PHP 5 Recipes

A Problem-Solution Approach

Lee Babin, Nathan A. Good,
Frank M. Kromann, Jon Stephens

Apress®

PHP 5 Recipes: A Problem-Solution Approach

ISBN (pbk): 1-59059-509-2

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Chris Mills
Technical Reviewer: Rob Kunkle
Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis, Jason Gilmore, Jonathan Hassell, Chris Mills, Dominic Shakeshaft, Jim Sumser
Associate Publisher: Grace Wong
Project Manager: Kylie Johnston
Copy Edit Manager: Nicole LeClerc
Copy Editor: Kim Wimpsett
Assistant Production Director: Kari Brooks-Copony
Production Editor: Katie Stence
Compositor and Artist: Van Winkle Design Group
Proofreader: April Eddy
Indexer: Broccoli Information Management
Interior Designer: Van Winkle Design Group
Cover Designer: Kurt Krames
Manufacturing Manager: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Source Code section.

CHAPTER 3

Performing Math Operations

Math is one of the fundamental elements of most programming languages. Math allows the programmer to perform anything from simple additions to advanced calculations. Even though PHP was designed to create dynamic Hypertext Markup Language (HTML) documents, it has evolved to a general-purpose programming language that includes a strong and flexible math implementation.

The implementation of math in PHP looks very much like the implementation in C. In fact, many of the functions are implemented as simple wrappers around the math functions found in C libraries.

3-1. Numeric Data Types

Working with numbers or numeric data and math functions in PHP is simple. Basically, you have two data types to work with, floating point and integer. The internal representations for these values are the C data types double and int, and these data types follow the same set of rules as in C.

We've designed most the samples in this chapter to work with the command-line interface (CLI) version of PHP. If you use the samples with a web server and view the results in a browser, you may see different formatting than the results shown in this chapter. This is especially true if a recipe is using a variable-width font to present the data. In most cases, we show the generated output following the code sample. When the output generates HTML output, we will use a figure to display the result.

Note The minimum and maximum values for integer values depend on the system architecture where PHP is running. On a 32-bit operating system, an integer can be between –2,147,483,648 and 2,147,483,647.

PHP is a loosely typed scripting language where variables change the data type as needed by calculations. This allows the engine to perform type conversions on the fly. So, when numbers and strings are included in a calculation, the strings will be converted to a numeric value before the calculation is performed, and numeric values are converted to strings before they are concatenated with other strings. In the following example, a string and an integer value are added, and the result is an integer value.

The Code

```
<?php
// Example 3-1-1.php
$a="5";
$b= 7 + $a;
echo "7 + $a = $b";
?>
```

How It Works

The variable `$a` is assigned a string value of 5, and then the variable `$b` is assigned the value of the calculation of 7 plus the value of `$a`. The two values are of different types, so the engine will convert one of them so they are both the same type. The operator + indicates the addition of numeric values to the string, which is converted to a numeric value of 5 before the addition. The last line displays the calculation, and the result is as follows:

```
7 + 5 = 12
```

PHP will also convert the data types of one or more values in a calculation in order to perform the calculation correctly. In the following example, the float is converted to an integer before the binary and (&) operation is executed.

The Code

```
<?php
// Example 3-1-2.php
$a = 3.5;
$b = $a & 2;
echo "$a & 2 = $b";
?>
```

How It Works

The variable `$a` is assigned a floating-point value of 3.5. Then, the variable `$b` is assigned the result of the calculation of `$a` and 2 with the binary and operation. In this case, the floating-point value is converted to an integer (3) before the binary and operation is performed. If you look at the binary values of 3 and 2, you will see these are 011 and 010; if you then perform the operation on each bit, you get the result (`0 & 0 = 0`, `1 & 0 = 0`, and `1 & 1 = 1`).

```
3.5 & 2 = 2
```

And as the next example shows, PHP will perform an additional conversion on the resulting data type if the result of a calculation requires that. So, when an integer is divided by an integer, the resulting value might be an integer or a float depending on the result and not on the operation.

The Code

```
<?php
// Example 3-1-3.php
$a = 5;
$b = $a / 2;
echo "$a / 2 = $b\n";

$a = 6;
$b = $a / 2;
echo "$a / 2 = $b\n";
?>
```

How It Works

This example shows two integer divisions. No data type conversions are needed before the calculations, as both sides of the division operator are numeric, but in the first case where 5 is divided by 2, the result is 2.5, so that value must be stored in a floating-point data type. In the other calculation, where 6 is divided by 2, the result is 6 and can be stored in an integer data type.

```
5 / 2 = 2.5
6 / 2 = 3
```

PHP has a number of functions to test the data type of a variable. Three of these functions test whether the variable contains a numeric value, or, more specifically, whether it is a float or an integer.

The function `is_numeric()` checks if the value passed as the argument is numeric, and as shown in the next example, it will return a boolean value: `true` for integers, floats, and string values with a numeric content and `false` for all other data types. The following example shows how you can use the `is_numeric()` function.

The Code

```
<?php
// Example 3-1-4.php
$a = 1;
echo "is_numeric($a) = " . (is_numeric($a) ? "true" : "false") . "\n";

$a = 1.5;
echo "is_numeric($a) = " . (is_numeric($a) ? "true" : "false") . "\n";

$a = true;
echo "is_numeric($a) = " . (is_numeric($a) ? "true" : "false") . "\n";

$a = 'Test';
echo "is_numeric($a) = " . (is_numeric($a) ? "true" : "false") . "\n";
```

```
$a = '3.5';
echo "is_numeric($a) = " . (is_numeric($a) ? "true" : "false") . "\n";

$a = '3.5E27';
echo "is_numeric($a) = " . (is_numeric($a) ? "true" : "false") . "\n";

$a = 0x19;
echo "is_numeric($a) = " . (is_numeric($a) ? "true" : "false") . "\n";

$a = 0777;
echo "is_numeric($a) = " . (is_numeric($a) ? "true" : "false") . "\n";
?>
```

How It Works

This example shows how you can use the `is_numeric()` function on variables of different data types. In each of the tests, you use the tertiary operator `(?)` to print the string value of `true` or `false` depending on the result returned by the function.

```
is_numeric(1) = true
is_numeric(1.5) = true
is_numeric(1) = false
is_numeric(Test) = false
is_numeric(3.5) = true
is_numeric(3.5E27) = true
is_numeric(25) = true
is_numeric(511) = true
```

The functions `is_int()` and `is_float()` check for specific data types. These functions will return `true` if an integer or float is passed and `false` in any other case, even if a string with a valid numeric representation is passed.

The Code

```
<?php
// Example 3-1-5.php
$a = 123;
echo "is_int($a) = " . (is_int($a) ? "true" : "false") . "\n";

$a = '123';
echo "is_int($a) = " . (is_int($a) ? "true" : "false") . "\n";
?>
```

How It Works

This example shows how the function `is_int()` will return `true` if the value passed as the argument is an integer and `false` if it is anything else, even if the string contains a numeric value.

```
is_int(123) = true
is_int(123) = false
```

To test for other data types, PHP implements `is_bool()`, `is_string()`, `is_array()`, `is_object()`, `is_resource()`, and `is_null()`. All these functions take one argument and return a boolean value.

It is possible to force the engine to change the data type. This is called *typecasting*, and it works by adding `(int)`, `(integer)`, `(float)`, `(double)`, or `(real)` in front of the variable or value or by using the function `intval()` or `floatval()`. This next example shows how you can use the `is_int()` function with the `(int)` typecasting to force a string value to be converted to an integer before the type is checked.

The Code

```
<?php
// Example 3-1-6.php
$a = 123;
echo "is_int($a) = " . (is_int($a) ? "true" : "false") . "\n";

$a = '123';
echo "is_int((int)$a) = " . (is_int((int)$a) ? "true" : "false") . "\n";
?>
```

How It Works

This example works as the previous example does, but because of the implicit typecasting of the string to an integer before calling the `is_int()` function, both tests will return `true`.

```
is_int(123) = true
is_int((int)123) = true
```

Using typecasting might force the value to become a zero value. This will happen if the value is an object, an array, or a string that contains a non-numeric value and if this variable is typecast to an integer or floating-point value.

When the `intval()` function is used on strings, it's possible to pass a second parameter that specifies the base to use for the conversion. The default value is 10, but it's possible to use base 2 (binary), 8 (octal), 16 (hexadecimal), or any other value such as 32 or 36, as shown in the following example.

The Code

```
<?php
// Example 3-1-7.php
echo intval('123', 10) . "\n";
echo intval('101010', 2) . "\n";
echo intval('123', 8) . "\n";
echo intval('123', 16) . "\n";

echo intval('H123', 32) . "\n";
echo intval('H123', 36) . "\n";
?>
```

How It Works

This example takes numeric values with different bases and converts them to decimal representations (base 10). A decimal value uses the digits 0123456789, a binary value uses the digits 01, an octal value uses the digits 01234567, and a hexadecimal value uses the digits 0123456789abcdef. The digits for base 32 and base 36 are 0123456789 and the first 22 or 26 letters of the alphabet. So, the value `H123` does not denote a hexadecimal value.

```
123
42
83
291
558147
794523
```

The `intval()` function will also work on boolean and float types, returning the integer value. The integer value of a boolean variable is 0 for `false` and 1 for `true`. For a float value, this function will truncate the value at the decimal point.

When working with integers, it is sometimes necessary to convert between different base values. The PHP interpreter will accept integers as part of the script, in decimal, octal, and hexadecimal form, and automatically convert these to the internal decimal representation. Using the octal and hexadecimal forms can make the code more readable. You can use the octal form when setting file permissions, as this is the notation used on most Unix and Unix-like systems, and you can use the hexadecimal form when defining constants where you need to have a single bit set in each constant.

```
<?php
// Example 3-1-8.php
chmod("/mydir", 0755);

define('VALUE_1', 0x001);
define('VALUE_2', 0x002);
define('VALUE_3', 0x004);
define('VALUE_4', 0x008);
define('VALUE_5', 0x010);
define('VALUE_6', 0x020);
?>
```

It is easier to read and define constants based on single bits when using the hexadecimal representation, where each digit represents 4 bits, than when using with decimal representation, where the same values would be 1, 2, 4, 8, 16, and 32.

Sometimes it's also useful to convert integer values to other bases such as binary or base 32 and base 36, as used in the previous example. You can use the function `base_convert()` to convert any integer value from one base to another. The function takes one numeric and two integer parameters, where the first parameter is the number to be converted. This value can be an integer or a string with a numeric representation. The second parameter is the base to convert from, and the third parameter is the base to convert to. The function will always return a string value, even if the result is an integer in the decimal representation.

The Code

```
<?php
// Example 3-1-9.php
echo base_convert('123', 10, 10) . "\n";
echo base_convert('42', 10, 2) . "\n";
echo base_convert('83', 10, 8) . "\n";
echo base_convert('291', 10, 16) . "\n";

echo base_convert('558147', 10, 32) . "\n";
echo base_convert('794523', 10, 36) . "\n";

echo base_convert('abcd', 16, 8) . "\n";
echo base_convert('abcd', 16, 2) . "\n";?>
```

How It Works

In this example, you saw the same values as in the previous example, but this example uses the `base_convert()` function to do the reverse conversion. In addition, this example also shows conversions between bases other than the decimal 10.

```
123
101010
123
123
h123
h123
125715
1010101111001101
```

Remember that the maximum width of an integer value in PHP is 32-bit. If you need to convert integer values with more than 32 bits, you can use the GMP extension (see recipe 3-6).

You can assign a value to a variable in a few ways in PHP (see Chapter 10). The most basic form is the assignment `$a = 10;`, where `$a` is given the integer value 10. If the variable exists, the old value will be lost, and if the variable is used for the first time, the internal structure will be allocated. There is no need to declare variables before use, and any variable can be reassigned to another value with another type at any time.

For variables of a numeric type, it is also possible to assign a new value and at the same time use the existing value in the calculation of the new value. You do this with `$a += 5;`, where the new value of `$a` will be the old value plus 5. If `$a` is unassigned at the time the statement is executed, the engine will generate a warning and assume the old value of 0 before calculating the new value.

Tables 3-1, 3-2, and 3-3 show the arithmetic, bitwise, and assignment operators that are available in PHP.

Table 3-1. *Arithmetic Operators*

Example	Operation	Result
`-$a`	Negation	Negative value of `$a`
`$a + $b`	Addition	Sum of `$a` and `$b`
`$a - $b`	Subtraction	Difference of `$a` and `$b`
`$a * $b`	Multiplication	Product of `$a` and `$b`
`$a / $b`	Division	Quotient of `$a` and `$b`
`$a % $b`	Modulus	Remainder of `$a` divided by `$b`

Table 3-2. *Bitwise Operators*

Example	Operation	Result
`$a & $b`	And	Bits that are set in both `$a` and `$b` are set.
`$a | $b`	Or	Bits that are set in either `$a` or `$b` are set.
`$a ^ $b`	Xor	Bits that are set in $`$a` or `$b` but not in both are set.
`~ $a`	Not	Bits that are set in `$a` are not set, and vice versa.
`$a << $b`	Shift left	Shift the bits of `$a` to the left `$b` steps.
`$a >> $b`	Shift right	Shift the bits of `$a` to the right `$b` steps.

Table 3-3. *Assignment Operators*

Example	Operation	Result
`$a += $b`	Addition	`$a = $a + $b`
`$a -= $b`	Subtraction	`$a = $a - $b`
`$a *= $b`	Multiplication	`$a = $a * $b`
`$a /= $b`	Division	`$a = $a / $b`
`$a %= $b`	Modulus	`$a = $a % $b`
`$a &= $b`	Bitwise and	`$a = $a & $b`
`$a |= $b`	Bitwise or	`$a = $a | $b`
`$a ^= $b`	Bitwise xor	`$a = $a ^ $b`
`$a <<= $b`	Left-shift	`$a = $a << $b`
`$a >>= $b`	Right-shift	`$a = $a >> $b`

■ **Note** If you use bitwise operators on strings, the system will apply the operation on the string character by character. For example, `123 & 512` equals 102. First, the values 1 and 5 are "anded" together in binary terms, that is, 001 and 101; only the last bit is common, so the first character becomes 001. The next two values are 2 and 1 (or in binary values, 10 and 01). These two values are "anded" together to make 00, or 0. And finally, 3 and 2 are "anded" together, so that's 11 and 10 with the result of 10, or 2. So, the resulting string is 102.

Integer values can be signed and unsigned in the range from –2,147,483,648 to 2,147,483,647. If a calculation on any integer value causes the result to get outside these boundaries, the type will automatically change to float, as shown in the following example.

The Code

```
<?php
// Example 3-1-10.php
$i = 0x7FFFFFFF;
echo "$i is " . (is_int($i) ? "an integer" : "a float") . "\n";
$i++;
echo "$i is " . (is_int($i) ? "an integer" : "a float") . "\n";
?>
```

How It Works

The variable `$i` is assigned a value corresponding to the largest integer number PHP can handle, and then `is_int()` is called to verify that `$i` is an integer. Then the value of `$i` is incremented by 1, and the same check is performed again.

```
2147483647 is an integer
2147483648 is a float
```

In other languages with strict type handling, the increment of `$i` by 1 would lead to overflow, and the result would be a negative value of –2,147,483,648.

Comparing integer values is simple and exact because there is a limited number of values and each value is well defined. This is not the case with floating-point values, where the precision is limited. Comparing two integers with = will result in a `true` value if the two integers are the same and `false` if they are different. This is not always the case with floating-point values. These are often looked at with a numeric string representation that might change during processing. The following example shows that a simple addition of two floating-point variables compared to a variable with the expected value result can lead to an unexpected result.

The Code

```
<?php
// Example 3-1-11.php
$a=50.3;
$b=50.4;
$c=100.7;

if ($a + $b == $c) {
  echo "$a + $b == $c\n";
}
else {
  echo "$a + $b != $c\n";
}
?>
```

How It Works

Three variables are each assigned a floating-point value; then, a calculation is performed with the two first values, and the result is compared to the last value. One would expect that the output from this code would indicate that `$a + $b == $c`.

```
50.3 + 50.4 != 100.7
```

This result indicates that PHP is having trouble with simple floating-point operations, but you will find the same result in other languages. It is possible to compare floating-point values, but you should avoid the == operator and use the <, >, >=, and <= operators instead. In the following example, the loop goes from 0 to 100 in steps of 0.1, and the two checks inside the loop print a line of text when `$i` reaches the value 50.

The Code

```
<?php
// Example 3-1-12.php
for ($i = 0; $i < 100; $i += 0.1) {
  if ($i == 50) echo '$i == 50' . "\n";
  if ($i >= 50 && $i < 50.1) echo '$i >= 50 && $i < 50.1' . "\n";
}
?>
```

How It Works

This code creates a loop where the variable `$i` starts as an integer with the value 0. The code in the loop is executed as long as the value of `$i` is less than 100. After each run-through, the value of `$i` is incremented by 0.1. So, after the first time, `$i` changes to a floating-point value. The code in the loop uses two different methods to compare the value of `$i`, and, as the result shows, only the second line is printed.

```
$i >= 50 && $i < 50.1
```

Another way to make sure the values are compared correctly is to typecast both sides of the == operator to integers like this: `if ((int)$i == 50) echo '$i == 50' . "\n";`. This will force the engine to compare two integer values, and the result will be as expected.

3-2. Random Numbers

Random numbers are almost a science by themselves. Many different implementations of random number generators exist, and PHP implements two of them: `rand()` and `mt_rand()`. The `rand()` function is a simple wrapper for the random function that is defined in `libc` (one of the basic libraries provided by the compiler used to build PHP). `mt_rand()` is a drop-in replacement with well-defined characteristics (Mersenne Twister), and `mt_rand()` is even much faster than the version from `libc`.

Working with random number generation often requires seeding the generator to avoid generating the same random number each time the program is executed. This is also the case for PHP, but since version 4.2.0, this seeding takes place automatically. It is still possible to use the `srand()` and `mt_srand()` functions to seed the generators, but it's no longer required.

You can use both random generators with no arguments or with two arguments. If no arguments are passed, the functions will return an integer value between 0 and `RAND_MAX`, where `RAND_MAX` is a constant defined by the C compilers used to generate PHP. If two arguments are passed, these will be used as the minimum and maximum values, and the functions will return a random value between these two numbers, both inclusive.

Both random generators provide functions to get the value of `MAX_RAND`. The next example shows how to use these functions.

The Code

```
<?php
// Example 3-2-1.php
echo "getrandmax = " . getrandmax() . "\n";
echo "mt_getrandmax = " . mt_getrandmax() . "\n";
?>
```

How It Works

On a Linux or Unix system, this sample code produces this output:

```
Getrandmax() = 2147483647
mt_getrandmax() = 2147483647
```

On a Windows system, the same code produces this output:

```
Getrandmax() = 32767
mt_getrandmax() = 2147483647
```

This difference is caused by the different `libc` implementations of the random number generators and the `MAX_RAND` value between different platforms.

You can generate random numbers (integer values) between 0 and `MAX_RAND` by calling `rand()` or `mt_rand()` without any arguments, as shown in the next example.

The Code

```
<?php
// Example 3-2-2.php
echo "rand() = " . rand() . "\n";
echo "mt_rand() = " . mt_rand() . "\n";
?>
```

How It Works

On Windows and Linux systems, the output from this code would look like the following, though the values will be different each time the script is executed:

```
rand() = 9189
mt_rand() = 1101277682
```

In many cases, it's required to get a random value with other minimum and maximum values than the default. This is where the two optional arguments are used. The first argument specifies the minimum value, and the second specifies the maximum value. The following example shows how to get a random value from 5 to 25.

The Code

```
<?php
// Example 3-2-3.php
echo "rand(5, 25) = " . rand(5, 25) . "\n";
echo "mt_rand(5, 25) = " . mt_rand(5, 25) . "\n";
?>
```

How It Works

This example prints two random values from 5 to 25.

```
rand(5, 25) = 8
mt_rand(5, 25) = 6
```

Random values are not restricted to positive integers. The following example shows how to get random values from –10 to 10.

The Code

```
<?php
// Example 3-2-4.php
echo "rand(-10, 10) = " . rand(-10, 10) . "\n";
echo "mt_rand(-10, 10) = " . mt_rand(-10, 10) . "\n";
?>
```

How It Works

This example prints two random values between –10 and 10.

```
rand(-10, 10) = 5
mt_rand(-10, 10) = -6
```

Generating random numbers with these two functions will always result in an integer value. With some simple math it is possible to change this to generate a random floating-point value. So, if you want to generate random floating-point values from 0 to 10 with two decimals, you could write a function called `frand()`, as shown next.

The Code

```
<?php
// Example 3-2-5.php
function frand($min, $max, $decimals = 0) {
  $scale = pow(10, $decimals);
  return mt_rand($min * $scale, $max * $scale) / $scale;
}

echo "frand(0, 10, 2) = " . frand(0, 10, 2) . "\n";
?>
```

How It Works

The function takes two mandatory arguments and one optional argument. If the third argument is omitted, the function will work as `mt_rand()` and return an integer. When the third argument is given, the function calculates a scale value used to calculate new values for the minimum and maximum and to adjust the result from `mt_rand()` to a floating-point value within the range specified by `$min` and `$max`. The output from this sample looks like this:

```
frand(0, 10, 2) = 3.47
```

Working with currency values might require the generation of random numbers with fixed spacing. Generating random values between $0 and $10 and in steps of $0.25 would not be possible with the `frand()` function without a few modifications. By changing the third parameters from `$decimals` to `$precision` and changing the logic a bit, it is possible to generate random numbers that fit both models, as shown in the following example.

The Code

```
<?php
// Example 3-2-6.php
function frand($min, $max, $precision = 1) {
  $scale = 1/$precision;
  return mt_rand($min * $scale, $max * $scale) / $scale;
}

echo "frand(0, 10, 0.25) = " . frand(0, 10, 0.25) . "\n";
?>
```

■ **Note** There are no checks on the `$precision` value. Setting `$precision = 0` will cause a division-by-zero error.

How It Works

The output from the sample looks like this:

```
frand(0, 10, 0.25) = 3.25
```

Changing the precision parameter to 0.01 gives the same result as in the first example, and changing it to 3 causes the function to return random values between 0 and 10 in steps of 3. The possible values are 0, 3, 6, and 9, as shown in the next example.

The Code

```
<?php
// Example 3-2-7.php
function frand($min, $max, $precision = 1) {
  $scale = 1/$precision;
  return mt_rand($min * $scale, $max * $scale) / $scale;
}

echo "frand(0, 10, 3) = " . frand(0, 10, 3) . "\n";
?>
```

How It Works

The precision parameter has been changed to 3, so the ||$scale value will be 1/3. This reduces the internal minimum and maximum values to 0 and 3, and the result of mt_rand() is divided by $scale, which is the same as multiplying by 3. The internal random value will be 0, 1, 2, or 3, and when that's multiplied by 3, the possible values are 0, 3, 6, or 9.

```
frand(0, 10, 3) = 6
```

■ **Note** The arguments to mt_rand() are expected to be integers. If other types are passed, the values are converted to integers before calculating the random value. This might cause the minimum and maximum values to be truncated, if the calculation of $min * $scale or $max * $scale results in a floating-point value.

You can also use the random number generators to generate random strings. This can be useful for generating passwords. The next example defines a function called GeneratePassword() that takes two optional arguments. These arguments specify the minimum and maximum lengths of the generated password.

The Code

```
<?php
// Example 3-2-8.php
function GeneratePassword($min = 5, $max = 8) {
  $ValidChars = "abcdefghijklmnopqrstuvwxyz123456789";
  $max_char = strlen($ValidChars) - 1;
  $length = mt_rand($min, $max);
  $password = "";
  for ($i = 0; $i < $length; $i++) {
    $password .= $ValidChars[mt_rand(0, $max_char)];
  }
  return $password;
}

echo "New Password = " . GeneratePassword() . "\n";
echo "New Password = " . GeneratePassword(4, 10) . "\n";
?>
```

How It Works

The output from this script could look like this:

```
New Password = bbeyq
New Password = h3igij3bd7
```

The `mt_rand()` function is first used to get the length of the new password and then used within the `for` loop to select each character randomly from a predefined string of characters. You could extend this string to include both uppercase and lowercase characters and other characters that might be valid.

The variable `$max_char` defines the upper limit of the random number generation. This is set to the length of the string of valid characters minus 1 to avoid the `mt_rand()` function from returning a value that is outside the string's boundaries.

3-3. Logarithms and Exponents

PHP implements the `log()`, `log10()`, `exp()`, and `pow()` functions, as well as `logp1()` and `expm1()` that are marked as experimental, to calculate logarithms and exponents.

The `exp()` and `log()` functions are considered to be the inverse of each other, and they use `e` as the base. This number is called *neperian*, or the natural logarithm base. `$e = exp(1);` gives the value of `e`, and it's equal to 2.71828182846. This number is also defined as a constant called `M_E`, and it's defined as 2.7182818284590452354.

The following example shows the calculation of `e` and the inverse nature of the two functions.

The Code

```
<?php
// Example 3-3-1.php
$e = exp(1);
echo "$e\n";
$i = log($e);
echo "$i\n";
?>
```

How It Works

This example calculates the value of `e` and assigns it to the variable `$e`, which is printed before it is used as a parameter to the `log()` function.

```
2.71828182846
1
```

You can calculate logarithms with other base values by dividing the result of the `log()` function with `log(base)`. If the base is 10, it is faster to use the built-in `log10()` function, but for all other values of base, you can use this method.

The Code

```
<?php
// Example 3-3-2.php
$val = 100;
$i = log($val);
echo "log($val) = $i\n";
$i10 = log($val) / log(10);
echo "log($val) / log(10) = $i10\n";
$i10 = log10($val);
echo "log10($val) = $i10\n";
?>
```

How It Works

This example calculates the natural logarithm of 10 and prints it. Then it uses the nature of the logarithmic functions to calculate `log10` of the same value, and at last it uses the building `log10()` function to verify the result. The output from this example looks like this:

```
log(100) = 4.60517018599
log(100) / log(10) = 2
log10(100) = 2
```

The `pow()` function calculates one number to the power of another number. The function takes two arguments, where the first is the base and the second is the exponent. The return value will be an integer, if possible, or a float value. In the case of an error, the return value will be `FALSE`. When the base value is `e`, the `pow()` function becomes equal to the `exp()` function.

■ **Note** PHP cannot handle negative base values if the exponent is a noninteger.

The following example shows how to use the `pow()` function with integers, floats, and both negative and positive numbers.

The Code

```
<?php
// Example 3-3-3.php
echo pow(2, 8) . "\n";
echo pow(-2, 5) . "\n";
echo pow(-2.5, 5) . "\n";
echo pow(0, 0) . "\n";
echo pow(M_E, 1) . "\n";
echo pow(3.2, 4.5) . "\n";
echo pow(2, -2) . "\n";
echo pow(-2, -3) . "\n";
?>
```

How It Works

This example shows how the `pow()` function can calculate the power of both positive and negative values of both integer and floating-point types.

```
256
-32
-97.65625
1
2.71828182846
187.574977246
0.25
-0.125
```

Another special case of `pow()` is that when the exponent is 0.5, the function is equal to the `sqrt()` function.

When presenting data in a graphical form (bar or line charts), it is sometimes practical to use a logarithmic scale to avoid small values being too close to the X axis and to reduce the visual difference between small and large values.

The next example uses a simple HTML-based technology to draw bar charts from an array of data. The script defines two constants used by the `ShowChart()` function to select a linear or logarithmic scale when drawing the chart. The `ShowChart()` function takes three arguments, where the first is the array of data used to draw the chart, the second is the optional chart type, and the third is an optional height value used to calculate the scaling of the data. In this case, the data used is hard-coded, but this part of the script could use a database connection or a log file from a web server to fetch the data. The final part of the script is where the HTML document is created and sent to the client.

The Code

```
<?php
// Example 3-3-4.php
define('BAR_LIN', 1);
define('BAR_LOG', 2);

function ShowChart($arrData, $iType = BAR_LIN, $iHeight = 200) {
  echo '<table border=0><tr>';

  $max = 0;
  foreach($arrData as $y) {
    if ($iType == BAR_LOG) {
      $y = log10($y);
    }
    if ($y > $max) $max = $y;
  }
  $scale = $iHeight / $max;
```

```
  foreach($arrData as $x=>$y) {
    if ($iType == BAR_LOG) {
      $y = log10($y);
    }
    $y = (int)($y*$scale);
    echo "<td valign=bottom>
            <img src=dot.png width=10 height=$y>
          </td>
          <td width=5> </td>";
  }
  echo '</tr></table>';
}

$arrData = array(
  150,
  5,
  200,
  8,
  170,
  50,
  3
);

echo '<html><body>';

echo 'Show chart with linear scale';
ShowChart($arrData, BAR_LIN);

echo '<br>Show chart with logarithmic scale';
ShowChart($arrData, BAR_LOG);

echo '</body></html>';
?>
```

How It Works

The `ShowChart()` function uses a small image of 1×1 pixels to generate the bars. Each bar is represented with the image being scaled to a height and width that matches the data in the first array passed to the function. The second parameter selects linear or logarithmic scale, and the third parameter defines the height of the entire chart. Figure 3-1 shows the resulting charts with linear and logarithmic scale.

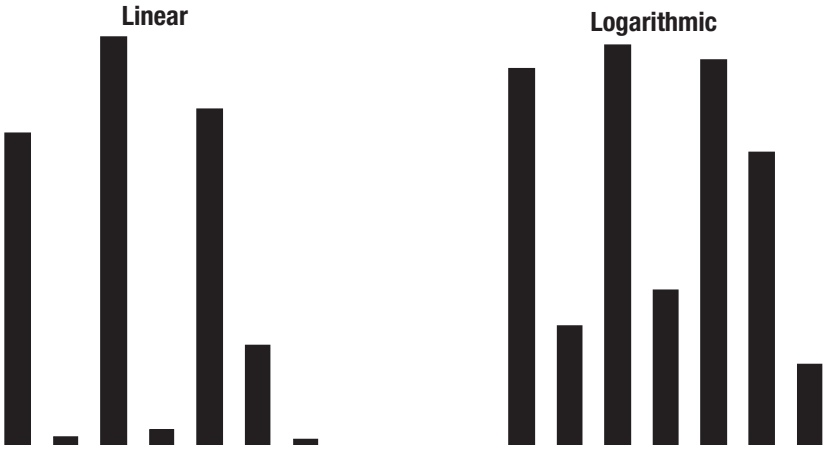

Figure 3-1. *Sample bar charts with linear and logarithmic scale*

Using plain HTML to generate charts is not optimal because of the limitations of the markup language. It's possible to generate more advanced charts with the GD (GIF, PNG, or JPG images) and Ming (Flash movies) extensions. Figure 3-2 shows an example of a bar chart generated with the Ming extension.

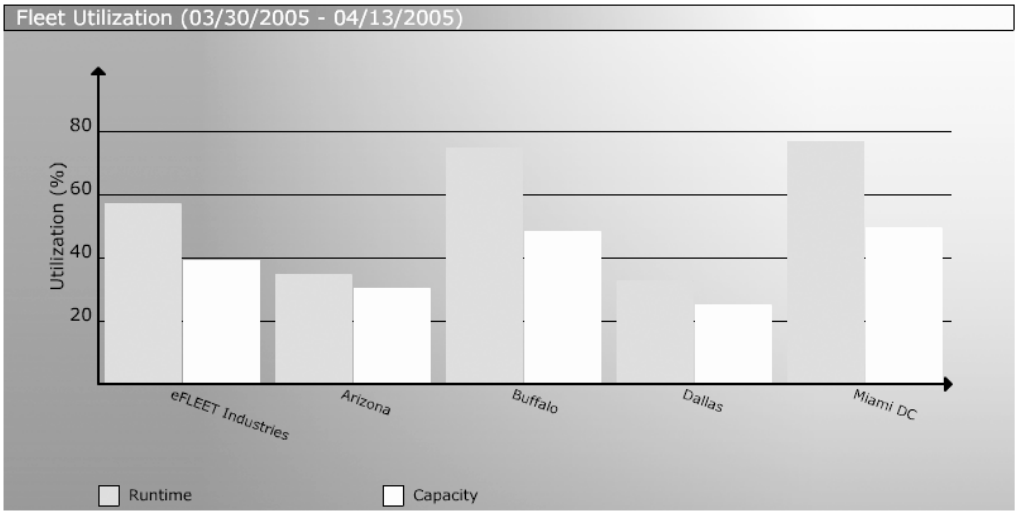

Figure 3-2. *Bar chart generated with the Ming extension*

3-4. Trigonometric Functions

PHP implements a full set of trigonometric and hyperbolic functions as well as a few functions to convert from degrees to radians and back. A number of constants, including `M_PI` (3.1415926538979323846), and a few derivatives are also defined to make life easier for the developer, as described in Table 3-4 and Table 3-5.

Table 3-4. *Trigonometric Functions*

Name	Description
`cos()`	Cosine
`sin()`	Sine
`tan()`	Tangent
`acos()`	Arc cosine
`asin()`	Arc sine
`atan()`	Arc tangent
`atan2()`	Arc tangent of two variables
`pi()`	A function returning pi (the same as `M_PI`)
`deg2rad()`	Degree to radians
`rad2deg()`	Radians to degrees

Table 3-5. *Hyperbolic Functions*

Name	Description
`cosh()`	Hyperbolic cosine `(exp(arg) + exp(-arg))/2`
`sinh()`	Hyperbolic sine `(exp(arg) - exp(-arg))/2`
`tanh()`	Hyperbolic tangent `sinh(arg)/cosh(arg)`
`acosh()`	Inverse hyperbolic cosine
`asinh()`	Inverse hyperbolic sine
`atanh()`	Inverse hyperbolic tangent

■ **Note** `acosh()`, `asinh()`, and `atanh()` are not implemented on the Windows platform.

You can use the trigonometric functions to calculate positions of elements in a plane. This can be useful when using GD or Ming extensions to generate dynamic graphical content. If a line that starts in (0, 0) and ends in (100, 0) is to be duplicated starting at (20, 20) and rotated 35 degrees, you can calculate the ending point with the trigonometric functions, as shown in the following example.

The Code

```
<?php
// Example 3-4-1.php
$start = array(0, 0);
$end = array(100, 0);
$length = sqrt(pow($end[0] - $start[0], 2) + pow($end[1] - $start[1], 2));

$angle = 35;
$r = deg2rad($angle);
$new_start = array(20, 20);
$new_end = array(
  $new_start[0] + cos($r) * $length,
  $new_start[1] + sin($r) * $length
);

var_dump($new_end);
?>
```

How It Works

The first line is defined by two sets of coordinates. These are assigned as arrays to the variables `$start` and `$end` and used to calculate the length of the line. Setting the starting point of the new line is as simple as assigning `$new_start` the coordinates as an array. This is then used together with the angle (35 degrees) to calculate the value of `$new_end`.

```
array(2) {
  [0]=>
  float(101.915204429)
  [1]=>
  float(77.3576436351)
}
```

The arguments to `cos()`, `sin()`, and `tan()` should always be given in radians, and the values returned by `acos()`, `asin()`, and `atan()` will always be in radians. If you need to operate on angles specified in degrees, you can use the `deg2rad()` and `rad2deg()` functions to convert between the two.

Trigonometric functions have a wide range of usages; one of them is to calculate the distance between two locations on the earth. Each location is specified by a set of coordinates. Several different methods with varying accuracy are available, and they can more or less compensate for the fact that the earth is not a perfect sphere. One of the simplest methods is called the Great Circle Distance, and it's based on the assumptions that 1 minute of arc is 1 nautical mile and the radius of the earth is 6,364.963 kilometers (3,955.00465 miles). These assumptions work fine when both locations are far from the poles and equator.

The formula used to calculate the distance takes the longitude and latitude for each location, and it looks like this:

```
D = R * ARCOS (SIN(L1) * SIN(L2) + COS(L1) * COS(L2) * COS(DG))
```

This formula returns the distance in kilometers or miles (depending on the radius value) and assumes all the trigonometric functions to be working in degrees. For most calculators it is possible to choose degrees or radians, but for PHP only radians is available, so you need to convert everything to and from degrees. It would also be nice to have the function return the result in miles or kilometers.

`R` is the earth's radius in kilometers or miles, `L1` and `L2` are the latitude of the first and second locations in degrees, and `DG` is the longitude of the second location minus the longitude of the first location, also in degrees. Latitude values are negative south of the equator. Longitudes are negative to the west with the center being the Greenwich mean time (GMT) line.

Calculating the distance between two locations starts by finding the longitude and latitude of each location and inserting the values in the formula. A Google search is an easy way to find longitude and latitude values for many popular locations. You can use maps and even Global Positioning System (GPS) receivers. As an example, we have chosen Copenhagen and Los Angeles. Copenhagen is located east of the GMT line at 12.56951 and north of the equator at 55.67621, and Los Angeles is located west of the GMT line at –118.37323 and a bit closer to the equator at 34.01241.

To make the calculations a little easier, you can start by creating a function that will return the distance between two locations in either kilometers or miles. The function is called `GetDistance()`; it takes four mandatory parameters and one optional parameter. The two constants (`KM` and `MILES`) select the format of the return value as well as define the earth's radius in both formats.

The Code

```
<?php
// Example 3-4-2.php
define('KM', 6364.963);
define('MILES', 3955.00465);

function GetDistance($la1, $lo1, $la2, $lo2, $r = KM) {
  $l1 = deg2rad($la1);
  $l2 = deg2rad($la2);
  $dg = deg2rad($lo2 - $lo1);
  $d = $r * acos(sin($l1) * sin($l2) + cos($l1) * cos($l2) * cos($dg));
  return $d;
}

// Copenhagen
$lat1 = 55.67621;
$long1 = 12.56951;
```

```
// Los Angeles
$lat2 =  34.01241;
$long2 = -118.37323;

echo "The distance from Copenhagen to Los Angeles is " .
      round(GetDistance($lat1, $long1, $lat2, $long2)) . " km\n";
echo "The distance from Copenhagen to Los Angeles is " .
      round(GetDistance($lat1, $long1, $lat2, $long2, MILES)) . " miles\n";
?>
```

How It Works

Two constants define the radius of the earth in kilometers and miles. The same two constants are used as parameters to the `GetDistance()` function, so there is no need for additional constants here. The `GetDistance()` function takes four mandatory parameters that specify the latitude and longitude of each point for which the distance should be calculated.

The `round()` function is used on the return value, before printing, to get rid of any decimals, because the calculation is not that accurate anyway. The output from the script is the distance between Copenhagen and Los Angeles in kilometers and in miles:

```
The distance from Copenhagen to Los Angeles is 9003 km
The distance from Copenhagen to Los Angeles is 5594 miles
```

3-5. Formatting of Numeric Data

Except for warnings, errors, and so on, most output from PHP is generated by a few functions, such as `echo`, `print()`, and `printf()`. These functions convert the argument to a string and send it to the client (console or web browser). The PHP-GTK extension uses other methods/functions to generate output. You can use the `sprintf()` function in the same way as the `printf()` function, except it returns the formatted string for further processing. The conversion of numbers to string representation takes place in a simple way without any special formatting, except for a few options used with the `|printf()` function. It is possible to embed integer and floating-point values in strings for easy printing, as shown in the following sample.

The Code

```
<?php
// Example 3-5-1.php
$i = 123;
$f = 12.567;

echo "\$i = $i and \$f = $f\n";
?>
```

How It Works

Two numeric variables are defined and assigned an integer and a floating-point value. The two variables are then embedded in a string. This generates the following output:

```
$i = 123 and $f = 12.567
```

Other functions can format numeric values before the value is output to the client. You can convert an integer into a string representation with a different base using one of these functions: `decbin()`, `decoct()`, `dechex()`, or `base_convert()`. The `base_convert()` function can convert an integer to any base, as you saw in recipe 3-1, but the first three functions make the code a bit more readable, and there is no need for additional parameters. Three functions—`bindec()`, `octdec()`, and `hexdec()`—can convert binary, octal, and hexadecimal strings to decimal integer values; again, these conversions can be handled by `base_convert()`, but the result will be a string value for any conversion, where the three other functions will return an integer or a float depending on the number of bits needed to represent the number.

When decimal numbers (integers or floats) are presented, it's common to use a decimal point, a thousand separator, and a fixed number of decimals after the decimal point. This makes it much easier to read the value when it contains many digits. In PHP the function `number_format()` converts integers and floating-point values into a readable string representation. The function takes one, two, or four parameters. The first parameter is the numeric value to be formatted. This is expected to be a floating-point value, but the function allows it to be an integer or a string and performs the conversion to a float when needed.

■ **Note** If a non-numeric string value is passed as the first parameter, the internal conversion will result in 0. No warnings or errors will be generated.

The second parameter indicates the number of decimals after the decimal point. The default number of decimals is zero. The third and fourth parameters specify the character for the decimal point and thousand separator. The default values are a dot (`.`) and a comma (`,`), but you can change to any character. The following example shows how you can format an integer and a floating-point value with the `number_format()` function.

The Code

```
<?php
// Example 3-5-2.php
$i = 123456;
$f = 98765.567;

$si = number_format($i, 0, ',', '.');
$sf = number_format($f, 2);

echo "\$si = $si and \$sf = $sf\n";
?>
```

How It Works

Two floating-point values are defined and formatted with the `number_format()` function. The first value is presented as an integer with zero decimals, the decimal point is represented with a comma (not shown), and the thousand separator is a dot. The second value is formatted with two decimals and uses the system default for the decimal point and thousand separator.

```
$si = 123.456 and $sf = 98,765.57
```

You can use two other functions to format numbers: `printf()` and `sprintf()`. Both functions take one or more arguments, where the first argument is a string that describes the format and the remaining arguments replace placeholders defined in the formatting string with values. The main difference between the two functions is the way the output is handled. The `printf()` function sends the output directly to the client and returns the length of the printed string; `sprintf()` returns the string to the program. Both functions follow the same formatting rules, where `%` followed by a letter indicates a placeholder. Table 3-6 lists the allowed placeholders.

Table 3-6. `printf()` *and* `sprintf()` *Formatting Types*

Type	Description
`%`	A literal percent character. No argument is required.
`b`	The argument is treated as an integer and presented as a binary number.
`c`	The argument is treated as an integer and presented as the character with that American Standard Code for Information Interchange (ASCII) value.
`d`	The argument is treated as an integer and presented as a (signed) decimal number.
`e`	The argument is treated as scientific notation (for example, 1.2e+2).
`|u`	The argument is treated as an integer and presented as an unsigned decimal number.
`f`	The argument is treated as a float and presented as a floating-point number (locale aware).
`F`	The argument is treated as a float and presented as a floating-point number (nonlocale aware). Available since PHP 4.3.10 and PHP 5.0.3.
`o`	The argument is treated as an integer and presented as an octal number.
`s`	The argument is treated and presented as a string.
`x`	The argument is treated as an integer and presented as a hexadecimal number (with lowercase letters).
`X`	The argument is treated as an integer and presented as a hexadecimal number (with uppercase letters).

The following example shows how you can format an integer and a floating-point value with the `printf()` function.

The Code

```
<?php
// Example 3-5-3.php
$i = 123456;
$f = 98765.567;

printf("\$i = %x and \$i = %b\n", $i, $i);
printf("\$i = %d and \$f = %f\n", $i, $f);
printf("\$i = %09d and \$f = %0.2f\n", $i, $f);
?>
```

How It Works

This example shows how the `printf()` function can format numbers as different data types.

```
$i = 1E240 and $i = 111100010010000000
$i = 123456 and $f = 98765.567000
$i = 000123456 and $f = 98765.57
```

It is also possible to use typecasting to convert numbers to strings; this works as if the variable were embedded in a string, as shown in the next example.

The Code

```
<?php
// Example 3-5-4.php
$i = 123456;
$f = 98765.567;

echo "\$i = " . (string)$i . "\n";
echo "\$f = " . (string)$f . "\n";
?>
```

How It Works

The two variables are typecast into a string value and used to generate the output.

```
$i = 123456
$f = 98765.567
```

On systems where libc implements the function strfmon(), PHP will also define a function called money_format(). The function takes a formatting string and a floating-point number as arguments. The result of this function depends on the setting of the LC_MONETARY category of the locale settings. You can change this value with the setlocale() function before calling money_format(). This function can convert only one floating-point value at the time, and the formatting string can contain one placeholder along with other characters that will be returned with the formatted number.

The placeholder is defined as a sequence of the following elements:

- The % character indicates the beginning of the placeholder.
- Optional flags.
- Optional width.
- Optional left precision.
- Optional right precision.
- A conversion character.

The following example shows a few ways of formatting currency values and shows how to change the locale setting before showing a money value, as well as a few other formatting options.

The Code

```
<?php
// Example 3-5-5.php
$number = 1234.56;
setlocale(LC_MONETARY, 'en_US');
echo money_format('%i', $number) . "\n";

setlocale(LC_MONETARY, 'en_DK');
echo money_format('%.2i', $number) . "\n";

$number = -1234.5672;
setlocale(LC_MONETARY, 'en_US');
echo money_format('%(#10n', $number) . "\n";
echo money_format('%(#10i', $number) . "\n";
?>
```

How It Works

A floating-point value is passed to the money_format() function. Before each call to this function, the LC_MONETARY value is changed by a call to the setlocale() function.

```
USD 1,234.56
DKK 1.234,56
($        1,234.57)
(USD         1,234.57)
```

3-6. Math Libraries

PHP comes with two math extensions: BCMath and GMP. BCMath is a binary calculator that supports numbers of any size and precision. This extension is bundled with PHP. (It's compiled by default on Windows systems; on Unix systems it can be enabled with the `-enable-bcmath` configure option.) There is no need for external libraries. The GMP extension is a wrapper around the GNU MP library, and it allows you to work with arbitrary-length integers. This extension requires the GNU library and can be included by adding `-with-gmp` when configuring PHP. (For binary Windows distributions this will be included in the `php_gmp.dll` file). Table 3-7 shows the functions implemented by the BCMath extension.

Table 3-7. *BCMath Functions*

Name	Description
`bcadd()`	Adds two numbers
`bccomp()`	Compares two numbers
`|bcdiv()`	Divides two numbers
`bcmod()`	Calculates the remainder with the division of two numbers
`bcmul()`	Multiplies two numbers
`bcpow()`	Raises one number to the power of another
`bcpowmod()`	Raises one number to the power of another, raised by the specified modulus
`bcscale()`	Sets the default scale for all BCMath functions
`bcsqrt()`	Calculates the square root of a number
`bcsub()`	Subtracts two numbers

Most of these functions take an optional scale parameter. If the scale parameter is omitted, the functions will use the value defined by a call to `bcscale()`. The scale parameter defines the number of decimals returned by the function, as shown in the following example.

The Code

```
<?php
// Example 3-6-1.php
bcscale(3);
$a = 1.123;
$b = 2.345;

$c = bcadd($a, $b);
echo "$c\n";

$c = bcadd($a, $b, 1);
echo "$c\n";
?>
```

How It Works

Two floating-point values are defined and added with the `bcadd()` function using the default scale (3) set by a call to `bcscae()`. Then the same two values are added, but this time the default scale is overwritten by the third argument. Note how the result is truncated and not rounded.

```
3.468
3.4
```

The GMP extension implements a long list of functions (see Table 3-8) that can be used to manipulate large integer values (more than 32 bits).

Table 3-8. *GMP Functions*

Name	Description
`gmp_abs`	Calculates absolute value
`gmp_add`	Adds numbers
`|gmp_and`	Logical and
`gmp_clrbit`	Clears bit
`gmp_cmp`	Compares numbers
`gmp_com`	Calculates one's complement
`|gmp_div_q`	Divides numbers
`gmp_div_qr`	Divides numbers and gets quotient and remainder
`gmp_div_r`	Remainder of the division of numbers
`gmp_div`	Alias of `gmp_div_q()`
`gmp_divexact`	Exact division of numbers
`gmp_fact`	Factorial
`gmp_gcd`	Calculates GCD
`gmp_gcdext`	Calculates GCD and multipliers
`gmp_hamdist`	Hamming distance
`gmp_init`	Creates GMP number
`gmp_intval`	Converts GMP number to integer
`gmp_invert`	Inverse by modulo
`gmp_jacobi`	Jacobi symbol
`gmp_legendre`	Legendre symbol
`gmp_mod`	Modulo operation
`gmp_mul`	Multiplies numbers
`gmp_neg`	Negates number
`gmp_or`	Logical or
`gmp_perfect_square`	Perfect square check
`gmp_popcount`	Population count

Name	Description
gmp_pow	Raises number into power
gmp_powm	Raises number into power with modulo
gmp_prob_prime	Checks if number is "probably prime"
gmp_random	Random number
gmp_scan0	Scans for 0
gmp_scan1	Scans for 1
gmp_setbit	Sets bit
gmp_sign	Sign of number
gmp_sqrt	Calculates square root
gmp_sqrtrem	Square root with remainder
gmp_strval	Converts GMP number to string
gmp_sub	Subtracts numbers
gmp_xor	Logical xor

The following is an alternative to base_convert() that works on integers up to 32-bit.

The Code

```
<?php
// Example 3-6-2.php
if (!extension_loaded("gmp")) {
  dl("php_gmp.dll");
}
/*use gmp library to convert base. gmp will convert numbers > 32bit*/
function gmp_convert($num, $base_a, $base_b)
{
       return gmp_strval(gmp_init($num, $base_a), $base_b);
}

echo "12345678987654321 in hex is: " .
  gmp_convert('12345678987654321', 10, 16) . "\n";
?>
```

How It Works

This example takes a large integer value and converts it into a hexadecimal representation. The output will look like this:

```
12345678987654321 in hex is: 2bdc546291f4b1
```

Note that all the integer values are represented as strings.

■ **Note** Loading the GMP extension as a DLL will work only on Windows systems, and using the `dl()` function will work only for CLI and Common Gateway Interface (CGI) versions of PHP. For the Unix system, the GMP extension will be built-in or must be loaded as `gmp.so`.

The large integer values are stored internally as resource types. The function `gmp_init()` takes two parameters, where the first is a string representation and the second is an optional base value if the integer is given in a base value other than 10. The function `gmp_strval()` can convert a GMP resource to a readable string value. The rest of the functions manipulate one or more large integer values.

3-7. A Static Math Class

The math functions in PHP are, for the most part, designed to be used directly as functions and procedures, but with the new object model introduced in PHP 5 it's possible to create a static `Math()` class that will act like math classes in other languages such as Java or JavaScript.

■ **Note** It's always faster to call the functions directly than it is to use classes to wrap around the functions. However, static classes can make function names easier to remember, as they can be defined closer to what is used in other languages.

The next example shows how you can create and use a simple static `Math()` class. Using the `static` keyword in front of class members and methods makes it possible to use these without instantiating the class.

The Code

```
<?php
// Example math.php
define('RAND_MAX', mt_getrandmax());

class Math {
  static $pi = M_PI;
  static $e = M_E;

  static function pi() {
    return M_PI;
  }
  static function intval($val) {
    return intval($val);
  }
```

```
  static function floor($val) {
    return floor($val);
  }
  static function ceil($val) {
    return ceil($val);
  }
  static function round($val, $decimals = 0) {
    return round($val, $decimals);
  }
  static function abs($val) {
    return abs($val);
  }
  static function floatval($val) {
    return floatval($val);
  }
  static function rand($min = 0, $max = RAND_MAX) {
    return mt_rand($min, $max);
  }
  static function min($var1, $var2) {
    return min($var1, $var2);
  }
  static function max($var1, $var2) {
    return max($var1, $var2);
  }
}

$a = 3.5;

echo "Math::\$pi = " . Math::$pi . "\n";
echo "Math::\$e = " . Math::$e . "\n";
echo "Math::intval($a) = " . Math::intval($a) . "\n";
echo "Math::floor($a) = " . Math::floor($a) . "\n";
echo "Math::ceil($a) = " . Math::ceil($a) . "\n";
echo "Math::round(Math::\$pi, 2) = " . Math::round(Math::$pi, 2) . "\n";
echo "Math::abs(-$a) = " . Math::abs(-$a) . "\n";
echo "Math::floatval($a) = " . Math::floatval($a) . "\n";
echo "Math::rand(5, 25) = " . Math::rand(5, 25) . "\n";
echo "Math::rand() = " . Math::rand() . "\n";
echo "Math::min(2, 28) = " . Math::min(3, 28) . "\n";
echo "Math::max(3, 28) = " . Math::max(3, 28) . "\n";
?>
```

How It Works

The output from this script is simple but shows how the class is used:

```
Math::$pi = 3.14159265359
Math::$e = 2.71828182846
Math::intval(3.5) = 3
Math::floor(3.5) = 3
Math::ceil(3.5) = 4
Math::round(Math::$pi, 2) = 3.14
Math::abs(-3.5) = 3.5
Math::floatval(3.5) = 3.5
Math::rand(5, 25) = 13
Math::rand() = 1651387578
Math::min(2, 28) = 3
Math::max(3, 28) = 28
```

The JavaScript `Math()` class does not implement the `intval()`, `floatval()`, and `rand()` functions, and the `round()` function does not take a second argument to specify the number of decimals. The following example shows the same code in JavaScript.

The Code

```
<html>
<!-- Example math.html -->
<body>
<script language=JavaScript>
  a = 3.5;
  document.write('Math.PI = ' + Math.PI + '<br>');
  document.write('Math.E = ' + Math.E + '<br>');
  document.write('floor(' + a + ') = ' + Math.floor(a) + '<br>');
  document.write('ceil(' + a + ') = ' + Math.ceil(a) + '<br>');
  document.write('round(Math.PI) = ' + Math.round(Math.PI) + '<br>');
  document.write('min(3, 28) = ' + Math.min(3, 28) + '<br>');
  document.write('max(3, 28) = ' + Math.max(3, 28) + '<br>');
</script>
</body>
</html>
```

How It Works

Figure 3-3 shows the output in a browser.

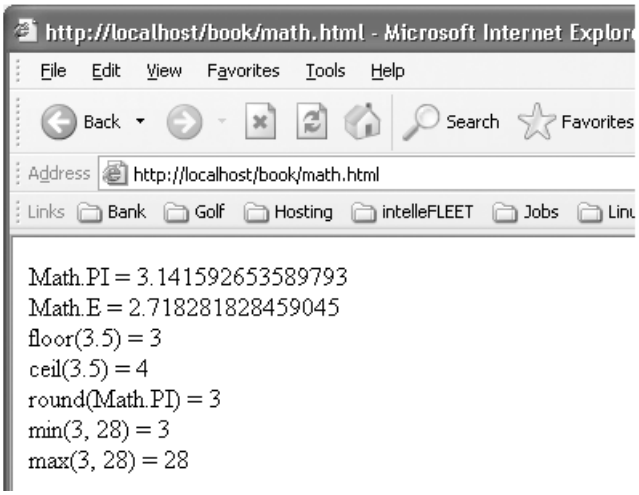

Figure 3-3. *Using the* `Math()` *class in JavaScript*

Summary

This chapter demonstrated how you can use many of the built-in math functions and operators in conjunction with the advantages of a loosely typed language such as PHP to calculate simple but advanced computations.

We first covered the basic data types and how PHP handles them when assigning and calculating values. Then we discussed the conversion of integers between different base values.

Next, we talked about random numbers and how to build functions to generate random values of floating-point or string data types.

The next two topics were logarithmic and trigonometric functions. These functions have a wide range of usages, but this chapter concentrated on how you can use them to generate charts and calculate the distance between two points on the earth.

Then, we discussed two extensions for handling math on numbers that do not fit into the simple numeric data types of PHP. Finally, we showed how you can create a static math class and use it like you would implement math classes in other languages.

Looking Ahead

In Chapter 4, Jon Stephens will demonstrate how to use arrays as complex data types in PHP. The chapter will show how you can manipulate arrays, how you can search arrays to find a specific value, and how you can sort and traverse arrays with different methods.