Chapter 5. Numbers

Another fundamental building block of data in PHP is numbers. It's easy to find different types of numbers in the world around us. The page number in a book is often printed in the footer. Your smartwatch displays the current time and perhaps the number of steps you've taken today. Some numbers can be impossibly large, others impossibly small. Numbers can be whole, fractional, or irrational like π.

In PHP, numbers are represented natively in one of two formats: as integers (the `int` type) or as floating-point numbers (the `float` type). Both numeric types are highly flexible, but the range of values you can use depends on the processor architecture of your system—a 32-bit system has tighter bounds than a 64-bit system.

PHP defines several constants that help programs understand the available range of numbers in the system. Given that the capabilities of PHP will differ greatly based on how it was compiled (for 32 or 64 bits), it is wise to use the constants defined in Table 5-1 rather than trying to determine what these values will be in a program. It's always safer to defer to the operating system and language defaults.

Table 5-1. Constant numeric values in PHP

Constant	Description
`PHP_INT_MAX`	The largest integer value supported by PHP. On 32-bit systems, this will be `2147483647`. On 64-bit systems, this will be `9223372036854775807`.
`PHP_INT_MIN`	The smallest integer value supported by PHP. On 32-bit systems, this will be `-2147483648`. On 64-bit systems, this will be `-9223372036854775808`.

Constant	Description
`PHP_INT_SIZE`	Size of integers in bytes for this build of PHP.
`PHP_FLOAT_DIG`	Number of digits that can be rounded in a `float` and back without a loss in precision.
`PHP_FLOAT_EPSILON`	The smallest representable positive number *x* such that *x* `+ 1.0 !== 1.0`.
`PHP_FLOAT_MIN`	Smallest representable positive floating-point number.
`PHP_FLOAT_MAX`	Largest representable floating-point number.
`-PHP_FLOAT_MAX`	Not a separate constant, but the way to represent the smallest negative floating-point number.

The unfortunate limitation of PHP's number systems is that very large or very small numbers cannot be represented natively. Instead, you need to leverage an extension like BCMath or GNU Multiple Precision Arithmetic Library (GMP), both of which wrap operating system–native operations on numbers. I'll cover GMP specifically in Recipe 5.10.

The recipes that follow cover many of the problems developers need to solve with numbers in PHP.

5.1 Validating a Number Within a Variable

Problem

You want to check whether a variable contains a number, even if that variable is explicitly typed as a string.

Solution

Use `is_numeric()` to check whether a variable can be successfully cast as a numeric value—for example:

```
$candidates = [
    22,
    '15',
    '12.7',
    0.662,
    'infinity',
    INF,
    0xDEADBEEF,
    '10e10',
    '15 apples',
    '2,500'
];

foreach ($candidates as $candidate) {
    $numeric = is_numeric($candidate) ? 'is' : 'is NOT'

    echo "The value '{$candidate}' {$numeric} numeric.'
}
```

The preceding example will print the following when run in a console:

```
The value '22' is numeric.
The value '15' is numeric.
The value '12.7' is numeric.
The value '0.662' is numeric.
The value 'infinity' is NOT numeric.
The value 'INF' is numeric.
The value '3735928559' is numeric.
The value '10e10' is numeric.
The value '15 apples' is NOT numeric.
The value '2,500' is NOT numeric.
```

Discussion

At its core, PHP is a dynamically typed language. You can easily interchange strings for integers (and vice versa), and PHP will try to infer your intention,

dynamically casting values from one type to another as needed. While you can (and probably should) enforce strict typing as discussed in Recipe 3.4, often you will explicitly need to encode numbers as strings.

In those situations, you will lose the ability to identify numeric strings by leveraging PHP's type system. A variable passed into a function as a `string` will be invalid for mathematical operations without an explicit cast to a numeric type (`int` or `float`). Unfortunately, not every string that contains numbers is numeric.

The string `15 apples` contains a number but is not numeric. The string `10e10` contains non-numeric characters but is a valid numeric representation.

The difference between strings that have numbers and truly numeric strings can be best illustrated through a userland implementation of PHP's native `is_numeric()` function, as defined in Example 5-1.

Example 5-1. Userland `is_numeric()` implementation

```
function string_is_numeric(string $test): bool
{
    return $test === (string) floatval($test);
}
```

Applied to the same array of `$candidates` from the Solution example, Example 5-1 will accurately verify numeric strings in everything *except* the literal `INF` constant and the `10e10` exponent shorthand. This is because `floatval()` will strip any non-numeric characters from the string entirely while converting it to a floating-point number prior to `(string)` casting things back to a string.[1]

The userland implementation isn't adequate for every situation, so you should use the native implementation to be safe. The goal of `is_numeric()` is to indicate whether a given string can be safely cast to a numeric type without losing information.

See Also

PHP documentation for `is_numeric()` .

5.2 Comparing Floating-Point Numbers

Problem

You want to test for equality of two floating-point numbers.

Solution

Define an appropriate error bound (called `epsilon`) that represents the greatest acceptable difference between the two numbers and evaluate their difference against it as follows:

```
$first = 22.19348234;
$second = 22.19348230;

$epsilon = 0.000001;

$equal = abs($first - $second) < $epsilon; // true
```

Discussion

Floating-point arithmetic with modern computers is less than exact because of the way machines internally represent numbers. Different operations you might calculate by hand and assume to be precise can trip up the machines you rely on.

For example, the mathematic operation `1 - 0.83` is obviously `0.17`. It's simple enough to mentally calculate or even work out on paper. But asking a computer to calculate this will produce a strange result, as demonstrated in Example 5-2.

Example 5-2. Floating-point subtraction

```
$a = 0.17;
$b = 1 - 0.83;

var_dump($a == $b);
```

❶

```
var_dump($a);
                                    ❷

var_dump($b);
                                    ❸
```

```
bool(false)                         ❶
float(0.17)                         ❷
float(0.17000000000000004)
```

When it comes to floating-point arithmetic, the best computers can do is an approximate result within an acceptable margin of error. As a result, comparing this result to an expected value requires the explicit definition of that margin (`epsilon`) and a comparison to that margin rather than an exact value.

Rather than leverage either of PHP's equality operators (a double or triple equals sign), you can define a function to check for the *relative* equality of two floats, as shown in Example 5-3.

Example 5-3. Comparing equality of floating-point numbers

```
function float_equality(float $epsilon): callable
{
    return function(float $a, float $b) use ($epsilon):
    {
        return abs($a - $b) < $epsilon;
    };
}

$tight_equality = float_equality(0.0000001);
$loose_equality = float_equality(0.01);

var_dump($tight_equality(1.152, 1.152001));
                                    ❶

var_dump($tight_equality(0.234, 0.2345));
                                    ❷

var_dump($tight_equality(0.234, 0.244));
                                    ❸

var_dump($loose_equality(1.152, 1.152001));
                                    ❹

var_dump($loose_equality(0.234, 0.2345));
                                    ❺
```

```
var_dump($loose_equality(0.234, 0.244));
                                        ❻
```

```
bool(false)   ❶
bool(false)   ❷
bool(false)   ❸
bool(true)    ❹
bool(true)    ❺
bool(true)    ❻
```

See Also

PHP documentation on floating-point numbers.

5.3 Rounding Floating-Point Numbers

Problem

You want to round a floating-point number either to a fixed number of decimal places or to an integer.

Solution

To round a floating-point number to a set number of decimal places, use `round()` while specifying the number of places:

```
$number = round(15.31415, 1);
// 15.3
```

To explicitly round up to the nearest whole number, use `ceil()`:

```
$number = ceil(15.3);
// 16
```

To explicitly round down to the nearest whole number, use `floor()`:

```
$number = floor(15.3);
// 15
```

Discussion

All three functions referenced in the Solution examples— `round()`, `ceil()`, and `floor()` —are intended to operate on any numeric value but will return a `float` after operating on it. By default, `round()` will round to zero digits after the decimal point but will still return a floating-point number.

To convert from a `float` to an `int` for any of these functions, wrap the function itself in `intval()` to convert to an integer type.

Rounding in PHP is more flexible than merely rounding up or down. By default, `round()` will always round the input number away from 0 when it's halfway there. This means numbers like 1.4 will round down, while 1.5 will round up. This also holds true with negative numbers: −1.4 will be rounded towards 0 to −1, while −1.5 will be rounded away from 0 to −2.

You can change the behavior of `round()` by passing an optional third argument (or by using named parameters as shown in Recipe 3.3) to specify the rounding mode. This argument accepts one of four default constants defined by PHP, as enumerated in Table 5-2.

Table 5-2. Rounding mode constants

Constant	Description
`PHP_ROUND_HALF_UP`	Rounds a value away from 0 when it is halfway there, making 1.5 into 2 and −1.5 into −2
`PHP_ROUND_HALF_DOWN`	Rounds a value towards 0 when it is halfway there, making 1.5 into 1 and −1.5 into −1

Constant	Description
PHP_ROUND_HALF_EVEN	Rounds a value towards the nearest even value when it is halfway there, making both 1.5 and 2.5 into 2
PHP_ROUND_HALF_ODD	Rounds a value towards the nearest odd value when it is halfway there, making 1.5 into 1 and 2.5 into 3

Example 5-4 illustrates the effect of each rounding mode constant when applied to the same numbers.

Example 5-4. Rounding floats in PHP with different modes

```
echo 'Rounding on 1.5' . PHP_EOL;
var_dump(round(1.5, mode: PHP_ROUND_HALF_UP));
var_dump(round(1.5, mode: PHP_ROUND_HALF_DOWN));
var_dump(round(1.5, mode: PHP_ROUND_HALF_EVEN));
var_dump(round(1.5, mode: PHP_ROUND_HALF_ODD));

echo 'Rounding on 2.5' . PHP_EOL;
var_dump(round(2.5, mode: PHP_ROUND_HALF_UP));
var_dump(round(2.5, mode: PHP_ROUND_HALF_DOWN));
var_dump(round(2.5, mode: PHP_ROUND_HALF_EVEN));
var_dump(round(2.5, mode: PHP_ROUND_HALF_ODD));
```

The preceding example will print the following to your console:

```
Rounding on 1.5
float(2)
float(1)
float(2)
float(1)
Rounding on 2.5
float(3)
float(2)
float(2)
float(3)
```

See Also

PHP documentation on floating-point numbers, the `round()` function, the `ceil()` function, and the `floor()` function.

5.4 Generating Truly Random Numbers

Problem

You want to generate random integers within specific bounds.

Solution

Use `random_int()` as follows:

```
// Random integer between 10 and 225, inclusive
$random_number = random_int(10, 225);
```

Discussion

When you need randomness, you most often need explicitly true, completely unpredictable randomness. In those situations, you can rely on the cryptographically secure pseudorandom number generators built into the machine itself. PHP's `random_int()` function relies on these operating system–level number generators rather than implementing its own algorithm.

On Windows, PHP will leverage either `CryptGenRandom()` or the `Cryptography API: Next Generation (CNG)` depending on the language version in use. On Linux, PHP leverages a system call to `getrandom(2)`. On any other platform, PHP will fall back on the system-level */dev/urandom* interface. All of these APIs are well tested and proven to be cryptographically secure, meaning they generate numbers with sufficient randomness that they are indistinguishable from noise.

NOTE

In rare situations, you'll want a random number generator to produce a predictable series of pseudorandom values. In those circumstances, you can rely on algorithmic generators like the Mersenne Twister, which is further discussed in Recipe 5.5.

PHP doesn't natively support a method to create a random floating-point number (i.e., selecting a random decimal between 0 and 1). Instead, you can use `random_int()` and your knowledge of integers in PHP to create your own function to do exactly that, as shown in Example 5-5.

Example 5-5. Userland function for generating a random floating-point number

```
function random_float(): float
{
    return random_int(0, PHP_INT_MAX) / PHP_INT_MAX;
}
```

This implementation of `random_float()` lacks bounds because it will always generate a number between 0 and 1, inclusively. This might be useful to create random percentages, either for creating artificial data or for selecting randomly sized samples of arrays. A more complicated implementation might incorporate bounds as shown in Example 5-6, but often being able to choose between 0 and 1 is utility enough.

Example 5-6. Userland function for generating a random `float` within bounds

```
function random_float(int $min = 0, int $max = 1): floa
{
    $rand = random_int(0, PHP_INT_MAX) / PHP_INT_MAX;

    return ($max - $min) * $rand + $min;
}
```

This newer definition of `random_float()` merely scales the original definition to the newly defined bounds. If you were to leave the default bounds in place, the function reduces down to the original definition.

See Also

PHP documentation on `random_int()` .

5.5 Generating Predictable Random Numbers

Problem

You want to predict random numbers in such a way that the sequence of numbers is the same every time.

Solution

Use the `mt_rand()` function after passing a predefined seed into `mt_srand()` —for example:

```
function generate_sequence(int $count = 10): array
{
    $array = [];

    for ($i = 0; $i < $count; $i++) {
        $array[] = mt_rand(0, 100);
    }

    return $array;
}

mt_srand(42);
$first = generate_sequence();

mt_srand(42);
$second = generate_sequence();

print_r($first);
print_r($second);
```

Both arrays in the preceding example will have the following contents:

```
Array
(
    [0] => 38
    [1] => 32
    [2] => 94
    [3] => 55
    [4] => 2
    [5] => 21
    [6] => 10
    [7] => 12
    [8] => 47
    [9] => 30
)
```

Discussion

When writing any other example about truly random numbers, the best anyone can do is to illustrate what the output *might* look like. When it comes to the output of `mt_rand()`, however, the output will be the same on every computer, given you're using the same seed.

NOTE

PHP automatically seeds `mt_rand()` at random by default. It is not necessary to specify your own seed unless your goal is deterministic output from the function.

The output is the same because `mt_rand()` leverages an algorithmic pseudorandom number generator called the *Mersenne Twister*. This is a well-known and heavily used algorithm first introduced in 1997; it's also used in languages like Ruby and Python.

Given an initial seed value, the algorithm creates an initial state and then generates seemingly random numbers by executing a "twist" operation on that state. The advantage of this approach is that it's deterministic—given the same seed, the algorithm will create the same sequence of "random" numbers every time.

WARNING

Predictability in random numbers can be hazardous to certain computing operations, specifically to cryptography. The use cases requiring a deterministic sequence of pseudorandom numbers are rare enough that `mt_rand()` should be avoided as much as possible. If you need to generate random numbers, leverage true sources of randomness like `random_int()` and `random_bytes()`.

Creating a pseudorandom but predictable sequence of numbers might be useful in creating object IDs for a database. You can easily test that your code operates correctly by running it multiple times and verifying the output. The disadvantage is that algorithms like the Mersenne Twister can be gamed by an outside party.

Given a sufficiently long sequence of seemingly random numbers and knowledge of the algorithm, it is trivial to reverse the operation and identify the original seed. Once an attacker knows the seed value, they can generate every possible "random" number your system will leverage moving forward.

See Also

PHP documentation on `mt_rand()` and `mt_srand()`.

5.6 Generating Weighted Random Numbers

Problem

You want to generate random numbers in order to select a specific item from a collection at random, but you want some items to have a higher chance of being selected than others. For example, you want to select the winner of a particular challenge at an event, but some participants have earned more points than others and need to have a greater chance of being selected.

Solution

Pass a map of choices and weights into an implementation of `weighted_random_choice()`, as demonstrated in Example 5-7.

Example 5-7. Implementation of a weighted random choice

```
$choices = [
    'Tony'  => 10,
    'Steve' => 2,
    'Peter' => 1,
    'Wanda' => 4,
    'Carol' => 6
];

function weighted_random_choice(array $choices): string
{
    arsort($choices);

    $total_weight = array_sum(array_values($choices));
    $selection = random_int(1, $total_weight);

    $count = 0;
    foreach ($choices as $choice => $weight) {
        $count += $weight;
        if ($count >= $selection) {
            return $choice;
        }
    }

    throw new Exception('Unable to make a choice!');
}

print weighted_random_choice($choices);
```

Discussion

In the Solution example, each possible choice is assigned a weight. To choose a final option, you can *order* each option by weight, with the highest-weighted option coming first in the list. You then identify a random number somewhere in the field of total possible weights. That random number selects which of the options you chose.

This is easiest to visualize on a number line. In the Solution example, Tony is entered into the selection with a weight of 10 and Peter with a weight of 1. This means Tony is 10 times as likely to win as Peter, but it's still possible *neither* of them will be chosen. Figure 5-1 illustrates the relative weight of

each if you order the possible choices by weight and print them on a number line.

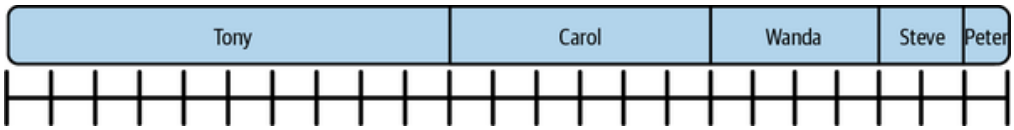

Figure 5-1. Potential selections ordered and visualized by weight

The algorithm defined in `weighted_random_choice()` will check whether the selected random number is within the bounds of each possible choice and, if not, move on to the next candidate. If, for any reason, you're unable to make a selection, the function will throw an exception.[2]

It is possible to verify the weighted nature of this choice by executing a random selection a thousand times and then plotting the relative number of times each choice is picked. Example 5-8 shows how such a repeated choice can be tabulated, while Figure 5-2 illustrates the outcome. Both demonstrate just how much more likely Tony is to be chosen than any other option in the candidate array.

Example 5-8. Repeated selection of a weighted random choice

```
$output = fopen('output.csv', 'w');
fputcsv($output, ['selected']);

foreach (range(0, 1000) as $i) {
    $selection = weighted_random_choice($choices);
    fputcsv($output, [$selection]);
}
fclose($output);
```

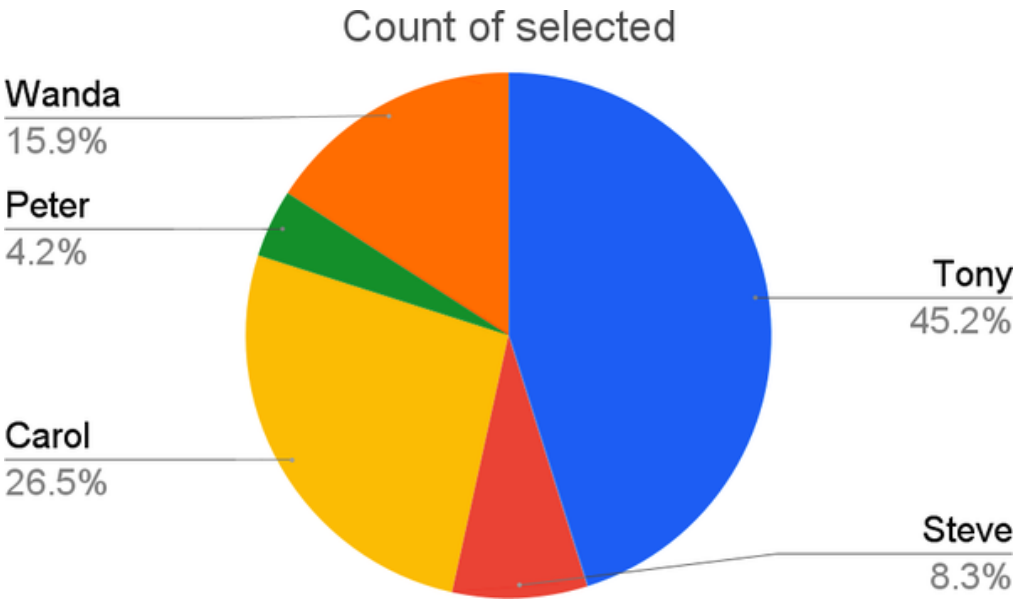

Figure 5-2. Pie chart illustrating the relative number of times each choice is selected

This illustration of outcomes after 1,000 iterations clearly demonstrates that Tony is chosen roughly 10 times more frequently than Peter. This lines up perfectly with his having a weight of 10 to Peter's 1. Likewise, Wanda's weight of 4 reliably lines up with her being chosen twice as frequently as Steve, who has a weight of 2.

Given that the choices here are random, running the same experiment again will result in slightly different percentages for each candidate. However, the integer weights of each will always translate into roughly the same distribution of choices.

See Also

PHP documentation on `random_int()` and `arsort()` as well as Recipe 5.4 for further examples of `random_int()` in practice.

5.7 Calculating Logarithms

Problem

You want to calculate the logarithm of a number.

Solution

For natural logarithms (using base `e`), use `log()` as follows:

```
$log = log(5);
// 1.6094379124341
```

For any arbitrary base logarithm, specify the base as a second optional parameter:

```
$log2 = log(16, 2);
// 4.0
```

Discussion

PHP supports the calculation of logarithms with its native Math extension. When you call `log()` without specifying a base, PHP will fall back on the default `M_E` constant, which is coded to the value of `e`, or approximately 2.718281828459.

If you try to take the logarithm of a negative value, PHP will always return `NAN`, a constant (typed as a `float`) that represents *not a number*. If you attempt to pass a negative base, PHP will return a `ValueError` and trigger a warning.

Any positive, nonzero base is supported by `log()`. Many applications use base 10 so frequently that PHP supports a separate `log10()` function for just that base. This is functionally equivalent to passing the integer `10` as a base to `log()`.

See Also

PHP documentation on the various functionality supported by the Math extension, including `log()` and `log10()`.

5.8 Calculating Exponents

Problem

You want to raise a number to an arbitrary power.

Solution

Use PHP's `pow()` function as follows:

```
// 2^5
$power = pow(2, 5); // 32

// 3^0.5
$power = pow(3, 0.5); // 1.7320508075689

// e^2
$power = pow(M_E, 2); // 7.3890560989306
```

Discussion

PHP's `pow()` function is an efficient way to raise any number to an arbitrary power and return either the integer or floating-point result. In addition to the function form, PHP provides a special operator for raising a number to a power: `**`.

The following code is equivalent to the use of `pow()` in the Solution examples:

```
// 2^5
$power = 2 ** 5; // 32

// 3^0.5
$power = 3 ** 0.5; // 1.7320508075689

// e^2
$power = M_E ** 2; // 7.3890560989306
```

WARNING

While the mathematical shorthand for exponentiation is usually the caret (^), this character in PHP is reserved for the XOR operator. Review Chapter 2 for more details on this and other operators.

While raising the constant `e` to an arbitrary power is possible through `pow()`, PHP also ships with a specific function for that use: `exp()`. The statements `pow(M_E, 2)` and `exp(2)` are functionally equivalent. They are implemented via different code and, because of the way floating-point numbers are represented internally by PHP, will return slightly different results.[3]

See Also

PHP documentation on `pow()` and `exp()`.

5.9 Formatting Numbers as Strings

Problem

You want to print a number with thousands separators to make it more readable to end users of your application.

Solution

Use `number_format()` to automatically add thousands separators when converting a number to a string. For example:

```
$number = 25519;
print number_format($number);
// 25,519

$number = 64923.12
print number_format($number, 2);
// 64,923.12
```

Discussion

PHP's native `number_format()` function will automatically group thousands together as well as round decimal digits to the given precision. You can also optionally *change* the decimal and thousands separators to match a given locale or format.

For example, assume you want to use periods to separate thousands groups and a comma to separate decimal digits (as is common in Danish number formats). To accomplish this, you would leverage `number_format()` as follows:

```
$number = 525600.23;

print number_format($number, 2, ',', '.');
// 525.600,23
```

PHP's native `NumberFormatter` class provides similar utility but gives you the ability to explicitly define the locale rather than needing to remember a specific regional format.[4] You can rewrite the preceding example to use `NumberFormatter` specifically with the `da_DK` locale to identify a Danish format as follows:

```
$number = 525600.23;

$fmt = new NumberFormatter('da_DK', NumberFormatter::DE
print $fmt->format($number);
// 525.600,23
```

See Also

PHP documentation on `number_format()` and the `NumberFormatter` class.

5.10 Handling Very Large or Very Small Numbers

Problem

You need to use numbers that are too large (or too small) to be handled by PHP's native integer and floating-point types.

Solution

Use the GMP library:

```
$sum = gmp_pow(4096, 100);
print gmp_strval($sum);
// 172184794563857506180673776960526354835799247454486
// 740691241745619397484537236046173286370919031961587
// 102499160988272871734465950347165599088088467989652
// 190565262313456852682405692098925737660379665847351
// 578587782701380797240772477647874555986712746271362
// 4435913511141036261376
```

Discussion

PHP supports two extensions for working with numbers either too large or too small to be represented with native types. The BCMath extension is an interface to a system-level *basic calculator* utility that supports arbitrary precision mathematics. Unlike native PHP types, BCMath supports working with up to 2,147,483,647 decimal digits so long as the system has adequate memory.

Unfortunately, BCMath encodes all numbers as regular strings in PHP, which makes using it somewhat difficult in modern applications that target strict type enforcement.[5]

The GMP extension is a valid alternative also available to PHP that does not have this drawback. Internal to itself, numbers are stored as strings. They are, however, wrapped as `GMP` objects when exposed to the rest of PHP. This distinction helps clarify whether a function is operating on a small number encoded as a string or a large one necessitating the use of an extension.

NOTE

BCMath and GMP act on and return integer values rather than floating points. If you need to conduct operations on floating-point numbers, you might need to increase the size of your numbers by an order of magnitude (i.e., multiply by 10) and then reduce them again once your calculations are complete in order to account for decimals or fractions.

GMP isn't included with PHP by default, although many distributions will make it available quite easily. If you're compiling PHP from source, doing so with the `--with-gmp` option will add support automatically. If you're using a package manager to install PHP (for example, on a Linux machine) you can likely install the `php-gmp` package to add this support directly.[6]

Once available, GMP will empower you to execute any mathematic operation you desire on numbers of limitless size. The caveat is that you can no longer use native PHP operators and must use a functional format defined by the extension itself. Example 5-9 presents some translations from native operators to GMP function calls. Note that the return type of each function call is a `GMP` object, so you must convert it back to either a number or a string by using `gmp_intval()` or `gmp_strval()`, respectively.

Example 5-9. Various mathematical operations and their GMP function equivalents

```
$add = 2 + 5;
$add = gmp_add(2, 5);

$sub = 23 - 2;
$sub = gmp_sub(23, 2);

$div = 15 / 4;
$div = gmp_div(15, 4);

$mul = 3 * 9;
$mul = gmp_mul(3, 9);

$pow = 4 ** 7;
$pow = gmp_pow(4, 7);

$mod = 93 % 4;
$mod = gmp_mod(93, 4);

$eq = 42 == (21 * 2);
$eq = gmp_cmp(42, gmp_mul(21, 2));
```

The final illustration in Example 5-9 introduces the `gmp_cmp()` function, which allows you to compare two GMP-wrapped values. This function will return a positive value if the first parameter is greater than the second, 0 if they are equal, and a negative value if the second parameter is greater than the

first. It's effectively the same as PHP's spaceship operator (introduced in Recipe 2.4) rather than an equality comparison, which potentially provides more utility.

See Also

PHP documentation on GMP.

5.11 Converting Numbers Between Numerical Bases

Problem

You want to convert a number from one base to another.

Solution

Use the `base_convert()` function as follows:

```
// Base 10 (decimal) number
$decimal = '240';

// Convert from base 10 to base 16
// $hex = 'f0'
$hex = base_convert($decimal, 10, 16);
```

Discussion

The `base_convert()` function attempts to convert a number from one base to another, which is particularly helpful when working with hexadecimal or binary strings of data. PHP will work only with bases between 2 and 36. Bases higher than 10 will use alphabet characters to represent additional digits— `a` is equal to `10`, `b` to `11`, all the way to `z` being equal to `35`.

Note that the Solution example passes a *string* into `base_convert()` rather than an integer or a float value. This is because PHP will attempt to cast the input string to a number with an appropriate base before converting it to

another base and returning a string. Strings are the best way to represent hexadecimal or octal numbers in PHP, but they're generic enough they can represent numbers of *any* base.

PHP supports several other base-specific conversion functions in addition to the more generic `base_convert()`. These additional functions are enumerated in Table 5-3.

WARNING

PHP supports two functions for converting back and forth between binary data and its hexadecimal representation: `bin2hex()` and `hex2bin()`. These functions are *not* intended for converting a string representation of binary (e.g., `11111001`) into hexadecimal but will instead operate the binary *bytes* of that string.

Table 5-3. Specific base conversion functions

Function name	From base	To base
`bindec()`	Binary (encoded as a `string`)	Decimal (encoded as an `int` or, for size reasons, a `float`)
`decbin()`	Decimal (encoded as an `int`)	Binary (encoded as a `string`)
`hexdec()`	Hexadecimal (encoded as a `string`)	Decimal (encoded as an `int` or, for size reasons, a `float`)
`dechex()`	Decimal (encoded as an `int`)	Hexadecimal (encoded as a `string`)
`octdec()`	Octal (encoded as a `string`)	Decimal (encoded as an `int` or, for size reasons, a `float`)

Function name	From base	To base
`decoct()`	Decimal (encoded as an `int`)	Octal (encoded as a `string`)

Note that, unlike `base_convert()`, the specialized base conversion functions often work with numeric types directly. If you are using strict typing, this will avoid requiring an explicit cast from a numeric type to a `string` before changing bases, which *would* be required with `base_convert()`.

See Also

PHP documentation on `base_convert()`.

[1] For more on type casting, review "Type Casting".

[2] Exceptions and error handling are discussed at length in Chapter 12.

[3] For more on the acceptable differences between floating-point numbers, review Recipe 5.2.

[4] The `NumberFormatter` class itself is part of PHP's intl extension. This module is not built in by default and might need to be installed/enabled for the class to be available.

[5] Review Recipe 3.4 for more on strict typing in PHP.

[6] Native extensions are covered in depth in Chapter 15.