Chapter 7. Arrays

Arrays are ordered maps—constructs that associate specific values to easily identified keys. These maps are effective ways to build both simple lists and more complex collections of objects. They're also easy to manipulate—adding or removing items from an array is straightforward and supported through multiple functional interfaces.

Types of Arrays

There are two forms of arrays in PHP—numeric and associative. When you define an array without explicitly setting keys, PHP will internally assign an integer index to each member of the array. Arrays are indexed starting with 0 and increase by steps of 1 automatically.

Associative arrays can have keys of either strings or integers, but generally use strings. String keys are effective ways to "look up" a particular value stored in an array.

Arrays are implemented internally as hash tables, allowing for effective direct associations between keys and values. For example:

```
$colors = [];
$colors['apple']  = 'red';
$colors['pear']   = 'green';
$colors['banana'] = 'yellow';

$numbers = [22, 15, 42, 105];

echo $colors['pear']; // green
echo $numbers[2]; // 42
```

Unlike simpler hash tables, though, PHP arrays also implement an iterable interface allowing you to loop through all of their elements one at a time. Iteration is fairly obvious when keys are numeric, but even with associative arrays, the elements have a fixed order because they're stored in memory.

Recipe 7.3 details different ways to act on each element in both types of arrays.

In many circumstances, you might also be met with objects or classes that look and feel like an array but are not actually arrays. In fact, any object that implements the `ArrayAccess` interface can be used as an array.[1] These more advanced implementations push the limits of what is possible with arrays beyond mere lists and hash tables.

Syntax

PHP supports two different syntaxes for defining arrays. Those who have worked in PHP for some time will recognize the `array()` construct that allows for the literal definition of an array at runtime as follows:

```
$movies = array('Fahrenheit 451', 'Without Remorse', 'E
```

An alternative and terser syntax is to use square brackets to define the array. The preceding example could be rewritten as follows with the same behavior:

```
$movies = ['Fahrenheit 451', 'Without Remorse', 'Black
```

Both formats can be used to create nested arrays (where an array contains another array) and can be used interchangeably as follows:

```
$array = array(1, 2, array(3, 4), [5, 6]);
```

Though mixing and matching syntaxes as in the preceding example is possible, it is highly encouraged to remain consistent within your application and to use one form or the other—not both. All of the examples in this chapter will use the short array syntax (square brackets).

All arrays in PHP map from keys to values. In the preceding examples, the arrays merely specified values and let PHP assign keys automatically. These are considered *numeric* arrays as the keys will be integers, starting at 0. More complex arrays, like the nested construct illustrated in Example 7-1, assign

both values and keys. This is done by mapping from a key to a value with a two-character arrow operator (`=>`).

Example 7-1. Associative array with nested values

```
$array = array(
    'a' => 'A',
    'b' => ['b', 'B'],
    'c' => array('c', ['c', 'K'])
);
```

While not a syntactic requirement, many coding environments and integrated development environments (IDEs) will automatically align the arrow operators in multiline array literals. This makes the code easier to read and is a standard adopted by this book as well.

The recipes that follow illustrate various ways developers can work with arrays—both numeric and associative—to accomplish common tasks in PHP.

7.1 Associating Multiple Elements per Key in an Array

Problem

You want to associate multiple items with a single array key.

Solution

Make each array value an array on its own—for example:

```
$cars = [
    'fast'     => ['ferrari', 'lamborghini'],
    'slow'     => ['honda', 'toyota'],
    'electric' => ['rivian', 'tesla'],
    'big'      => ['hummer']
];
```

Discussion

PHP places no requirement on the type of data used for a value in an array. However, keys are required to be either strings or integers. In addition, it is a hard requirement that every key in an array be unique. If you attempt to set multiple values for the same key, you will overwrite existing data, as shown in Example 7-2.

Example 7-2. Overwriting array data by assignment

```
$basket = [];

$basket['color']    = 'brown';
$basket['size']     = 'large';
$basket['contents'] = 'apple';
$basket['contents'] = 'orange';
$basket['contents'] = 'pineapple';

print_r($basket);

// Array
// (
//     [color] => brown
//     [size] => large
//     [contents] => pineapple
// )
```

As PHP only allows one value per unique key in an array, writing further data to that key overwrites its value in the same way that you might reassign the value of a variable in your application. If you do need to store multiple values in one key, use a nested array.

The Solution example illustrates how *every* key could point to its own array. However, PHP does not require this to be true—all but one key could point to a scalar and just the key that needs multiple items could point to an array. In Example 7-3, you'll use a nested array to store multiple items rather than accidentally overwriting a single value stored in a specific key.

Example 7-3. Writing an array to a key

```
$basket = [];
```

```
$basket['color']    = 'brown';
$basket['size']     = 'large';
$basket['contents'] = [];
$basket['contents'][] = 'apple';
$basket['contents'][] = 'orange';
$basket['contents'][] = 'pineapple';

print_r($basket);

// Array
// (
//    [color] => brown
//    [size] => large
//    [contents] => Array
//        (
//            [0] => apple
//            [1] => orange
//            [2] => pineapple
//        )
// )

echo $basket['contents'][2]; // pineapple
```

To leverage the elements of a nested array, you loop over them just as you would the parent array. For example, if you wanted to print all of the data stored in the `$basket` array from Example 7-3, you would need two loops, as in Example 7-4.

Example 7-4. Accessing array data in a loop

```
foreach ($basket as $key => $value) {
                                ❶

    if (is_array($value)) {
                                ❷

        echo "{$key} => [" . PHP_EOL;

        foreach ($value as $item) {
                                ❸

            echo "\t{$item}" . PHP_EOL;
        }

        echo ']' . PHP_EOL;
    } else {
```

```
            echo "{$key}: $value" . PHP_EOL;
        }
    }

    // color: brown
    // size: large
    // contents => [
    //     apple
    //     orange
    //     pineapple
    // ]
```

❶ The parent array is associative, and you need both its keys and values.

❷ You use one branch of logic for nested arrays, another for scalars.

❸ Since you know the nested array is numeric, ignore the keys and iterate over only the values.

See Also

Recipe 7.3 for further examples of iterating through arrays.

7.2 Initializing an Array with a Range of Numbers

Problem

You want to build an array of consecutive integers.

Solution

Use the `range()` function as follows:

```
$array = range(1, 10);
print_r($array);

// Array
// (
//     [0] => 1
//     [1] => 2
//     [2] => 3
```

```
//     [3] => 4
//     [4] => 5
//     [5] => 6
//     [6] => 7
//     [7] => 8
//     [8] => 9
//     [9] => 10
// )
```

Discussion

PHP's `range()` function automatically iterates over a given sequence, assigning a value to a key based on the definition of that sequence. By default, and as illustrated in the Solution example, the function steps through sequences one at a time. But this isn't the limit of the function's behavior—passing a third parameter to the function will change its step size.

You could iterate over all even integers from 2 to 100 as follows:

```
$array = range(2, 100, 2);
```

Likewise, you could iterate over all *odd* integers from 1 to 100 by changing the starting point of the sequence to 1. For example:

```
$array = range(1, 100, 2);
```

The start and end parameters of `range()` (the first two parameters, respectively) can be integers, floating-point numbers, or even strings. This flexibility allows you to do some pretty amazing things in code. For example, rather than counting natural numbers (integers), you could produce an array of floating-point numbers as follows:

```
$array = range(1, 5, 0.25);
```

When passing string characters to `range()`, PHP will begin enumerating ASCII characters. You can leverage this functionality to quickly build an array representative of the English alphabet, as shown in Example 7-5.

NOTE

PHP will internally use any and all printable ASCII characters, based on their decimal representation, to complete a request to `range()`. This is an efficient way to enumerate printable characters, but you need to keep in mind where special characters such as `=`, `?`, and `)` fall within the ASCII table, particularly if your program is expecting alphanumeric values in the array.

Example 7-5. Creating an array of alphabetical characters

```
$uppers = range('A', 'Z');
```
❶

```
$lowers = range('a', 'z');
```
❷

```
$special = range('!', ')');
```
❸

❶ Returns all uppercase characters from `A` through `Z`

❷ Returns all lowercase characters from `a` through `z`

❸ Returns an array of special characters: `[!, ", #, $, %, &, ', (, )]`

See Also

PHP documentation on `range()`.

7.3 Iterating Through Items in an Array

Problem

You want to perform an action on every element in an array.

Solution

For numeric arrays, use `foreach` as follows:

```
foreach ($array as $value) {
    // Act on each $value
}
```

For associative arrays, use `foreach()` with optional keys as follows:

```
foreach ($array as $key => $value) {
    // Act on each $value and/or $key
}
```

Discussion

PHP has the concept of *iterable objects* and, internally, that's precisely what an array is. Other data structures can also implement iterable behavior,[2] but *any* iterable expression can be provided to `foreach` and will return the items it contains one at a time in a loop.

WARNING

PHP does not implicitly unset the variable used within a `foreach` loop when you exit the loop. You can still explicitly reference the *last* value stored in `$value` in the Solution examples in the program outside the loop!

The most important thing to remember, though, is that `foreach` is a *language construct*, not a function. As a construct, it acts on a given expression and applies the defined loop over every item within that expression. By default, that loop does not modify the contents of an array. If you want to make the values of an array mutable, you must pass them into the loop by reference by prefixing the variable name with an `&` character as follows:

```
$array = [1, 2, 3];

foreach ($array as &$value) {
    $value += 1;
}

print_r($array); // 2, 3, 4
```

WARNING

Versions of PHP prior to 8.0 supported an `each()` function that would maintain an array cursor and return the current key/value pair of the array before advancing that cursor. This function was deprecated in PHP 7.2 and fully removed as of the 8.0 release, but you will likely find legacy examples of its use in books and online. Upgrade any occurrences of `each()` to an implementation of `foreach` to ensure forward compatibility of your code.

An alternative approach to using a `foreach` loop is to create an explicit `for` loop over the keys of the array. Numeric arrays are easiest as their keys are already incrementing integers starting at 0. Iterating over a numeric array is relatively straightforward as follows:

```
$array = ['red', 'green', 'blue'];

$arrayLength = count($array);
for ($i = 0; $i < $array_length; $i++) {
    echo $array[$i] . PHP_EOL;
}
```

TIP

While it's possible to place a call to `count()` to identify the upper bounds of a `for` loop directly within the expression, it's better to store the length of an array outside the expression itself. Otherwise, your `count()` will be reinvoked on every iteration of the loop to check that you're still in bounds. For small arrays, this won't matter; as you start working with larger collections, though, the performance drain of repeated `count()` checks will become problematic.

Iterating over an associative array with a `for` loop is a tiny bit different. Instead of iterating over the elements of the array directly, you'll want to iterate over the keys of the array directly. Then use each key to extract the corresponding value from the array as follows:

```
$array = [
    'os'   => 'linux',
    'mfr'  => 'system76',
```

```
    'name' => 'thelio',
];

$keys = array_keys($array);
$arrayLength = count($keys);
for ($i = 0; $i < $arrayLength; $i++) {
    $key = $keys[$i];
    $value = $array[$key];

    echo "{$key} => {$value}" . PHP_EOL;
}
```

See Also

PHP documentation on the `foreach` and `for` language constructs.

7.4 Deleting Elements from Associative and Numeric Arrays

Problem

You want to remove one or more elements from an array.

Solution

Delete an element by targeting its key or numeric index directly with `unset()`:

```
unset($array['key']);

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

Delete more than one element at a time by passing multiple keys or indexes into `unset()` as follows:

```
unset($array['first'], $array['second']);

unset($array[3], $array[4], $array[5]);
```

Discussion

In PHP, `unset()` actually destroys any reference to the memory containing the specified variable. In the context of this Solution, that variable is an element of an array, so unsetting it removes that element from the array itself. In an associative array, this takes the form of deleting the specified key and the value it represented.

In a numeric array, `unset()` does far more. It both removes the specified element and effectively converts the numeric array into an associative array with integer keys. On the one hand, this is likely the behavior you wanted in the first place, as demonstrated in Example 7-6.

Example 7-6. Unsetting elements in a numeric array

```
$array = range('a', 'z');

echo count($array) . PHP_EOL;
                              ❶

echo $array[12] . PHP_EOL;
                              ❷

echo $array[25] . PHP_EOL;
                              ❸


unset($array[22]);
echo count($array) . PHP_EOL;
                              ❹

echo $array[12] . PHP_EOL;
                              ❺

echo $array[25] . PHP_EOL;
                              ❻
```

❶ The array by default represents all English characters from `a` through `z`, so this line prints `26`.

❷ The 13th letter in the alphabet is `m`. (Remember that arrays start at index `0`.)

❸ The 26th letter in the alphabet is `z`.

❹ With the element removed, the array has decreased in size to `25`!

❺ The 13th letter in the alphabet is *still* `m`.

The 15th letter in the alphabet is still `o` .

The 26th letter in the alphabet is *still* `z` . Further, this index is still valid, as removing an element doesn't re-index the array.

You can typically ignore the indexes of numeric arrays because they're set by PHP automatically. This makes the behavior of `unset()` implicitly converting these indexes into numeric keys somewhat surprising. With a numeric array, attempting to access an index greater than the length of the array results in an error. Once you've used `unset()` with the array and decreased its size, however, you will often end up with an array that has numeric keys greater than the size of the array, as was illustrated in Example 7-6.

If you want to return to the world of numeric arrays after removing an element, you can re-index the array entirely. PHP's `array_values()` function returns a new, numerically indexed array that contains only the values of the specified array. For example:

```
$array = ['first', 'second', 'third', 'fourth'];

unset($array[2]);

$array = array_values($array);
```

The default array has numeric indexes: `[0 => first, 1 => second, 2 => third, 3 => fourth]` .

Unsetting an element removes it from the array but leaves the indexes (keys) unchanged: `[0 => first, 1 => second, 3 => fourth]` .

The call to `array_values()` gives you a *new* array with brand-new, properly incrementing numeric indexes: `[0 => first, 1 => second, 2 => fourth]` .

An additional option for removing elements from an array is to use the `array_splice()` function.[3] This function will remove a portion of an array and replace it with something else.[4] Consider Example 7-7, where

`array_splice()` is used to replace elements of an array with *nothing*, thus removing them.

Example 7-7. Removing elements of an array with `array_splice()`

```
$celestials = [
    'sun',
    'mercury',
    'venus',
    'earth',
    'mars',
    'asteroid belt',
    'jupiter',
    'saturn',
    'uranus',
    'neptune',
    'pluto',
    'voyagers 1 & 2',
];

array_splice($celestials, 0, 1);
                              ❶

array_splice($celestials, 4, 1);
                              ❷

array_splice($celestials, 8);
                              ❸


print_r($celestials);

// Array
// (
//     [0] => mercury
//     [1] => venus
//     [2] => earth
//     [3] => mars
//     [4] => jupiter
//     [5] => saturn
//     [6] => uranus
//     [7] => neptune
// )
```

❶ First, remove the sun to clean up a list of planets in the solar system.

❷

Once the sun is removed, the indexes of all objects shift. You still want to remove the asteroid belt from the list, so use its newly shifted index.

3. Finally, truncate the array by removing everything from Pluto to the end of the array.

Unlike `unset()`, the modified array created by `array_splice()` does *not* retain the numeric indexes/keys in numeric arrays! This might be a good way to avoid needing an extra call to `array_values()` after removing an item from an array. It's also an effective way to remove *continuous* elements from a numerically indexed array without needing to explicitly specify each element.

See Also

Documentation on `unset()`, `array_splice()`, and `array_values()`.

7.5 Changing the Size of an Array

Problem

You want to increase or decrease the size of an array.

Solution

Add elements to the end of the array by using `array_push()`:

```
$array = ['apple', 'banana', 'coconut'];
array_push($array, 'grape');

print_r($array);

// Array
// (
//     [0] => apple
//     [1] => banana
//     [2] => coconut
//     [3] => grape
// )
```

Remove elements from an array by using `array_splice()` :

```
$array = ['apple', 'banana', 'coconut', 'grape'];
array_splice($array, 1, 2);

print_r($array);

// Array
// (
//     [0] => apple
//     [1] => grape
// )
```

Discussion

Unlike many other languages, PHP doesn't require you to declare the size of an array. Arrays are dynamic—you can add or remove data from them whenever you want with no real downside.

The first Solution example merely adds a single element to the end of an array. While this approach is straightforward, it's not the most efficient. Instead, you can push an individual item into an array *directly* as follows:

```
$array = ['apple', 'banana', 'coconut'];
$array[] = 'grape';

print_r($array);

// Array
// (
//     [0] => apple
//     [1] => banana
//     [2] => coconut
//     [3] => grape
// )
```

The key difference between the preceding example and the one documented in the Solution is that of a function call. In PHP, function calls have more overhead than language constructs (like assignment operators). The preceding example is slightly more efficient, but only if it's used several times in an application.

If you are instead adding *multiple* items to the end of an array, the `array_push()` function will be more efficient. It accepts and appends many items at once, thus avoiding multiple assignments. Example 7-8 illustrates the difference between approaches.

Example 7-8. Appending multiple elements with `array_push()` versus assignment

```
$first = ['apple'];
array_push($first, 'banana', 'coconut', 'grape');

$second = ['apple'];
$second[] = 'banana';
$second[] = 'coconut';
$second[] = 'grape';

echo 'The arrays are ' . ($first === $second ? 'equal'

// The arrays are equal
```

If, rather than appending elements, you want to *prepend* them, you would use `array_unshift()` to place the specified items at the beginning of the array as follows:

```
$array = ['grape'];
array_unshift($array, 'apple', 'banana', 'coconut');

print_r($array);

// Array
// (
//     [0] => apple
//     [1] => banana
//     [2] => coconut
//     [3] => grape
// )
```

NOTE

PHP retains the order of elements passed to `array_unshift()` when prepending them to the target array. The first parameter will become the first element, the second the second, and so on until you reach the array's *original* first element.

Remember, arrays in PHP do not have a set size and can easily be manipulated in different ways. All of the preceding functional examples (`array_push()` , `array_splice()` , and `array_unshift()`) work well on numeric arrays and *do not change the order or structure* of their numerical indexes. You could just as easily add an element to the end of a numeric array by referencing a new index directly. For example:

```
$array = ['apple', 'banana', 'coconut'];
$array[3] = 'grape';
```

So long as the index your code references is continuous with the rest of the array, the preceding example will work flawlessly. If, however, your count is off and you introduce a gap in the index, you have effectively converted your numeric array to an associative one, just with numeric keys.

While all of the functions used in this recipe will work with associative arrays, they work primarily against numeric keys and will result in strange behavior when used against non-numeric ones. It would be wise to use these functions *only* with numeric arrays and to manipulate the sizes of associative arrays directly based on their keys.

See Also

Documentation on `array_push()` , `array_splice()` , and `array_unshift()` .

7.6 Appending One Array to Another

Problem

You want to combine two arrays into a single, new array.

Solution

Use `array_merge()` as follows:

```
$first = ['a', 'b', 'c'];
$second = ['x', 'y', 'z'];

$merged = array_merge($first, $second);
```

In addition, you can also leverage the spread operator (…) to combine arrays directly. Rather than a call to `array_merge()` , the preceding example then becomes this:

```
$merged = [...$first, ...$second];
```

The spread operator works for both numeric and associative arrays.

Discussion

PHP's `array_merge()` function is an obvious way to combine two arrays into one. It does, however, have slightly different behavior for numeric versus associative arrays.

WARNING

Any discussion of merging arrays will inevitably use the term *combine*. Note that `array_combine()` is itself a function in PHP. However, it doesn't merge two arrays as shown in this recipe. Instead, it creates a new array by using the two specified arrays —the first for the *keys* and the second for the *values* of the new array. It's a useful function but is not something you can use for merging two arrays.

For numeric arrays (like those in the Solution example), all elements of the second array are appended to those of the first array. The function ignores the indexes of both, and the newly produced array has continuous indexes (starting from `0`) as if you'd built it directly.

For associative arrays, the keys (and values) of the second array are added to those of the first. If the two arrays have the same keys, the values of the

second array will overwrite those of the first. Example 7-9 illustrates how the data in one array overwrites that of the other.

Example 7-9. Overwriting associative array data with `array_merge()`

```
$first = [
    'title'  => 'Practical Handbook',
    'author' => 'Bob Mills',
    'year'   => 2018
];
$second = [
    'year'   => 2023,
    'region' => 'United States'
];

$merged = array_merge($first, $second);
print_r($merged);

// Array
// (
//     [title] => Practical Handbook
//     [author] => Bob Mills
//     [year] => 2023
//     [region] => United States
// )
```

There might be cases where you want to retain the data held in duplicate keys when you merge two or more arrays. In those circumstances, use `array_merge_recursive()` . Unlike the preceding example, this function will create an array containing the data defined in duplicate keys rather than overwriting one value with another. Example 7-10 rewrites the preceding example to illustrate how this happens.

Example 7-10. Merging arrays with duplicate keys

```
$first = [
    'title'  => 'Practical Handbook',
    'author' => 'Bob Mills',
    'year'   => 2018
];
$second = [
    'year'   => 2023,
    'region' => 'United States'
```

```
];

$merged = array_merge_recursive($first, $second);
print_r($merged);

// Array
// (
//     [title] => Practical Handbook
//     [author] => Bob Mills
//     [year] => Array
//         (
//             [0] => 2018
//             [1] => 2023
//         )
//
//     [region] => United States
// )
```

While the preceding examples combine only two arrays, there is no upper limit to the number of arrays you can merge with either `array_merge()` or `array_merge_recursive()`. Keep in mind how duplicate keys are handled by both functions as you begin merging more than two arrays at a time to avoid potentially losing data.

A third and final way to combine two arrays into one is with the literal addition operator: `+`. On paper, this has the appearance of adding two arrays together. What it really does is add any new key from the second array to the keys of the first. Unlike `array_merge()`, this operation will not overwrite data. If the second array has keys that duplicate any in the first array, those keys are ignored, and the data from the first array is used.

This operator also works *explicitly* with array keys, meaning it's not a good fit for numeric arrays. Two same-sized numeric arrays, when treated like associative arrays, will have the exact same keys because they have the same indexes. This means the second array's data will be ignored entirely!

See Also

Documentation for `array_merge()` and `array_merge_recursive()`.

7.7 Creating an Array from a Fragment of an Existing Array

Problem

You want to select a subsection of an existing array and use it independently.

Solution

Use `array_slice()` to select a sequence of elements from an existing array as follows:

```
$array = range('A', 'Z');
$slice = array_slice($array, 7, 4);

print_r($slice);

// Array
// (
//     [0] => H
//     [1] => I
//     [2] => J
//     [3] => K
// )
```

Discussion

The `array_slice()` function quickly extracts a continuous sequence of items from the given array based on a defined offset (position within the array) and length of elements to retrieve. Unlike `array_splice()`, it copies the sequence of items from the array, leaving the original array unchanged.

It's important to understand the full function signature to appreciate the power of this function:

```
array_slice(
    array $array,
    int   $offset,
```

```
    ?$int $length = null,
    $bool $preserve_keys = false
): array
```

Only the first two parameters—the target array and the initial offset—are required. If the offset is positive (or `0`), the new sequence will start at that position from the beginning of the array. If the offset is negative, the sequence will start that many positions back from the *end* of the array.

NOTE

The array offset is explicitly referencing the *position* within an array, not in terms of keys or indexes. The `array_slice()` function works on associative arrays as easily as it does on numeric arrays because it uses the relative positions of elements in the array to define a new sequence and ignores the array's actual keys.

When you define the optional `$length` argument, this defines the maximum number of items in the new sequence. Note that the new sequence is limited by the number of items in the original array, so if the length overruns the end of the array, your sequence will be shorter than you expected. Example 7-11 presents a quick example of this behavior.

Example 7-11. Using `array_slice()` with a too-short array

```
$array = range('a', 'e');
$newArray = array_slice($array, 4, 100);

print_r($newArray);

// Array
// (
//     [0] => e
// )
```

If the length specified is *negative*, then the sequence will stop that many elements away from the end of the target array. If the length is not specified (or is `null`), then the sequence will include everything from the original offset through the end of the target array.

The final parameter, `$preserve_keys`, tells PHP whether to reset the integer indexes of the slice of the array. By default, PHP will return a newly

indexed array with integer keys starting at `0`. Example 7-12 shows how the behavior of the function differs based on this parameter.

NOTE

The `array_slice()` function will always preserve string keys in an associative array regardless of the value of `$preserve_keys`.

Example 7-12. Key preservation behavior in `array_slice()`

```
$array = range('a', 'e');

$standard = array_slice($array, 1, 2);
print_r($standard);

// Array
// (
//     [0] => b
//     [1] => c
// )

$preserved = array_slice($array, 1, 2, true);
print_r($preserved);

// Array
// (
//     [1] => b
//     [2] => c
// )
```

Remember, numeric arrays in PHP can be thought of as associative arrays with integer keys that start at `0` and increment consecutively. With that in mind, it's easy to see how `array_slice()` behaves on associative arrays with both string and integer keys—it operates based on position rather than key, as shown in Example 7-13.

Example 7-13. Using `array_slice()` on an array with mixed keys

```
$array = ['a' => 'apple', 'b' => 'banana', 25 => 'cola'
print_r(array_slice($array, 0, 3));
```

```
// Array
// (
//     [a] => apple
//     [b] => banana
//     [0] => cola
// )

print_r(array_slice($array, 0, 3, true));

// Array
// (
//     [a] => apple
//     [b] => banana
//     [25] => cola
// )
```

In Recipe 7.4, you were introduced to `array_splice()` for deleting a sequence of elements from an array. Conveniently, this function uses a method signature similar to that of `array_slice()`:

```
array_splice(
    array &$array,
    int   $offset,
    ?int  $length = null,
    mixed $replacement = []
): array
```

The key difference between these functions is that one modifies the source array whereas the other does not. You might use `array_slice()` to work on a subset of a larger sequence in isolation or instead to fully separate two sequences from one another. In either case, the functions exhibit similar behavior and use cases.

See Also

Documentation on `array_slice()` and `array_splice()`.

7.8 Converting Between Arrays and

Strings

Problem

You want to convert a string into an array or combine the elements of an array into a string.

Solution

Use `str_split()` to convert a string to an array:

```
$string = 'To be or not to be';
$array = str_split($string);
```

Use `join()` to combine the elements of an array into a string:

```
$array = ['H', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r',
$string = join('', $array);
```

Discussion

The `str_split()` function is a powerful way to convert any string of characters into an array of like-sized chunks. By default, it will break the string into one-character chunks, but you can just as easily break a string into any number of characters. The last chunk in the sequence is only guaranteed to be *up to* the specified length. For example, Example 7-14 attempts to break a string down into five-character chunks, but note that the last chunk is fewer than five characters in length.

Example 7-14. Using `str_split()` with arbitrary chunk sizes

```
$string = 'To be or not to be';
$array = str_split($string, 5);
var_dump($array);

// array(4) {
//   [0]=>
//   string(5) "To be"
```

```
//   [1]=>
//   string(5) " or n"
//   [2]=>
//   string(5) "ot to"
//   [3]=>
//   string(3) " be"
// }
```

WARNING

Remember that `str_split()` works on bytes. When you're dealing with multibyte encoded strings, you will need to use `mb_str_split()` instead.

In some cases, you might want to split a string into separate words rather than individual characters. PHP's `explode()` function allows you to specify the separator on which to split things. This is handy for splitting a sentence into an array of its component words, as demonstrated by Example 7-15.

Example 7-15. Splitting a string into an array of words

```
$string = 'To be or not to be';
$words = explode(' ', $string);

print_r($words);

// Array
// (
//     [0] => To
//     [1] => be
//     [2] => or
//     [3] => not
//     [4] => to
//     [5] => be
// )
```

NOTE

While `explode()` appears to function similarly to `str_split()`, it cannot explode a string with an empty delimiter (the first parameter to the function). If you try to pass an empty string, you will be met with a `ValueError`. If you want to work with an array of characters, stick with `str_split()`.

Combining an array of strings into a single string requires the use of the `join()` function, which itself is merely an alias of `implode()`. That said, it's far more powerful than just being the inverse of `str_split()`, as you can optionally define a separator to be placed between newly concatenated code chunks.

The separator is optional, but the long legacy of `implode()` in PHP has led to two somewhat unintuitive function signatures as follows:

```
implode(string $separator, array $array): string

implode(array $array): string
```

If you want to merely combine an array of characters into a string, you can do so with the equivalent methods shown in Example 7-16.

Example 7-16. Creating a string from an array of characters

```
$array = ['H', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r',

$option1 = implode($array);

$option2 = implode('', $array);

echo 'The two are ' . ($option1 === $option2 ? 'identic

// The two are identical
```

Because you can explicitly specify the separator—the glue used to join each chunk of text—there are few limits to what `implode()` allows you to do. Assume your array is a list of words rather than a list of characters. You can use `implode()` to link them together as a printable and comma-delimited list, as in the following example:

```
$fruit = ['apple', 'orange', 'pear', 'peach'];

echo implode(', ', $fruit);

// apple, orange, pear, peach
```

See Also

Documentation on `implode()`, `explode()`, and `str_split()`.

7.9 Reversing an Array

Problem

You want to reverse the order of elements in an array.

Solution

Use `array_reverse()` as follows:

```
$array = ['five', 'four', 'three', 'two', 'one', 'zero'

$reversed = array_reverse($array);
```

Discussion

The `array_reverse()` function creates a new array where each element is in the reverse order of the input array. By default, this function does not preserve numeric keys from the source array but instead re-indexes each element. Non-numeric keys (in associative arrays) are left unchanged by this re-indexing; however, their order is still reversed as expected. Example 7-17 demonstrates how associative arrays are reordered by `array_reverse()`.

Example 7-17. Reversing associative arrays

```
$array = ['a' => 'A', 'b' => 'B', 'c' => 'C'];
$reversed = array_reverse($array);

print_r($reversed);

// Array
// (
//     [c] => C
//     [b] => B
```

```
//     [a] => A
// )
```

Since associative arrays can have numeric keys to begin with, the re-indexing behavior might produce unexpected results. Thankfully, it can be disabled by passing an optional Boolean parameter as the second argument when reversing an array. Example 7-18 shows how this indexing behavior impacts such arrays (and how it can be disabled).

Example 7-18. Reversing an associative array with numeric keys

```
$array = ['a' => 'A', 'b' => 'B', 42 => 'C', 'd' => 'D'
print_r(array_reverse($array));
```
❶

```
// Array
// (
//     [d] => D
//     [0] => C
//     [b] => B
//     [a] => A
// )

print_r(array_reverse($array, true));
```
❷

```
// Array
// (
//     [d] => D
//     [42] => C
//     [b] => B
//     [a] => A
// )
```

❶ The default value of the second parameter is `false`, which means numeric keys will not be preserved after the array is reversed.

❷ Passing `true` as a second parameter will still allow the array to reverse but will retain numeric keys in the new array.

See Also

Documentation on `array_reverse()` .

7.10 Sorting an Array

Problem

You want to sort the elements of an array.

Solution

To sort items based on default comparison rules in PHP, use `sort()` as follows:

```
$states = ['Oregon', 'California', 'Alaska', 'Washingtc
sort($states);
```

Discussion

PHP's native sorting system is built atop Quicksort, a common and relatively fast sorting algorithm. By default, it uses rules defined by PHP's comparison operators to determine the order of each element in the array.[5] You can, however, sort with different rules by passing a flag as the optional second parameter of `sort()` . Available sorting flags are described in Table 7-1.

Table 7-1. Sorting type flags

Flag	Description
`SORT_REGULAR`	Compare items normally by using default comparison operations
`SORT_NUMERIC`	Compare items numerically
`SORT_STRING`	Compare items as strings

Flag	Description
`SORT_LOCALE_STRING`	Compare items as strings by using the current system locale
`SORT_NATURAL`	Compare items by using “natural ordering”
`SORT_FLAG_CASE`	Combine with `SORT_STRING` or `SORT_NATURAL` by using a bitwise OR operator to compare strings without case-sensitivity

Sorting type flags are useful when the default sorting comparisons produce a sorted array that makes no sense. For example, sorting an array of integers as if they were strings would sort things incorrectly. Using the `SORT_NUMERIC` flag will ensure that integers are sorted in the correct order. Example 7-19 demonstrates how the two sorting types differ.

Example 7-19. Sorting integers with a regular versus numeric sort type

```
$numbers = [1, 10, 100, 5, 50, 500];
sort($numbers, SORT_STRING);
print_r($numbers);

// Array
// (
//     [0] => 1
//     [1] => 10
//     [2] => 100
//     [3] => 5
//     [4] => 50
//     [5] => 500
// )

sort($numbers, SORT_NUMERIC);
print_r($numbers);

// Array
// (
//     [0] => 1
//     [1] => 5
//     [2] => 10
```

```
//     [3] => 50
//     [4] => 100
//     [5] => 500
// )
```

The `sort()` function ignores array keys and indexes and sorts the elements of the array purely by their values. Thus, attempting to use `sort()` to sort an associative array will destroy the keys in that array. If you want to retain the keys in an array while still sorting by values, use `asort()`.

To do this, invoke `asort()` exactly the same way as you do `sort()`; you can even use the same flags as defined in Table 7-1. The resulting array will, however, retain the same keys as before, even though elements are in a different order. For example:

```
$numbers = [1, 10, 100, 5, 50, 500];
asort($numbers, SORT_NUMERIC);
print_r($numbers);

// Array
// (
//     [0] => 1
//     [3] => 5
//     [1] => 10
//     [4] => 50
//     [2] => 100
//     [5] => 500
// )
```

Both `sort()` and `asort()` will produce arrays sorted in ascending order. If you want to get an array in descending order, you have two options:

- Sort the array in ascending order, then reverse it as demonstrated in Recipe 7.9.
- Leverage `rsort()` or `arsort()` for numeric and associative arrays, respectively.

To reduce overall code complexity, the latter option is often preferable. The functions have the same signatures as `sort()` and `asort()` but merely reverse the order in which elements will be positioned in the resulting array.

See Also

Documentation on `arsort()`, `asort()`, `rsort()`, and `sort()`.

7.11 Sorting an Array Based on a Function

Problem

You want to sort an array based on a user-defined function or comparator.

Solution

Use `usort()` with a custom sorting callback as follows:

```
$bonds = [
    ['first' => 'Sean',    'last' => 'Connery'],
    ['first' => 'Daniel',  'last' => 'Craig'],
    ['first' => 'Pierce',  'last' => 'Brosnan'],
    ['first' => 'Roger',   'last' => 'Moore'],
    ['first' => 'Timothy', 'last' => 'Dalton'],
    ['first' => 'George',  'last' => 'Lazenby'],
];

function sorter(array $a, array $b) {
    return [$a['last'], $a['first']] <=> [$b['last'], $
}

usort($bonds, 'sorter');

foreach ($bonds as $bond) {
    echo "{$bond['last']}. {$bond['first']} {$bond['las
}
```

Discussion

The `usort()` function leverages a user-defined function as the comparison operation behind its sorting algorithm. You can pass in any callable as the second parameter, and every element of the array will be checked through this

function to determine its appropriate order. The Solution example references a callback by its name, but you could just as easily pass an anonymous function as well.

The Solution example further leverages PHP's newer spaceship operator to conduct a complex comparison between your array elements.[6] In this particular case, you want to sort James Bond actors first by last name, then by first name. The same function could be used for any collection of names.

A more powerful example is to apply custom sorting to dates in PHP. Dates are relatively easy to sort as they're part of a continuous series. But it's possible to define custom behavior that breaks those expectations. Example 7-20 attempts to sort an array of dates first based on the day of the week, then by the year, then by the month.

Example 7-20. User-defined sorting applied to dates

```
$dates = [
    new DateTime('2022-12-25'),
    new DateTime('2022-04-17'),
    new DateTime('2022-11-24'),
    new DateTime('2023-01-01'),
    new DateTime('2022-07-04'),
    new DateTime('2023-02-14'),
];

function sorter(DateTime $a, DateTime $b) {
    return
        [$a->format('N'), $a->format('Y'), $a->format('
        <=>
        [$b->format('N'), $b->format('Y'), $b->format('
}

usort($dates, 'sorter');

foreach ($dates as $date) {
    echo $date->format('l, F jS, Y') . PHP_EOL;
}

// Monday, July 4th, 2022
// Tuesday, February 14th, 2023
// Thursday, November 24th, 2022
// Sunday, April 17th, 2022
```

```
// Sunday, December 25th, 2022
// Sunday, January 1st, 2023
```

Like many other array functions discussed in this chapter, `usort()` ignores array keys/indexes and re-indexes the array as part of its operation. If you need to retain the index or key associations of elements, use `uasort()` instead. This function has the same signature as `usort()` but leaves the array keys untouched after sorting.

Array keys often hold important information about the data within the array, so retaining them during a sorting operation can prove critical at times. In addition, you might want to actually sort by the keys of the array rather than by the value of each element. In those circumstances, leverage `uksort()`.

The `uksort()` function will sort an array by its keys, using a function you define. Like `uasort()`, it respects the keys and leaves them in place after the array is sorted.

See Also

Documentation on `usort()`, `uasort()`, and `uksort()`.

7.12 Randomizing the Elements in an Array

Problem

You want to scramble the elements of your array so that their order is entirely random.

Solution

Use `shuffle()` as follows:

```
$array = range('a', 'e');
shuffle($array);
```

Discussion

The `shuffle()` function acts on an existing array that is passed into the function by reference. It completely ignores the keys of the array and sorts element values at random, updating the array in place. After shuffling, array keys are re-indexed starting from 0.

WARNING

While you won't receive an error if you shuffle an associative array, all information on keys will be lost during the operation. You should only ever shuffle numeric arrays.

Internally, `shuffle()` uses the Mersenne Twister pseudorandom number generator to identify a new, seemingly random order for each element in the array. This pseudorandom number generator is not suitable when true randomness is required (e.g., cryptography or security scenarios), but it is an effective way to quickly shuffle the contents of an array.

See Also

Documentation on `shuffle()`.

7.13 Applying a Function to Every Element of an Array

Problem

You want to transform an array by applying a function to modify every element of the array in turn.

Solution

To modify the array in place, use `array_walk()` as follows:

```
$values = range(2, 5);

array_walk($values, function(&$value, $key) {
```

```
    $value *= $value;
});

print_r($values);

// Array
// (
//     [0] => 4
//     [1] => 9
//     [2] => 16
//     [3] => 25
// )
```

Discussion

Looping through collections of data is a common requirement for PHP applications. For example, you may want to use collections to define repeated tasks. Or you may want to perform a particular operation on every item in a collection, like squaring values, as shown in the Solution example.

The `array_walk()` function is an effective way to both define the transformation you want applied and to apply it to the value of every element of the array. The callback function (the second parameter) accepts three arguments: the value and key for an element in the array and an optional `$arg` argument. This final argument is defined during the initial invocation of `array_walk()` and is passed to every use of the callback. It's an efficient way to pass a constant value to the callback, as shown in Example 7-21.

Example 7-21. Invoking `array_walk()` with an extra argument

```
function mutate(&$value, $key, $arg)
{
    $value *= $arg;
}

$values = range(2, 5);

array_walk($values, 'mutate', 10);

print_r($values);

// Array
```

```
// (
//     [0] => 20
//     [1] => 30
//     [2] => 40
//     [3] => 50
// )
```

Using `array_walk()` to modify an array in place requires passing array values *by reference* into the callback (note the extra `&` in front of the argument name). This function could also be used to merely walk over each element in the array and perform some other function *without* modifying the source array. In fact, that's the most common use of this function.

In addition to walking over every element of an array, you can walk over the *leaf* nodes in a nested array by using `array_walk_recursive()`. Unlike the preceding examples, `array_walk_recursive()` will traverse nested arrays until it finds a non-array element before applying your specified callback function. Example 7-22 handily demonstrates the difference between the recursive and nonrecursive function calls against a nested array. Specifically, if you are dealing with a nested array, `array_walk()` will throw an error and fail to do anything at all.

Example 7-22. Comparing `array_walk()` with `array_walk_recursive()`

```
$array = [
    'even' => [2, 4, 6],
    'odd'  => 1,
];

function mutate(&$value, $key, $arg)
{
    $value *= $arg;
}

array_walk_recursive($array, 'mutate', 10);
print_r($array);

// Array
// (
//     [even] => Array
//         (
//             [0] => 20
```

```
//             [1] => 40
//             [2] => 60
//         )
//
//     [odd] => 10
// )


array_walk($array, 'mutate', 10);

// PHP Warning: Uncaught TypeError: Unsupported operanc
```

In many situations, you might want to create a new copy of a mutated array without losing track of its original state. In those circumstances, `array_map()` might be a safer choice than `array_walk()`. Rather than modifying the source array, `array_map()` empowers you to apply a function to every element in the source array and return an entirely new array. The advantage is that you'll have both the original and the modified arrays available for further use. The following example leverages the same logic as the Solution example *without* changing the source array:

```
$values = range(2, 5);

$mutated = array_map(function($value) {
    return $value * $value;
}, $values);

print_r($mutated);

// Array
// (
//     [0] => 4
//     [1] => 9
//     [2] => 16
//     [3] => 25
// )
```

Here are some key differences to note between these two families of array functions:

- `array_walk()` expects the array first and the callback second.
- `array_map()` expects the callback first and the array second.

- `array_walk()` returns a Boolean flag, while `array_map()` returns a new array.
- `array_map()` does not pass keys into the callback.
- `array_map()` does not pass additional arguments into the callback.
- There is no recursive form of `array_map()`.

See Also

Documentation on `array_map()`, `array_walk()`, and `array_walk_recursive()`.

7.14 Reducing an Array to a Single Value

Problem

You want to iteratively reduce a collection of values to a single value.

Solution

Use `array_reduce()` with a callback as follows:

```
$values = range(0, 10);

$sum = array_reduce($values, function($carry, $item) {
    return $carry + $item;
}, 0);

// $sum = 55
```

Discussion

The `array_reduce()` function walks through every element of an array and modifies its own internal state to eventually arrive at a single answer. The Solution example walks through each element of a list of numbers and adds them all to the initial value of `0`, returning the final sum of all of the numbers in question.

The callback function accepts two parameters. The first is the value you're carrying over from the last operation. The second is the value of the current item in the array over which you're iterating. Whatever the callback returns will be passed into the callback as the `$carry` parameter for the next element in the array.

When you first start out, you pass an optional initial value (`null` by default) into the callback as the `$carry` parameter. If the reduction operation you're applying to the array is straightforward, you can often provide a better initial value, as done in the Solution example.

The biggest drawback of `array_reduce()` is that it does not handle array keys. In order to leverage any keys in the array as part of the reduction operation, you need to define your own version of the function.

Example 7-23 shows how you can instead iterate over the array returned by `array_keys()` to leverage elements' keys and values in the reduction. You pass both the array and callback into the closure processed by `array_reduce()` so you can both reference the *element* in the array defined by that key and apply your custom function to it. In the main program, you are then free to reduce an associative array the same way you would a numeric one—except you have an extra argument in your callback containing each element's key.

Example 7-23. Associative alternative to `array_reduce()`

```
function array_reduce_assoc(
    array $array,
    callable $callback,
    mixed $initial = null
): mixed
{
    return array_reduce(
        array_keys($array),
        function($carry, $item) use ($array, $callback)
            return $callback($carry, $array[$item], $it
        },
        $initial
    );
}

$array = [1 => 10, 2 => 10, 3 => 5];
```

```
$sumMultiples = array_reduce_assoc(
    $array,
    function($carry, $item, $key) {
        return $carry + ($item * $key);
    },
    0
);

// $sumMultiples = 45
```

The preceding code will return the sum of the keys of `$array` multiplied by their corresponding values—specifically, `1 * 10 + 2 * 10 + 3 * 5 = 45`.

See Also

Documentation on `array_reduce()`.

7.15 Iterating over Infinite or Very Large/Expensive Arrays

Problem

You want to iterate over a list of items that is too large to be held in memory or is too slow to generate.

Solution

Use a generator to yield one chunk of data at a time to your program, as follows:

```
function weekday()
{
    static $day = 'Monday';

    while (true) {
        yield $day;
```

```
            switch($day) {
                case 'Monday':
                    $day = 'Tuesday';
                    break;
                case 'Tuesday':
                    $day = 'Wednesday';
                    break;
                case 'Wednesday':
                    $day = 'Thursday';
                    break;
                case 'Thursday':
                    $day = 'Friday';
                    break;
                case 'Friday':
                    $day = 'Monday';
                    break;
            }
        }
    }

    $weekdays = weekday();
    foreach ($weekdays as $day) {
        echo $day . PHP_EOL;
    }
```

Discussion

Generators are a memory-efficient way to handle large pieces of data in PHP. In the Solution example, a generator produces weekdays (Monday through Friday) in order as an infinite series. An infinite series will not fit in the memory available to PHP, but the generator construct allows you to build it up one piece at a time.

Rather than instantiate a too-large array, you generate the first piece of data and return it via the `yield` keyword to whomever called the generator. This freezes the state of the generator and yields executional control back to the main application. Unlike a typical function that returns data once, a generator can provide data multiple times so long as it is still valid.

In the Solution example, the `yield` appears inside an infinite `while` loop, so it will continue enumerating weekdays forever. If you wanted the generator

to exit, you would do so using an empty `return` statement at the end (or merely break the loop and implicitly return).

TIP

Returning data from a generator is different from a usual function call. You typically return data with the `yield` keyword and exit a generator with an empty `return` statement. However, if the generator *does* have a final return, you must access that data by calling `::getReturn()` on the generator object. This additional method call often sticks out as odd, so unless your generator has a reason to return data outside its typical `yield` operation, you should try to avoid it.

Since the generator can provide data forever, you can iterate over that data by using a standard `foreach` loop. Similarly, you could leverage a limited `for` loop to avoid an infinite series. The following code leverages such a limited loop and the Solution's original generator:

```
$weekdays = weekday();
for ($i = 0; $i < 14; $i++) {
    echo $weekdays->current() . PHP_EOL;
    $weekdays->next();
}
```

Though the generator is defined as a function, internally PHP recognizes it as a generator and converts it to an instance of the `Generator` class. This class gives you access to the `::current()` and `::next()` methods and permits you to step over the generated data one piece at a time.

The control flow within the application passes back and forth between the main program and the generator's `yield` statement. The first time you access the generator, it runs internally up to `yield` and then returns control (and possibly data) to the main application. Subsequent calls to the generator start *after* the `yield` keyword. Loops are required to force the generator back to the beginning in order to `yield` again.

See Also

Overview on generators.

[1] Class inheritance is discussed in Chapter 8, and object interfaces are explicitly covered in Recipe 8.7.

[2] See Recipe 7.15 for examples of very large iterable data structures.

[3] Take care not to confuse `array_splice()` with `array_slice()`. The two functions have vastly different uses, and the latter is covered in Recipe 7.7.

[4] The `array_splice()` function will also *return* the elements it extracted from the target array, in the event you need to use that data for some other operation. See Recipe 7.7 for further discussion of this behavior.

[5] Review "Comparison Operators" for further details on comparison operators and their usage.

[6] The spaceship operator is explained at length in Recipe 2.4, which also introduces an example use of `usort()`.