much more efficient to write.

The `implode()` function may have rendered our `foreach` loop unnecessary in this case, but don't let that fool you. A `foreach` loop is the right tool to use in plenty of scenarios. In general, when the code you want to apply to each element in an array is more sophisticated than simply printing out that element's value, a `foreach` loop is likely appropriate.

Functions with a Variable Number of Arguments

One important application for arrays is that you can use them to declare functions that accept a variable number of arguments. When we created custom functions in Chapter 5, we needed to know exactly how many arguments each function would take in so we could define the function with the corresponding number of parameters. This isn't always possible, however.

For example, say we want to declare a `sum()` function that takes in an unspecified quantity of numbers, adds them all up, and returns the result. We don't know whether the user will pass two numbers, three numbers, or more as arguments, so we can't create a separate parameter for each number. Instead, we use a single parameter to represent all the numbers, and we write an ellipsis, or three dots (`...`), before the parameter name. This syntax tells PHP to treat the parameter as an array and to fill it with all the arguments provided, however many there are.

Listing 7-13 shows how this approach works by declaring the `sum()` function just described. Remember that functions should be declared in a separate file from the code that calls them, so create a *my_functions.php* file containing the contents of this listing.

```
<?php
function sum(...$numbers): int
{
    $total = 0;

  ❶ foreach ($numbers as $number) {
        $total += $number;
    }

    return $total;
}
```

Listing 7-13: A function to take in a variable number of integer arguments and return their sum

We declare the `sum()` function, which returns an integer. It has a single parameter, `... $numbers`. Thanks to the ellipsis, any arguments the function receives will be assigned as elements to a local array called `$numbers`. Notice that we don't specify a data type for the parameter when using the ellipsis notation; we know the overall `$numbers` variable will be of the `array` type, although the individual elements of the array can be of any type. Inside the function body, we initialize `$total` to `0`. Then we use a `foreach` loop ❶ to cycle through the elements of the `$numbers` array, adding the value of each element to `$total`. Once the loop has finished, we return `$total`, which holds the sum of the arguments.

NOTE *Our `sum()` function doesn't include any logic to confirm that the elements in `$numbers` are actually numbers. A real-world function would need some form of data validation and would perhaps return `NULL` or indicate invalid data some other way if the arguments provided aren't all numbers. Also note that PHP already has a built-in `array_sum()` function that totals up the numbers in an array. We've implemented our own version for demonstration purposes.*

Listing 7-14 shows a *main.php* script to read in the `sum()` function declaration and test it out with a variable number of arguments.

```
<?php
require_once 'my_functions.php';

print sum(1, 2, 3) . "\n";
print sum(20, 40) . "\n";
print sum(1, 2, 3, 4, 5, 6, 7) . "\n";
```

Listing 7-14: A main script that calls `sum()` with different numbers of arguments

After reading in and executing *my_functions.php* with `require_once`, we make three calls to `sum()`, each with a different number of arguments, and print the results. The script produces this output:

```
6
60
28
```

The three printed sums have been correctly calculated. This indicates that our `sum()` function has successfully collected the variable number of arguments in an array.

Array Copies vs. Array References

Say you have a variable containing an array, and you assign it as the value of a second variable. In some languages, such as Python and JavaScript, the second variable would be assigned a *reference* to the original array. The two variables would then refer to the same array in the computer's memory, so a change to one variable would apply to the other variable as well. In PHP,

however, the default is to assign the second variable a *copy* of the array. Because the two variables have their own separate arrays, a change to one won't impact the other. Listing 7-15 returns to our `$animals` array to illustrate this point.

```
<?php
$animals = ['cat', 'dog', 'giraffe', 'elephant'];

$variable2 = $animals;
array_pop($variable2);
var_dump($animals);
```

Listing 7-15: Copying an array

We declare our usual `$animals` array, then assign `$animals` to `$variable2`. This creates a separate copy of the array in `$variable2`, so anything we do to one array should have no effect on the other. To prove it, we use `array_pop()` to remove the last element from the `$variable2` array, then print the original `$animals` array. Here's the result:

```
array(4) {
    [0]=>
    string(3) "cat"
    [1]=>
    string(3) "dog"
    [2]=>
    string(7) "giraffe"
    [3]=>
    string(8) "elephant"
}
```

All four animal strings are still present in the `$animals` array, even though we deleted the final element (`'elephant'`) from the `$variable2` array, so the variables indeed hold separate arrays.

If you want to assign the second variable a reference to the original array, as is customary in other languages, rather than a copy of the array, then use the reference operator (`&`) at the time of assignment. Listing 7-16 updates the code from Listing 7-15 to show the difference.

```
<?php
$animals = ['cat', 'dog', 'giraffe', 'elephant'];

$variable2 = &$animals;
array_pop($variable2);
var_dump($animals);
```

Listing 7-16: Using the reference operator when assigning an array

This time we prefix the `$animals` array with the reference operator (`&`) when assigning it to `$variable2`. This means a change to one variable will now apply to the other variable as well, since both refer to the same array in memory. The updated script results in this output:

```
array(3) {
    [0]=>
    string(3) "cat"
    [1]=>
    string(3) "dog"
    [2]=>
    string(7) "giraffe"
}
```

The output reveals that popping element `3` from the `$variable2` array also removed element `3` from the `$animals` array. This confirms that both `$variable2` and `$animals` refer to the same array in memory.

One of these approaches to array assignment isn't inherently better than the other; they're just different. Sometimes it's best to copy an array before manipulating it. For example, a web page might offer the user a chance to edit a shopping list, while providing a Cancel button to undo those edits. In this case, you'll want to work with a copy of the shopping list array until the changes are confirmed, since you may need to revert to the original array if the user clicks Cancel. Other times, it's preferable to have multiple variables referencing the same array in memory. Perhaps the code contains logic that chooses one of several arrays and so needs to set a variable to be a reference to the chosen array.

The key point to take away from this section is that PHP defaults to copying the array unless you use the reference operator (`&`). If you've learned a different programming language before PHP, or if you learn another language in the future, it's important to understand the difference between assignment of a copy and assignment of a reference, and to know which behavior the language uses by default.

Treating Strings as Arrays of Characters

In a way, you can think of a string as an array of individual characters. This can be useful since you may sometimes want to "navigate" through the string character by character for tasks such as encryption, spellchecking, and so on, just as you'd traverse the elements of an array.

As you saw in Chapter 3, the characters in a string are numbered from 0, just like the elements in a simple array. In fact, you can use the same square-bracket notation for accessing an array element to also access a specific character from a string. For example, if `$name` held the string `'Smith'`, `$name[0]` would return `'S'`, `$name[1]` would return `'m'`, and so on. Strings also support *negative* integer keys for counting characters backward from the end of the string: `$name[-1]` returns `'h'` (the last character), `$name[-2]` returns `'t'`, and so on.

NOTE *Unlike strings, arrays themselves don't interpret negative integer keys as counting backward from the end of the array. Instead, `$animals[-1]` would be interpreted as an element of the `$animals` array with an actual key of `-1`. While you* can *manually assign negative integers as keys to array elements, I personally can't remember ever needing to do so.*

Listing 7-17 shows an example of using array key syntax to access individual characters from a string.

```
<?php
$name = 'Smith';

$firstChar = $name[0];
$secondToLastChar = $name[-2];

print "first character = '$firstChar'\n";
print "second to last character = '$secondToLastChar'\n";
```

Listing 7-17: Using square-bracket notation to access string characters

We assign the string `'Smith'` to the `$name` variable. Next, we copy the string's first character (`$name[0]`) to the `$firstChar` variable and its second-to-last character (`$name[-2]`) to `$secondToLastChar`. We then print out messages with the values of these variables, producing the following output:

```
first character = 'S'
second to last character = 't'
```

Unlike with arrays, we can't pass a string to a `foreach` loop to cycle through all its characters. However, we *can* use PHP's built-in `str_split()` function to convert a string into an actual array of individual characters, then pass that array to a `foreach` loop, as shown in Listing 7-18.

```
<?php
$name = 'Smith';

$characters = str_split($name);
foreach ($characters as $key => $character) {
    print "character with key $key = '$character'\n";
}
```

Listing 7-18: Using `str_split()` and `foreach` to loop over the characters in a string

We pass the same `$name` string to `str_split()`. By default, this function breaks the string into individual characters, assigns them as elements of an array, and returns the result, which we store in the `$characters` variable. We then use a `foreach` loop to access each key and value in the array version of the string and print them out. Here's the result:

```
character with key 0 = 'S'
character with key 1 = 'm'
character with key 2 = 'i'
character with key 3 = 't'
character with key 4 = 'h'
```

The output shows that we've successfully looped through the characters from the original string after first converting the string to an array with `str_split()`.

The `str_split()` function has an optional second argument controlling the number of characters for each string element in the resulting array. The argument defaults to `1`, splitting the string into individual characters, but if we'd called `str_split($name, 2)`, for example, then the resulting array would contain two-character strings: `['Sm', 'it', 'h']`.

Other Array Functions

We've discussed some built-in functions for working with arrays in this chapter, but PHP has many more. Other useful functions that apply to arrays include the following:

`sort()` Sorts an array's values in ascending order (alphabetical for string values, numerical order for number values)

`usort()` Sorts the values into a custom order based on a user-defined function

`array_flip()` Swaps the keys and values for each array element

`array_slice()` Returns a new array containing a subsequence of elements from an existing array

`array_walk()` Calls a user-defined function on each element of an array

`array_map()` Calls a user-defined function on each element of an array and returns a new array of the results

`array_rand()` Returns random keys from an array

For a full list of array functions, see the PHP documentation at *https://www.php.net/manual/en/ref.array.php*.

Summary

In this chapter, we've begun exploring arrays, a compound data type that stores multiple values under a single variable name, with each value having its own identifying key. For now, we've focused on simple arrays, whose keys are integers. The chapter introduced various techniques for adding, subtracting, and accessing array elements, as well as functions for obtaining information about an array, such as `count()` and `isset()`. You also learned how to work with each array element in sequence by using a `foreach` loop. In some cases, PHP provides built-in functions for handling common array tasks, such as the `implode()` function that joins all the elements of an array into a single string. These functions sometimes allow you to replace complete loops and conditional statements with a single function call.

Exercises

1. Create a `$colors` array containing the string names of five colors. Print a random color from the array.

   Hint: You can get a valid random key by calling `array_rand($colors)`.

2. Write a `foreach` loop to print each of the colors from your array in Exercise 1 on a new line, in the following form:

```
color 0 = blue
color 1 = red
...
```

3. Use `array_pop()` to print the last element of your array of colors from Exercise 1. Then use `var_dump()` to show that this item has been removed from the array.
4. Create an array containing integer ages `23`, `31`, and `55`. Use built-in functions to calculate and print out the number of items in the array and their average.