Contents of The Lectures

Chapter 1, Introduction to PHP

It talks about the history of PHP and gives a lightning-fast overview if what is possible with PHP programs.

Chapter 2, Language Basics

It concisely guides PHP program elements such as identifiers, data types, operators, and flow-control statements.

Chapter 3, Functions

Discusses user-defined functions, including scope, variable-length parameter lists, and variable and anonymous functions.

Chapter 4, Strings

It covers your PHP code’s functions when building, dissecting, searching, and modifying strings.

Chapter 5, Arrays

It covers your PHP code’s functions when building, dissecting, searching, and modifying strings.

Chapter 6, Objects

Covers PHP’s updated object-oriented features. In this chapter, you’ll learn about classes, objects, inheritance, and introspection.

Chapter 7, Dates and Times

Discusses date and time manipulations like time zones and data math.

Chapter 8, Web Techniques

It talks about techniques most PHP programmers eventually want to use, including processing web form data, maintaining state, and dealing with SSL.

Chapter 9, Databases

Discusses PHP’s modules and functions for working with databases, using MySQL database as examples. Also, SQLite and PDO database interfaces are covered. NoSQL concepts are also covered here.

Chapter 10, Graphics

Demonstrates how to create and modify image files in various formats within PHP.

Chapter 11, PDF

Explains how to create dynamic PDF files from a PHP application.

Chapter 12, XML

Introduces PHP’s extensions for generating and parsing XML data.

Chapter 13, JSON

It covers JavaScript Object Notation (JSON), a standardized data-interchange format that is extremely lightweight and human-readable.

Chapter 14, Security

Provides valuable advice and guidance for programmers creating secure scripts. You’ll learn programming best practices to help you avoid mistakes that can lead to disaster.

Chapter 15, Application Techniques

It discusses coding techniques like implementing code libraries, dealing with output in unique ways, and error handling.

Chapter 16, Web Services

Describes techniques for dealing with external communication via REST tools and cloud connections.

Chapter 17, Debugging PHP

Discusses techniques for debugging PHP code and for writing debuggable PHP code.

Chapter 18, PHP on Disparate Platforms

Discusses the tricks and traps of the Windows port of PHP. It also discusses some of the features unique to Windows, such as COM.

**CHAPTER 1 – Introduction to PHP**

PHP is a simple yet powerful language designed for creating HTML content. This chapter covers essential background on the PHP language. It describes the nature and history of PHP, which platforms it runs on, and how to configure it. This chapter ends by showing you PHP in action, with a quick walkthrough of several PHP programs that illustrate common tasks, such as processing form data, interacting with a database, and creating graphics.

**What Does PHP DO?**

PHP can be used in two primary ways:

Server-side scripting

PHP was originally designed to create dynamic web content and is still best suited for that task. To generate HTML, you need a PHP parser and a web server to send the coded document files. PHP has also become popular for generating dynamic content via database connections, XML documents, graphics, PDF files, and more.

Command-line scripting

PHP can run scripts from the command line like Perl, awk, or the Unix shell. You might use the command-line scripts for system administration tasks, such as backup and log parsing; even some SRON job-type scripts can be done this way (As nonvisual PHP tasks).

In this document, however, we concentrate on the first item: using PHP to develop dynamic web content.

PHP runs on all major operating systems, from Unix variants (including Linux, FreeBSD, Ubuntu, Debian, and Solaris) to Windows and macOS. It can be used with all leading web servers, including Apache, Nginx, and OpenBSD servers, to name a few, even cloud environments like Azure and Amazon.

The language itself is extremely flexible. For example, you aren’t limited to outputting just HTML or other text files- any document format can be generated. PHP has built-in support for generating PDF files, GIF, JPEGs, and PNG images.

One of PHP’s most significant features is its wide-ranging support for databases. PGP supports all major databases (including MYSQL, PostgreSQL, Oracle, Sybase, MS-SQL, DB2, and ODBC-compliant databases), and even many obscure ones. Even the more recent NoSQL-style databases like CouchDB and MongoDB are also supported. Creating web pages with dynamic content from a database with PHP is remarkably simple.

Finally, PHP provides a library of PHP code to perform common tasks, such as database abstraction, error handling, and so on, with the PHP Extension and Application Repository (PEAR). PEAR is a framework and distribution system for reusable PHP components.

**A Brief History of PHP**

Rasmus Lerdorf first conceived of PHP in 1994, but the PHP that people use today differs from the initial version. To understand how PHP got where it is now, it is useful to know the historical evolution of the language. Here’s that story, with ample comments and emails from Rasmus himself.

**The Evolution of PHP**

Here is the PHP 1.0 announcement that was posted to the Usenet newsgroup (*comp.infosystems.www.authoring.cgi*) in June 1995:

From: rasmus@io.org (Rasmus Lerdorf)

Subject: Announce: Personal Home Page Tools (PHP Tools)

Date: 1995/06/08

Message-ID: <3r7pgp$aa1@ionews.io.org>#1/1

organization: none

newsgroups: comp.infosystems.www.authoring.cgi

Announcing the Personal Home Page Tools (PHP Tools) version 1.0.

These tools are a set of small tight cgi binaries written in C.

They perform a number of functions including:

. Logging accesses to your pages in your own private log files

. Real-time viewing of log information

. Providing a nice interface to this log information

. Displaying last access information right on your pages

. Full daily and total access counters . Banning access to users based on their domain

. Password protecting pages based on users' domains

. Tracking accesses \*\* based on users' e-mail addresses \*\*

. Tracking referring URL's - HTTP\_REFERER support

. Performing server-side includes without needing server support for it

. Ability to not log accesses from certain domains (ie. your own)

. Easily create and display forms

. Ability to use form information in following documents

Here is what you don't need to use these tools:

. You do not need root access - install in your ~/public\_html dir

. You do not need server-side includes enabled in your server

. You do not need access to Perl or Tcl or any other script interpreter

. You do not need access to the httpd log files

The only requirement for these tools to work is that you have

the ability to execute your own cgi programs. Ask your system

administrator if you are not sure what this means.

The tools also allow you to implement a guestbook or any other

form that needs to write information and display it to users

later in about 2 minutes.

The tools are in the public domain distributed under the GNU

Public License. Yes, that means they are free!

For a complete demonstration of these tools, point your browser

at: http://www.io.org/~rasmus

--

Rasmus Lerdorf

rasmus@io.org

http://www.io.org/~rasmus

Note that this message’s URL and email address are long gone. The language of this announcement reflects people’s concerns at the time, such as password-protecting pages, easily creating forms, and accessing forms data on subsequent pages. The announcement illustrates PHP’s initial positioning as a framework for several use cases.

The announcement talks only about the tools that came with PHP, be behind the scenes, the goal was to create a framework to make it easy to extend PHP and add more tools. The business logic for these add-ons was written in C, a simple parser picked tags out of the HTML and called the various C functions. It was never really part of the plan to create a scripting language.

So what happened?

Rasmus started working on a rather large project for the University of Toronto that needed a tool to gather data from various places and present a nice web-based administration interface. OF course, he used PHP for that task, but for performance reasons, the various small tools of PHP 1.0 had to be brought together better and integrated into the web server.

Initially, some hacks to the NCSA web server were made to patch it to support the core PHP functionality. The problem with this approach was that you had to replace your web server software with this special, hacked-up version as a user. Fortunately, Apache was also gaining momentum around this time, and the Apache API made adding functionality like PHP to the server easier.

Over the next year or so, a lot was done, and the focus changed quite a bit. Here’s the PHP 2.0 (PHP/FI) announcement that was sent out in April 1996:

From: rasmus@madhaus.utcs.utoronto.ca (Rasmus Lerdorf)

Subject: ANNOUNCE: PHP/FI Server-side HTML-Embedded Scripting Language

Date: 1996/04/16

Newsgroups: comp.infosystems.www.authoring.cgi

PHP/FI is a server-side HTML embedded scripting language. It has built-in

access logging and access restriction features and also support for

embedded SQL queries to mSQL and/or Postgres95 backend databases.

It is most likely the fastest and simplest tool available for creating

database-enabled web sites.

It will work with any UNIX-based web server on every UNIX flavour out

there. The package is completely free of charge for all uses including

commercial.

Feature List:

. Access Logging

Log every hit to your pages in either a dbm or an mSQL database.

Having hit information in a database format makes later analysis easier.

. Access Restriction

Password protect your pages, or restrict access based on the refering URL

plus many other options.

. mSQL Support

Embed mSQL queries right in your HTML source files

. Postgres95 Support

Embed Postgres95 queries right in your HTML source files

. DBM Support

DB, DBM, NDBM and GDBM are all supported

. RFC-1867 File Upload Support

Create file upload forms

. Variables, Arrays, Associative Arrays

. User-Defined Functions with static variables + recursion

. Conditionals and While loops

Writing conditional dynamic web pages could not be easier than with

the PHP/FI conditionals and looping support

. Extended Regular Expressions

Powerful string manipulation support through full regexp support

. Raw HTTP Header Control

Lets you send customized HTTP headers to the browser for advanced

features such as cookies.

. Dynamic GIF Image Creation

Thomas Boutell's GD library is supported through an easy-to-use set of

tags.

It can be downloaded from the File Archive at: <URL:http://www.vex.net/php>

--

Rasmus Lerdorf

[rasmus@vex.net](mailto:rasmus@vex.net)

This was the first time the term scripting language was used. PHP 1.0’s simplistic tag-replacement code was replaced with a parser to handle a more sophisticated embedded tag language. By today’s standards, the tag language wasn’t particularly sophisticated, but compared to PHP 1.0, it certainly was.

The main reason for this change was that only some people who used PHP 1.0 were actually interested in using the C-based framework for creating add-ons. Most users were much more interested in being able to embed logic directly in their web pages for creating conditional HTML, custom tags, and other such features. PHP 1.0 users constantly requested the ability to add the hit-tracking footer or send different HTML blocks conditionally. This led to the creation of an if tag. Once you have if, you need else as well, and from there, it’s a slippery slope to the point where you write an entire scripting language, whether you want to or not.

By mid-1997, PHP version 2.0 had grown quite a bit and had attracted a lot of users, but there were still come stability problems with the underlying parsing engine. The project was also sill mostly a one-man effort, with a few contributions here and there. At this point, Zeev Suraski and Andi Gutmans in Tel Aviv, Israel, volunteered to rewrite the underlying parsing engine, and we agreed to make their rewrite the base for PHP version 3.0. Other people also volunteered to work on other parts of PHP, and the project changed from a one-person effort with a few contributors to a true open-source project with many developers worldwide.

Here is the PHP 3.0 announcement from June 1998:

June 6, 1998 -- The PHP Development Team announced the release of PHP 3.0,

the latest release of the server-side scripting solution already in use on

over 70,000 World Wide Web sites.

This all-new version of the popular scripting language includes support

for all major operating systems (Windows 95/NT, most versions of Unix,

and Macintosh) and web servers (including Apache, Netscape servers,

WebSite Pro, and Microsoft Internet Information Server).

PHP 3.0 also supports a wide range of databases, including Oracle,

Sybase, Solid, MySQ, mSQL, and PostgreSQL, as well as ODBC data sources.

New features include persistent database connections, support for the

SNMP and IMAP protocols, and a revamped C API for extending the language

with new features.

"PHP is a very programmer-friendly scripting language suitable for

people with little or no programming experience as well as the

seasoned web developer who needs to get things done quickly. The

best thing about PHP is that you get results quickly," said

Rasmus Lerdorf, one of the developers of the language.

"Version 3 provides a much more powerful, reliable, and efficient

implementation of the language, while maintaining the ease of use and

rapid development that were the key to PHP's success in the past,"

added Andi Gutmans, one of the implementors of the new language core.

"At Circle Net we have found PHP to be the most robust platform for

rapid web-based application development available today," said Troy

Cobb, Chief Technology Officer at Circle Net, Inc. "Our use of PHP

has cut our development time in half, and more than doubled our client

satisfaction. PHP has enabled us to provide database-driven dynamic

solutions which perform at phenomenal speeds."

PHP 3.0 is available for free download in source form and binaries for

several platforms at http://www.php.net/.

The PHP Development Team is an international group of programmers who

lead the open development of PHP and related projects.

For more information, the PHP Development Team can be contacted at

core@php.net.

After the release of PHP 3.0, usage really started to take off. Version 4.0 was prompted by a number of developers who were interested in making some fundamental changes to the architecture of PHP. These changes included abstracting the layer between the language and the web server, adding a thread-safety mechanism, and adding a more advance two-stage parse/execute tag-parsing system. This new parser, primary written by Zeev and Andi, was named the Zend engine. After a lot of work by a lot of developers, PHP 4.0 was released on May 22,2000.

As this document is transcribed, PHP version 7.3 has been released for some time. There have already been a few minor “dot” releases, and the stability of this current version is quite high. As you will see in this document, there have been some major advances made in this version of PHP, primarily in code processing on the server side. Many other minor changes, function additions, and feature enhancements have also been incorporated.

**The Widespread Use Of PHP**

Figure 1-1 shows the usage of PHP as compiled by W3 Techs as of July 2023. The most interesting piece of data here is that 77% of all the surveyed websites use it, with version 7 being the most widely used. If you look at the methodology used in the W3Techs surveys, you will see that they select the world's top 10 million sites (based on traffic; website popularity). As it evident, PHP has a very broad adoption indeed.

A screenshot of a web page

Description automatically generated

Figure 1-1 PHP usage as of July 2023

**Installing PHP**

As mentioned, PHP is available for many operating systems and platforms. Therefore, you are encouraged to consult the PHP documentation to find the environment that most closely fits the one you will be using and follow the appropriate setup instructions.

From time to time, you may also want to change the way PHP is configured. To do that , you will have to change the PHP configuration file and restart your web (Apache) server in order for those changes to take effect.

PHP’s configuration settings are usually maintained in a file called php.ini. The settings in this file control the behavior of PHP features, such as session handling and form processing. Later chapters refer to some of the php.ini options, but generally, the code in this Document does not require a customized configuration. See the PHP documentation for more information on configuring php.ini.

**A Walk Through of PHP**

PHP pages are generally HTML pages with PHP commands embedded in them. This is in contrast to many other dynamic web page solutions, which are scripts that generate HTML. The web server processes the PHP commands and sends their output (and any HTML from the file) to the browser. Example 1-1 shows a complete PHP page

<html>

    <head>

        <title> Look Out World</title>

    </head>

    <body>

        <?php echo "Hello, world!";?>

    </body>

</html>

Example 1-1. Hello\_world.php

Save the contents of Example 1-1 to a file, hello\_world.php, and point your browser to it. The results appear in Figure 1-2.

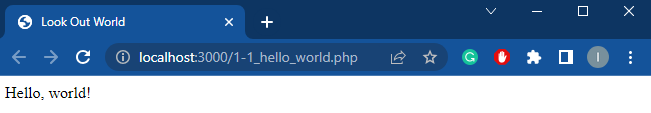


Figure 1-2. Output of hello\_world

The PHP echo command produces output (the string “Hello, world!” in this case) inserted into the HTML file. In this example the PHP code is placed between the <?php and ?> tags. There are other ways to tag your PHP code – see Chapter 2 for a full description.

**Configuration Page**

The PHP function phpinfo() creates an HTML page full of information on how PHP was installed and is currently configured. You can use it to see whether you have a particular extensions installed, or whether the php.ini file has been customized. Example 1-2 is a complete page that displayed the phpinfo() page.

<html>

<head>PHP info</head>

<body>

    <?php phpinfo();?>

</body>

</html>

Example 1-2. Using phpinfo()

A screenshot of a computer

Description automatically generated

Figure 1-3. Partial output of phpinfo()

**Forms**

Example 1-3 creates and processes a form. When the user submits the form, the information typed into the name field is sent back to this page via the $\_SERVER[‘PHP\_SELF’] form action. The PHP code tests for a name field and displays a greeting if it finds one.

<html>

    <head>

        <title>Personalized Greeting Form</title>

    </head>

    <body>

    <?php if(!empty($\_POST['name'])){

        echo "Greetings, {$\_POST['name']}, and welcome.";

    } ?>

    <form action="<?php echo $\_SERVER['PHP\_SELF']; ?>" method="post">

    Enter your name: <input type="text" name ="name" />

    <input type="submit"/>

    </form>

    </body>

</html>

The form and the message are shown in Figure 1-4.

A screenshot of a computer

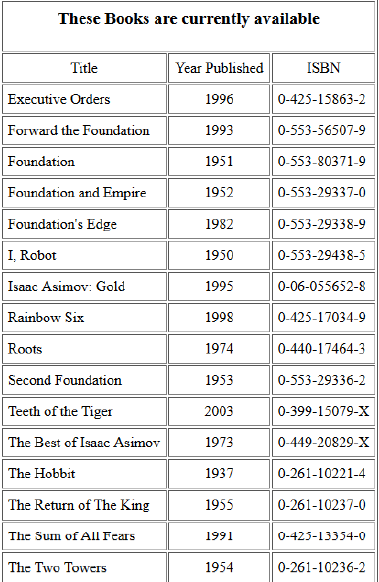
Description automatically generated

Figure 1-4. Form and greeting page

PHP programs access form values primarily through the $\_POST and $\_GET array variables. Chapter 8 discusses form and form processing in more detail.

**Databases**

PHP supports all the popular database systems, including MySQL, PostgreSQL, Oracle, Sybase, SQLite, and ODBC-compliant databases. Figure 1-5 shows part of a MySQL database query run through a PHP script, displaying the results of a book search on a book review site. It lists the book title, the year the book was published, and the book’s ISBN.

Figure 1-5. A MySQL book list query run through a PHP script

The code in Example 1-4 connects to the data base, uses a query to retrieve all available books with the WHERE clause), and produces a table as output for all returned results through a while loop. The SQL code for this sample database is in the provided file library. SQL. You can drop this code into MySQL after you create the library database and have the sample database at your disposal for testing out the following code sample as well as the related samples in Chapter 9.

<?php

$db = new mysqli("localhost", "peterwin", "password", "library");

// fictitious database connect function

// Check connection

if ($db->connect\_errno) {

    die("Connect Error (" . $db->connect\_errno . ") " . $db->connect\_error);

}

$sql = "SELECT \* FROM books WHERE available = 1 ORDER BY title";

$result = $db->query($sql);

?>

<!DOCTYPE html>

<html>

<head>

    <style>

        table {

            margin-left: auto;

            margin-right: auto;

            border-collapse: collapse;

        }

        th, td {

            padding: 6px;

            border: 1px solid black;

        }

        th {

            text-align: center;

        }

    </style>

</head>

<body>

<table>

    <tr>

        <td colspan="3">

            <h3 align="center">These Books are currently available</h3>

        </td>

    </tr>

    <tr>

        <th>Title</th>

        <th>Year Published</th>

        <th>ISBN</th>

    </tr>

    <?php while ($row = $result->fetch\_assoc()) { ?>

        <tr>

            <td><?php echo htmlspecialchars($row['title']); ?></td>

            <td align="center"><?php echo htmlspecialchars($row['pup\_year']); ?></td>

            <td><?php echo htmlspecialchars($row['ISBN']); ?></td>

        </tr>

    <?php } ?>

</table>

</body>

</html>

Database-provided dynamic content that drives the news, blogs, and e-commerce sites at the heart of the web. More details on accessing databases from PHP are given in Chapter 9. (Instructors note: if you receive the following error -Uncaught Error: Call to undefined function imagecreatefrompng()

Try the followingThe error you're encountering is caused by the GD library not being enabled in your PHP configuration. The GD library is required for image processing functions like imagecreatefrompng() and imagettftext(). To resolve this issue, you need to enable the GD library.

Here are the steps to enable the GD library:

Locate the PHP configuration file (php.ini). The location of this file can vary depending on your operating system and PHP installation.

Open the php.ini file in a text editor. Search for the following line: ;extension=gd

Remove the semicolon at the beginning of the line to uncomment it: extension=gd Save the php.ini file. Restart your web server to apply the changes.)

**Graphics**

With PHP, you can easily create and manipulate images using the GD extension. Example 1-5 provides a text entry field that lets the user specify the text for a button. It takes an empty button image file, and centers the text passed as the GET parameter ‘message’ on it. The result is sent back to the browser as a PNG image.

The form generated by example 1-5 is shown in Figure 1-6. The button created is shown in Figure 1-7.

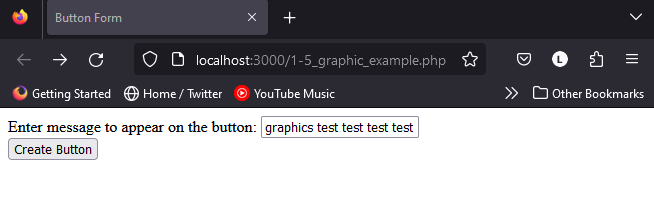


Figure 1-6. Button creation form

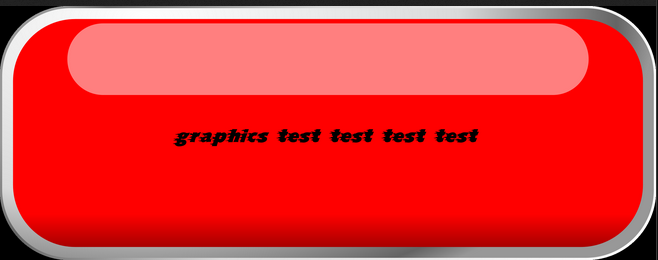


Figure 1-7. Button created

You can use GD to resize images dynamically, produce graphs, and much more. PHP also has several extensions to generate documents in Adobe’s popular PDF format. Chapter 10 covers dynamic image generation in depth, while Chapter 11 provides instructions on creating Adobe PDf files.

**What’s Next**

Now that you’ve tasted what is possible with PHP, you are ready to learn how to program in the language. We start with its basic structure, focusing on user-defined functions, string manipulation, and object-oriented programming. Then we move to specific application areas, such as the web, databases, graphics, XML, and security. We finish with quick references to the built-in functions and extensions. Master these chapters, and you will have mastered PHP!

**CHAPTER 2 – Language Basics**

This chapter provides a whirlwind tour of the core PHP language, covering such basic topics as data types, variables, operators, and flow-control statements. PHP is influenced by other programming languages such as Perl and C, so if you’ve had experience with those languages, PHP sould be easy to pick up. If PHP is one of your first programming languages, don’t panic. We start with the basic units of a PHP program and build up your knowledge from there.

**Lexical Structure**

The lexical structure of a programming language is the set of basic rules that governs how you write programs in that language. It is the language's lowest-level syntax and specifies what variables look like, what characters are used for comments, and how program statements are separated from each other.

**Case Sensitivity**

The names of user-defined classes and functions, as well as build-in constructs and keywords (such as echo, while, class, etc), are case-insensitive. Thus, these three lines are equivalent:

<html>

    <head>

        <title> Look Out World</title>

    </head>

    <body>

        <?php echo "Hello, world!";

            ECHO "Hello, world!";

            EcHo "Hello, world!";?>

    </body>

</html>

Variables, on the other hand, are case-sensitive. That is $name, $NAME, and $NaME are three different variables.

**Statements and Semicolons**

A statement is a collection of PHP code that does something. It can be as simple as a variable assignment or as complicated as a loop with multiple exit points. Here is a small sample of PHP statements, including function calls, some variable data assignments, and an if statement

<body>

        <?php echo "Hello, world";

        myFunction(42,"Web-Design-Tools");

        $a = 1;

        $name = "Elephant";

        $b=$a/25.0;

        if($a==$b)

        {

            echo "Rhyme? And Reason?";

        }?>

    </body>

PHP uses semicolons to separate simple statements. A compound statement that uses curly braces to mark a block of code, such as a conditional test or loop, does not need a semicolon after a closing brace. Unlike in other languages, in PHP, the semicolon before the closing brace is not optional:

<?php if($needed)

        {

            echo "We must have it!";// semicolon required here

        }// no semicolon required here after the brace

        ?>

The semicolon, however, is optional before a closing PHP tag:

<?php if($a==$b)

        {

            echo "Rhyme? And Reason?";

        }

        echo"Hello, world"// no semicolon required before closing tag

        ?>

Its good programming practice to include optional semicolons, as they make it easier to add code later.

**Whitespace and Line Breaks**

In general, whitespce doesn’t matter in a PHP program. You can spread a statement acress any number of lines, or lump a bunch of statements together on a single line. For example, this statement;

<?php raisePrices($inventory, $inflation, $costOfLiving, $greed);?>

Could just as well be written with more whitespace:

<?php raisePrices(

            $inventory,

            $inflation,

            $costOfLiving,

            $greed);?>

Or with less whitespace:

<?php raisePrices($inventory,$inflation,$costOfLiving,$greed);?>

You can use this flexible formatting to make your code more readable (by lining up assignments, indenting, etc.) Some lazy programmers use this freeform formatting and create unreadable code. For example, you could say theatrically. Give each character in your code a space making it impossible to read.

**Comments**

Comments give information to people who read your code, but PHP ignores them at execution time. Even if you think you’re the only person who will ever read your code, it’s a good idea to include comments in your code- in retrospect, code you wrote months ago could easily look like a stranger wrote it.

A goof practice is to make your comments sparse enough not to get in the way of the code itself but plentiful enough that you can use the comments to tell what’s happening. Don’t comment on obvious things, lest you bury the comments that describe tricky things. For example this is worthless:

<?php $x = 17;// store 17 into the variable $x?>

Whereas the comments on this complex regular expression will help whoever maintains your code:

<?php $text =preg\_replace('/&#(0-9)+;/', "chr('\\1')");// convert &#nnn; entities into characters?>

*The purpose of this code is to convert HTML numeric character references (e.g.,* ***&#65;*** *representing the character 'A') into their corresponding characters. For example, if the input text contains* ***&#65;****, the regular expression will match it, and* ***chr(65)*** *will convert it to the character 'A'.*

PHP provides several ways to include comments within your code, all of which are borrowed from existing languages such as C, C++, and the Unix shell. In general, use C-style comments to comment out code, and C++ style comments to comment on code.

**Shell-style comments**

When PHP encounters a hash mark character (#) within the code, everything from the has mark to the end of the line or end of the section of PHP code (whichever comes first) is considered a comment. This method commenting is found in Unix shell scripting languages and is useful for annotating single lines of code or making short notes.

Because the hash mark is visible on the page, shell-style comments are sometimes used to mark off blacks of code:

<?php #######################

       ## Cookie functions

       #######################?>

Sometimes they’re used before a line of code to identify what that code does, in which case they’re usually indented to the same level as the code for which the comment is intended:

<?php if($doubleCheck){

                # create an HTML form requesting that the user confirm the action

                echo confirmationForm();

                }?>

Short comments on a single line of code are often put on the same line as the code:

<?php $value = $p \* exp($r \* $t); # calculate compounded interest?>

When you’re tightly mixing HTML and PHP code, it can be useful to have the closing PHP tag terminate the comment:

 <?php $d=4; #Set $d to 4.?> then another <?php echo $d; ?>

**C++ comments**

When PHP encounters two slashes (//) within the code,, everything from the slashes to the end of the line or the end of the code section, whichever comes first., is considered a comment. This method of commenting is derived from C++. The result is the same as the shell comment style.

Here are the shell-style comment example, rewritten to use C++ comments:

<?php ////////////////////////

              // Cookie functions

              ////////////////////////

              if($doubleCheck)

              {

                // create an HTML form requesting that the user confirm //the action

                echo confirmationForm();

              }

              $value = $p \* exp($r \* $t); // calculate compounded interest

              ?>

**C comments**

While shell-style and C++-style comments are useful for annotating code or making short notes, longer comments require a different style. Therefore, PHP supports block comments whose syntax comes for the C programming language. When PHP encounters a slash followed by an asterisk (/\*), everything after that until it encounters an asterisk followed by a slash (\*/), is considered a comment. Unlike those shown earlier, this kind of comment can span multiple lines.

Here’s an example of a C-style multiline comment:

<?php /\* In this section, we take a bunch of variables and

                assign numbers to them. There is no real reason to

                do this, we're just having fun.

              \*/

            $a = 1;

            $b = 2;

            $c = 3;

            $d = 4;

              ?>

Because C-style comments have specific start and end markers, you can tightly integrate them with code. This tends to make your code harder to read and is discouraged:

<?php /\* These comments can be mixed with code too,

                see? \*/ $e = 5; /\* This works just fine. \*/

              ?>

C-style comments, unlike the other types, can continue past the end PHP tag markers. For example:

<?php

$l = 12;

$m = 13;

/\* A comment begins here

?>

<p>Some stuff you want to be HTML.</p>

<?= $n = 14; ?>

\*/

echo("l=$l m=$m n=$n\n");

?><p>Now <b>this</b> is regular HTML...</p>

l=12 m=13 n=

<p>Now <b>this</b> is regular HTML...</p>

              ?>

You can indent comments as you like:

 /\* There are no

special indenting or spacing

rules that have to be followed, either.

\*/

C-style comments can be useful for disabling sections of code. In the following example, we disabled the second and third statements and the inline comments by including them in a block comment. To enable the code, all we must do is remove the comment markers:

 <?php

        $f=6;

        /\*\*

         \* $g = 6; # This is a different style of comment

         \* $g = 8;

         \*/

        ?>

However, you must be careful not to attempt not to nest block comments:

<?php

        $i=9;

        /\*

        $j=10; /\* This is a comment\*/

        $k=11;

        Here is some comment text.

        \*/

        ?>

In this case, PHP tries (and fails) to execute the (non) statement Here is some comment text and returns an error.

**Literals**

A literal is a data value that appears directly in a program. The following are all literals in PHP:

<?php

        2001

        0xFE

        2.4142

        "Hello World"

        'Hi'

        true

        null

        ?>

**Identifiers**

An identifier is simply a name. In PHP, identifiers name variables, functions, constants, and classes. The first character of an identifier must be an ASCII letter (uppercase or lowercase), the underscore character (\_), or any other characters between ASCII 0x7F and ASCII 0xFF. After the initial character, these characters and the digits 0-9 are valid.

**Variable names**

Variable names always begin with a dollar sign ($) and are case-sensitive. Here are some valid variables names:

<?php

        $bill

        $head\_count

        $MaximumForce

        $I\_HEART\_PHP

        $\_underscore

        $\_int

        ?>

Here are some illegal variable names:

<?php

       $not valid

       $|

       $3wa

        ?>

These variables are all different due to case sensitivity:

<?php

      $hot\_stuff

      $Hot\_stuff

      $hot\_Stuff

      $HOT\_STUFF

        ?>

**Function names**

Function names are not case-sensitive (functions are discussed in more detail in Chapter 3). Here are some valid function names:

<?php

      tally

      list\_all\_users

      deleteTclFiles

      LOWERCASE\_IS\_FOR\_WIMPS

      \_hide

        ?>

These function names all refer to the same function:

<?php

      howdy

      HoWdY

      HOWDY

      HOWdy

      howdy

        ?>

**Class names**

Class names follow the standard rules for PHP identifiers and are also not casesensitive. Here are some valid class names:

<?php

      Person

      account

        ?>

Note: The class name stdClass is a reserved class name.

**Constants**

A constant is an identifier for a value that will not be changed; scaler values (Boolean, integer, double, and string) and arrays can be constants. Once set, the value of a constant cannot change. Constants are referred to by their identifiers and are set using the define() function:

<?php

      define('PUBLISHER', 'NOT ME I STOLE THIS');

      echo PUBLISHER;

        ?>

**Keywords**

A keyword (or reserved word) is a word set aside bt the language for its core functionality-you cannot give a function, class, or constant the same names a keyword. Table 2-1 lists the keywords in PHP which are case-insensitive.

A screen shot of a computer program

Description automatically generated

In addition, you cannot use an identifier that is the same as a built-in PHP function. For a complete list of these, see the Appendix.

**Data Types**

PHP provides eight types of values or data types. Four are scaler (single-value) types:

Integers, floating-point numbers, strings, and Booleans. Two are compound (collection) types: arrays and objects. The remaining two are special types: resource and NULL. Numbers, Booleans, resources, and NULL, are discussed in full here, while strings, arrays, and objects are big enough topics to get their chapters (Chapters 4.5 and 6, respectively).

**Integers**

Integers are whole numbers, such as 1,12,and 256. The range of acceptable values varies according to the details of your platform but typically extends from -2,127,483,648 to +-2,127,483,647—specifically, the range of the long data type of your C compiler. Unfortunately, the C standard doesn’t specify what long type range should have, l so on some systems; you might see a different integer range. Integer literals can be written in decimal, octal, binary, or hexadecimal. A sequence of digits represents decimal values without leading zeros. The sequence may begin with a plus (+) or a minus(-). If there is no sign, a positive is assumed. Examples of decimal integers include the following:

<?php

      1998

      -641

      +33

        ?>

Octal numbers consist of a leading 0 and a sequence of digits from 0 to 7. Like decimal numbers, octal numbers can be prefixed with a plus or minus> Here are some examples of octal values and their equivalent decimal values:

<?php

     0755 // decimal 493

     +010 // decimal 8

        ?>

Hexadecimal values begine with 0x, followed by a sequence of digits (0-9) or letters (A-F). The letters can be upper- or lowercase but are usually written in capitals. As with decimal and octal value, you can include a sign in hexadecimal numbers:

  0xFF // decimal 255

    0x10 // decimal 16

    –0xDAD1 // decimal −56017

Binary numbers begin with 0b, followed by a sequence of digits(0 and 1). AS with other values, you can include a sing in binary numbers:

    0b01100000 // decimal 96

    0b00000010 // decimal 2

    -0b10 // decimal -2

If you try to store a variable that is too large to be stored as an integer or is not a whole number, it will automatically be turned into a floating-point number.

Use the is\_int() function (or its is\_integer() alias) to test whether a vvalue is an integer

if(is\_int($x))

        {

            //$s is an integer

        }

**Floating-Point Numbers**

Floating-point number (often referred to as “real” numbers) represent numeric values with decimal digits. Like integer, their limits depend on your machine’s details. PHP floating-point numbers are equivalent to the range of the double data type of your C compiler. Usually, this allows numbers between 1.7E-308 and 1.7E+308 with 15 digits of accuracy. You can use the BC or GMP extensions if you need more accuracy or a wider range of integer values.

Php recognizes floating-point numbers written in two different formats. There’s the one we all use every day:

3.14

       0.17

       -7.1

But PHP also recognizes numbers in scientific. notation:

0.314E1 // 0.314\*10^1, or 3.14

       17.0E-3 // 17.0\*10^(-3), or 0.017

**(Instructors NOTE: Make a demonstration of this concept)**

Floating-point values are only approximate representations of numbers For example, on many systems, 3.5 is represented as 3.4999999999. This means you must take care to avoid writing that assumes floating-point numbers are represented completely accurately, such as directly comparing two floating-point values using ==. The normal approach is to compare to several decimal places:

if(inval($a \* 1000) == intval($b \* 1000))

        {

            // numbers equal to three decimal places

        }

Use the is\_float() function (or its is\_real() alias) to test whether a value is a floating-point number:

if(is\_float($x))

        {

            // $x is a floating-point number

        }

**Strings**

Because strings are so common in web applications, PHP includes core-level support for creating and manipulating strings. A string is a sequence of characters of arbitrary length. String literals are delimited by either single or double quotes:

'big dog'

        "fat hog"

Variables are expanded (interpolated) within double quotes, while within single quotes they are not:

$name = "Guido";

        echo"Hi, $name <br/>";

        echo'Hi, $name';

        Hi, Guido

        Hi, $name;

Double quotes also support a variety of string escapes, as Listed in Table 2-2.

A table with text on it

Description automatically generated

A single-quoted string recognizes \\ to get a literal backslash and \’ to ger a literal single quote:

  $dosPath = 'C:\\WINDOWS\\SYSTEM';

        $publisher = 'Tim O\'Reilly';

        echo "$dosPath $publisher";

        // OUTPUT C:\WINDOWS\SYSTEM Tim O'Reilly

To test whether two strings are equal, use the == (double equals) comparison operator:

if($a==$b)

        {

            echo"a and b are equal";

        }

Use the is\_string() function to test whether a value is a string:

if(is\_string($x))

        {

            //$x is a string

        }

PHP provides operators and functions to compare, disassemble, assemble, search, replace, and trim strings, as well as a host of specialized string sunctgions for working with HTTP, HTML, and SQL encodings. Because there are so many string-manipulation functions, we’re devoted a whole chapter (Chapter 4) to covering all the details.

**Booleans**

A Boolean value represents a truth value-it says whether something is true or not. Like most programming languages, php defines some value as true and other as false. Truthfulness and falseness determine the outcome of conditional code such as:

if ($alive) { ... }

In PHP, the following values all evaluate to false:

* The keyword false
* The integer 0
* The floating-point value 0.0
* The empty string (“ ”) and the string “0”
* An array with zero elements
* The NULL value

A value that is not false is true, including all resources values (which are described later in the section “Recources”).

PHP provides true and false keywords for clarity:

$x = 5;     //$x has a true value

        $x = true;  // clearer way to write it

        $y="";      // $y has a false value

        $y = false; // clear way to write it

Use the is\_boo() function to test whether a value is a Boolean:

if (is\_bool($x)) {

            // $x is a Boolean

            }

**Arrays**

An array holds a group of values, which you can identify by position (a number, with zero being the first position) or some identifying name (a string), called an associative index:

$person[0]="Edison";

       $person[1]="Wankel";

       $person[2]="Crapper";

       $creator['Light bulb']="Edison";

       $creator['Rotary Engine'] = "Wankel";

       $creator['Toilet'] = "Crapper";

The array() construct creates an array. Here are two example:

$person = array("Edison","Wankel","Crapper");

      $creator = array('Light bulb' => "Edison",

                       'Rotary Engine'=> "Wankel",

                       'Toilet'=> "Crapper");

There are several ways to loop through arrays, but most common is a foreach loop:

foreach($person as $name){

            echo"Hello,{$name}<br/>";

        }

        foreach($creator as $invention => $inventor){

            echo"{$inventor} invented the {$invention} <br/>";

        }

/\*OUTPUT: Hello, Edison

Hello, Wankel

Hello, Crapper

Edison created the Light bulb

Wankel created the Rotary Engine

Crapper created the Toilet

\*/

You can sort the elements of an array with the various sort functions:

sort($person);

// $person is now array("Crapper", "Edison", "Wankel")

asort($creator);

// $creator is now array ('Toilet' => "Crapper",

// 'Light bulb' => "Edison",

// 'rotary Engine' /> "Wankel");

Use the is\_array() function to test whether a value is an array:

if(is\_array($x))

{

    //$x is an array

}

There are functions for returning the number of items in the array fetching every value in the array, and much more. Arrays are covered in depth in Chapter 5.

**Objects**

PHP also supports object-oriented programming (OOP). OOP promotes clean, modular design; simplifies debugging and maintenance and assists with code reuse. Classes are the building blocks of object-oriented design. A class is a definition of a structure that contains properties (variables) and methods (functions). Classes are defined with the class keyword:

class Person{

    public $name='';

    function name($newname = NULL)

    {

        if(!is\_null($newname)){

            $this->name = $newname;

        }

        return $this->name;

    }

}

Once a class is defined, any number of objects can be made from it with the new keyword, and the object’s properties and methods can be accessed with the -> construct:

$ed = new Person;

$ed->name('Edison');

echo "Hello, {$ed->name}<br/>";

$tc = new Person;

$tc->name('Crapper');

echo "Look out below{$tc->name}";

/\*

OUTPUT:

Hello, Edison

Look out below Crapper

\*/

Use the is\_object() function to test whether a value is an object:

Chapter 6 describes classes and objects in much more detail, including inheritance, encapsulation, and introspection.

**Resources**

Many modules provide several functions for dealing with the outside world. For example, every database extension has at least a function to connect to the database, a function to close the connection to the database, and a function to query the database. Because you can have multiple database connections open simultaneously, the connect function gives you something to identify that unique connection when you call the query and close functions: a resource (or a handle).

Every active resource has a unique identifier. Each identifier is a numerical index into an internal PHP lookup table containing information about all the active resources. PHP maintains information about each resource in this table, including the number of references to (or uses of) the resource throughout the code. When the last reference to a resource value goes away, the extension that created the resource is called to perform tasks such as freeing any memory or closing any connection for that resource:

$res = database\_connect();// fictitious database connect function

    database\_query($res);

    $res = "boo";

    // database connection automatically closed because $res is redefined

The benefit of this automatic cleanup is best seen within functions when the resource is assigned to a local variable. When the function ends, the variable’s value is reclaimed by PHP:

function search(){

        $res=database\_connect();

        database\_query($res);

    }

When there are no more references to the resource, it’s automatically shut down. That said, most extensions provide a specific shutdown or close function, and it’s considered good to call that function explicitly when needed rather than relying on variable scoping to trigger resource cleanup.

Use the is\_resource() function to test whether a value is a resource:

**Callbacks**

Callbacks are functions or object methods some functions use, such as call\_user\_func(). Callbacks can also be created by the create\_function() method and through closures (described in Chapter 3):

 $callback = function()

    {

        echo "callback achieved";

    };

    call\_user\_func($callback);

**NULL**

There;s only one value of the NULL data type. That value is available through the case-insensitive keyword NULL. The NULL value represents a variable that has no value. (similar to Perl’s undef or Python’s None):

$alpha="beta";

    $aleph = null; // variable's value is gone

    $alpha=Null; // same

    $aleph = NULL; // same

Use the is\_null() function to test whether a value is NULL- for instance, to see whether a variable has a value:

if(is\_null($x)){

        // $x is NULL

    }

**Variables**

Variables in PHP are identifiers prefixed with a dollar sign ($). For example:

  $name

    $Age

    $\_debugging

    $MAXIMUM\_IMPACT

A variable may hold a value of any type. There is no compiler-time or runtime type checking on variables. You can replace a variable’s value with another of a different type:

    $what = "Fred";

    $what = 35;

    $what = array("Fred",35,"Wilma");

There is no explicit syntax for declaring variables in PHP. The first time the value of a variable is set, the variable is created in memory. In other words, setting a value to a variable also functions as a declaration. For example, this is a valid complete PHP program:

$day = 60\*60\*24

    echo "There are {$day} seconds in a day,";

    //OutPut There are 86400 seconds in a day.

A variable whose value has not been set behaves like the NULL value:

if($uninitializedVariable === NULL){

        echo "Yes!";

    }

    //OUTPUT: Yes!

**Variable Variables**

You can reference the value of a variable whose name is stored in another variable by prefacing the variable reference with an additional dollar sign ($). For example:

  $foo = "bar";

    $$foo ="baz";

After the second statement executes, the variable $bar has the value “baz“.

**Variable References**

IN PHP, references are how you create variable aliases or pointers. To make $black an alias for the variable $white, use:

$black =& $white;

The old value of $black, if any, is lost. Instead, $black is now another name for the value that is stored in $white:

$bigLongVariableName = "PHP";

    $short =& $bigLongVariableName;

    $bigLongVariableName .= " rocks!";

    print "\$short is $short <br/>";

    print "Long is $bigLongVariableName";

    /\*

    OUTPUT:

    $short is PHP rocks!

    Long is PHP rocks!

    \*/

    $short = "Programming $short";

    print"\Sshort is $short <br/>";

     /\*

    OUTPUT:

    $short is Programming PHP rocks!

    Long is Programming PHP rocks!

    \*/

After the assignment, the two variables are alternate names for the same value. Unsetting an aliased variable does not affect other names for that variable’s value, however:

$while = "snow";

    $black =& $white;

    unset($white);

    print $black;

    /\*

     \* OUTPUT: snow

     \*/

Functions can return values by reference (for example, to avoid copying large strings or arrays, as discussed in Chapter 3):

$while = "snow";

    $black =& $white;

    unset($white);

    print $black;

    /\*

     \* OUTPUT: snow

     \*/

**Variable Scope**

The scope of a variable, which is controlled by the location of the variable’s declaration, determines those parts of the program that can access it. There are four types of variable scope in PHP: local, global, static, and function parameters.

**Local scope**

A variable declared in a function is local to that function. It is visible only to code in that function (excepting nested function definitions); it is not accessible outside the function. In addition, by default, variables defined outside a function (global variables) are not accessible inside the function. For example, here’s a function that updates a local variable instead of a global variable:

function updateCounter()

     {

        $counter++:

     }

     $counter = 10;

     updateCounter();

     echo$counter;

     /\*

        OUTPUT:

        10

     \*/

The $counter inside the function is local to that function because we haven’t said otherwise. The function increments its private $counter variable, which is destroyed when the subroutine ends. The global $counter remains set at 10.

Only functions can provide local scope. Unlike in other languages, in PHP you can’t create a variable whose scope is a loop, conditional branch, or other type of block.

**Global scope**

Variables declared outside a function are global. That is, they can be accessed from any part of the program. However, by default, they are not available inside finctions. To allow a function to access a global variable within the function. Here’s how we can rewrite the updataCounter()function to allow it to access the global $counter variable:

function updateCounter()

     {

        global $counter;

        $counter++

     }

     $count = 10;

     updateCounter()

     echo $counter;

     /\*\*

      \* OUTPUT: 11

      \*/

A more cumbersome way to update the global variable is to use PHP’s $GLOBALS array instead of accessing the variable directly:

function updateCounter()

      {

        $GLOBALS['counter']++;

      }

      $counter = 10;

      updateCounter();

      echo $counter;

      /\*

        OUTPUT 11

      \*/

**Static variables**

A static variable retains its value between calls to a function but is visible only within that function. You declare a variable static with the static keyword. For example:

function updateCounter()

        {

            static $counter = 0;

            $counter++;

            echo "Static counter is now {$counter}<br/>"

        }

        $counter = 10;

        updateCounter();

        updateCounter();

        echo "Global counter is {$counter}";

        /\*

            OUTPUT:

            Static counter is now 1

            Static counter is now 2

            Global counter is 10

        \*/

**Function parameters**

As we’ll discuss in more detail in Chapter 3, a function definition can have named parameter:

function great($name)

        {

        echo"Hello, {$name}";

        }

        great("Janet");

        /\*

        OUTPUT: Hello, Janet

        \*/

Function parameters are local, meaning they are available only inside their functions. In this case, $name is inaccessible from outside greet().

**Garbage Collection**

PHP uses reference counting and copy-on-write to manage memory. Copy-On-write ensures that memory isn’t wasted when you copy values between variables, and reference counting ensures that memory is returned to the operating system when it is no longer needed.

To understand memory management in PHP, you must first understand the idea of a symbol table. There are two parts to a variable –its name (e.g, $name), and its value (e.g, “Fred”). A symbol table is an array that maps variable names to the position of their values in memory.

When you copy a value from one variable to another, PHP doesn’t get more memory for a copy of the value. Instead, it updates the symbol table to indicate that “both of these variables are names for the same chunk of memory.” So the following code doesn’t actually create a new array:

$worker = array("Fred", 35, "Wilma");

        $other = $worker; // array isn't duplicated in memory

If you subsequently modify either copy, PHP allocates the required memory and makes the copy:

$worker[1] = 36; // array is copied in memory,. value changed

By delaying the allocation and copying, PHP saves time and memory in many situations. This is copy-on-write.

Each value pointed to by a symbol table has a reference count, representing the number of ways to get to that piece of memory. After the initial assignment of the array to $worker and $worker to $other, the array pointed to by the symbol table entries for $worker and $other has a reference count of 2(it is actually three if you are looking at the reference count from the C API, but for this explanation and from a suer-space perspective, it is easier to think of it as 2). In order words, that memory can be reached through $worker or $other. But after $worker[1] is changed, PHP creates a new array for $worker, and the reference count of each array is only 1.

When a variable goes out of scope at the end of a function, such as function parameters and local variables, the reference count of its value is decreased by one. When a variable is assigned a value in a different area of memory, the reference count of the old value is decreased by one. When the reference count of a value reaches 0, its memory is released. This is reference counting.

Reference counting is the preferred way to manage memory. Keep variables local to functions, pass in values that the functions need to work on, and let reference counting take care of the memory management. If you do insist on trying to get a little more information or control over freeing a variables value, use the isset() and unset() functions.

To see if a variable has been set to something—even the empty string—use isset():

$s1 = isset($name);//$s1 is false

        $name = "Fred";

        $s2 = isset($name);//$s2 is true

Use unset() to remove a variable’s value:

$name = "Fred";

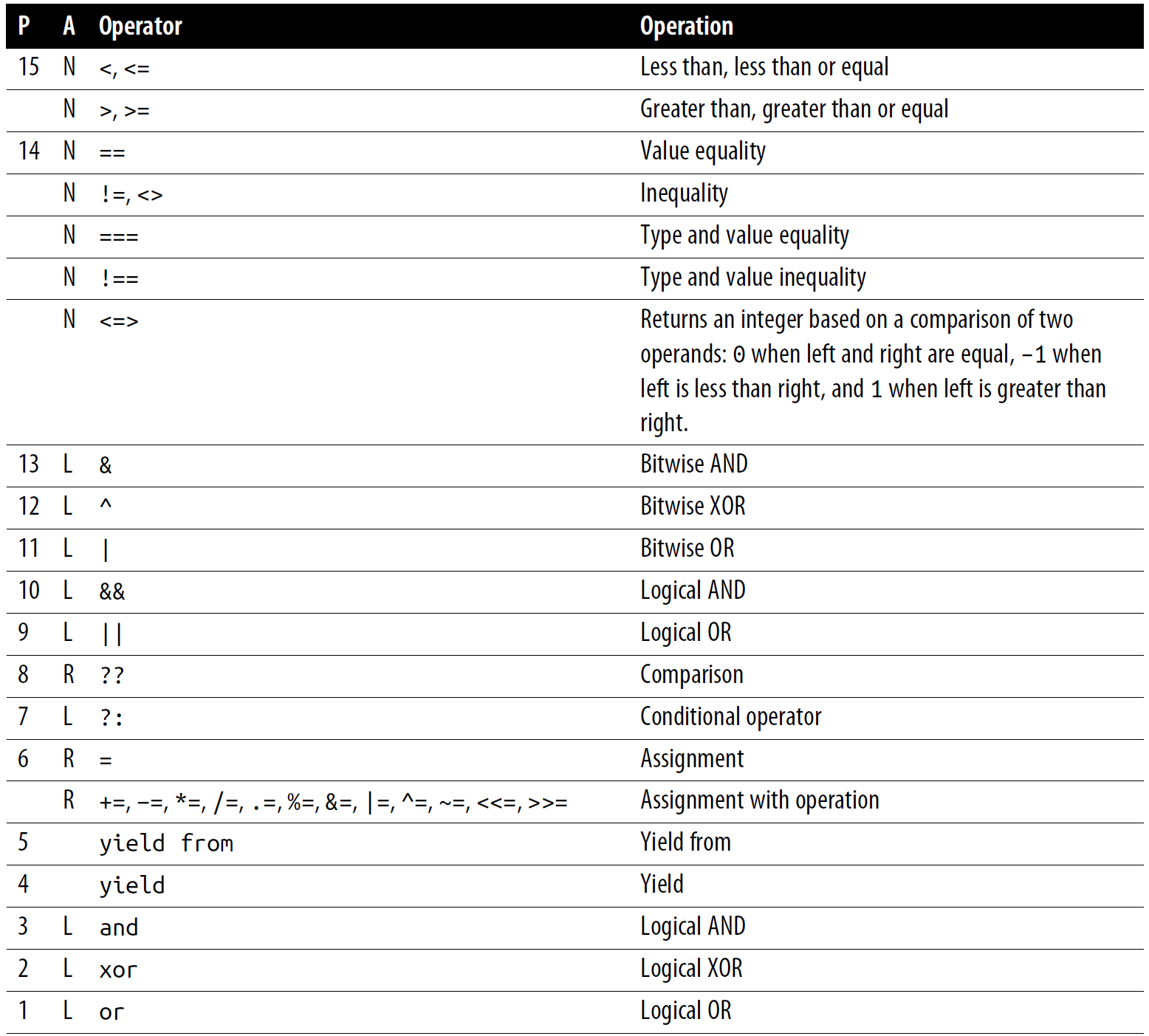
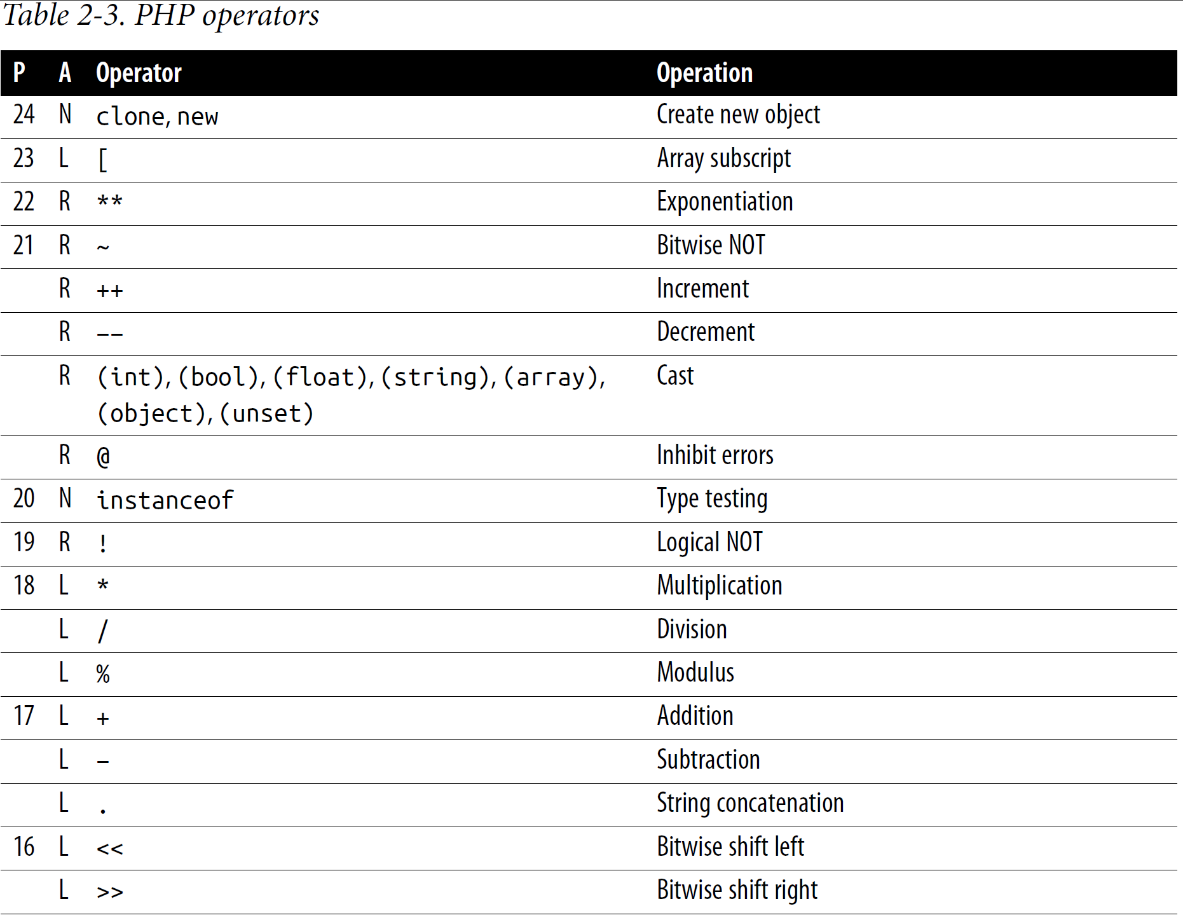
        unset($name);// $name is NULL

**Expressions and Operators**

An expression is a bit of PHP code that can be evaluated to produce a value. The simplest expressions are literal values and variables. A literal value evaluates to itself, while a variable evaluates to the value stored in the variable. More complex expressions can be formed using simple expressions and operators.

An operator takes some values (the operands) and dose something (e.g., adds them together). Operators are sometimes written as punctuation symbols—for instance, the + and – are familiar to us from math. Some operator modify their operands, while most do not.

Table 2-3 summarizes the operators in PHP, many of which were borrowed from C and Perl. The column labeled “P” gives the operator’s precedence; the operators are listed in precedence order, from highest to lowest. The column labeled “A” gives the operator’s associativity, which can be L (left-to-right), R (right-to-left), or N (nonassociative).



**Number of Operands**

Most operators in PHP are binary operators; they combine two operands (or expressions) into a single, more complex expressions. PHP also supports several unary operators, converting a single expression into a more complex one. Finally, PHP supports a few ternary operators that combine numerous expressions into a single expression.

**Operator Precedence**

The order in which operators in an expression are evaluated depends on their relative precedence. For example, you might write:

2+4\*3

Table 2-3 shows that the addition and multiplication operators have different precedence, with multiplication higher than addition. So the multiplication happens before the addition, giving 2 + 12, or 14, as the answer. Ig the precedence of addition and multiplication were reversed, 6 \* 3, or 18, would be the answer.

To force a particular order, you can group operands with the appropriate operator in parentheses. In our previous example, to get the value 18, you can use the expression:

(2 + 4) \* 3

It is possible to write all complex expressions (expressions containing more than a single operator) simply by putting the operands and operators in the appropriate order so that their relative precedence yields the answer you want. Most programmers, however, write the operators in the order that they feel makes the most sense to them, and add parentheses to ensure it makes sense to PHP as well. Getting precedence wrong leads to code like:

$x + 2 / $y >= 4 ? $z : $x << $z

This code is hard to read and is almost definitely not doing what the programmer expected it to do.

One way many programmers deal with the complex precedence rules in programming languages is to reduce precedence down to two rules:

* Multiplication and division have higher precedence than addition and subtraction.
* Use parentheses

**Operator Associativity**

Associativity defines the order in which operators with the same order of precedence are evaluated. For example, look at:

2 / 2 \* 2

The division and multiplication operators have the same precedence, but the result of the expression depends on which operation we do first:

2 / (2 \* 2) // 0.5

(2/2) \* 2   //2

The division and multiplication operators are left-associative; this means that in cases of ambiguity, the operators are evaluated from left to right. In this example, the correct result is 2.

**Implicit Casting**

For instance, many operators have expectations of their operands; binary math operators typically require both operands to be of the same type. PHP’s variables can store integers, floating-point numbers, strings, and more and to keep as much of the type details away from the programmer as possible, PHP converts values from one type to another as necessary.

The conversion of a value from one type to another is called casting. This kind of implicit casting is called type juggling in PHP. The rules of the type juggling done by arithmetic operators are shown in Table 2-4.

A black and white text

Description automatically generated

Some other operators have different expectations of their operands and thus have different rules. For example, the string concatenation operator converts both operands to strings before concatenating them:

3 . 3.74//gives the string 32.74

You can use a string anywhere PHP expects a number. The string is presumed to start with an integer or floating-point number. If no number is found at the start of the string, the numeric value of that string is 0. If the string contains a period (.) or upper- or lowercase e, evaluating it numerically produces a floating-point number. For example:

        "9 Lives" - 1; // 8 (int)

        "3.14 Pies" \* 2; // 6.28 (float)

        "9. Lives" - 1; // 8 (float / double)

        "1E3 Points of Light" + 1; // 1001 (float)

**Arithmetic Operators**

The arithmetic operators are operators you’ll recognize from everyday use. Most arithmetic operators are binary; however, the arithmetic negation and assertation operators are unary. These operators require numeric values, and non-numeric values are converted into numeric values by the rules described in the section “Casting Operators.” The arithmetic operators are:

**Addition (+)**

The result of the addition operator is the sum of the two operands.

**Subtraction (-)**

The result of the subtraction operator is the difference between the two operands –that is, the value of the second operand subtracted from the first.

**Multiplication (\*)**

The result of the multiplication operator is the product of the two operands. For example, 3 \* 4 is 12.

**Division (/)**

The result of the division operator is the quotient of the two operands. Dividing two integers can give an integer (e.g., 4 / 2) or a floating-point result (e.g., 1/2).

**Modulus(%)**

The modulus operator converts bother operands to integers and returns the remainder of the division of the first operand by the second operand. For example, 10 % 6 gives a remainder of 4.

**Arithmetic negation(-)**

The arithmetic negation operator returns the operand multiplied by -1, effectively changing its sign. For example, -(3 - 4) evaluates to 1. Arithmetic negation differs from the subtraction operator, even though both are written as a minus sign. Arithmetic negation is always unary and before the operand. Subtraction is binary and between its operands.

**Arithmetic assertion (+)**

The arithmetic assertion operator returns the operand multiplied by +1, which has no effect. It is used only as a visual cue to indicate the sign of a value. For example, +(3 - 4) evaluates to -1, just as (3-4) does.

**Exponentiation (\*\*)**

The exponentiation operator returns the result of raising $var1 to the power of $var2.

$var1 = 5;

        $var2 = 3;

        echo $var1 \*\* var2; //outputs 125

**String Concatenation Operator**

Manipulating strings is such a core part of PHP applications that PGP has a separate string concatenation operator(.). The concatenation operator appends the righthand operand to the lefthand operand and returns the resulting string. Operands are first converted to strings, if necessary. For example:

$n = 5;

        $s = 'There were '. $n . ' ducks.';

        /\*

        //OUTPUT

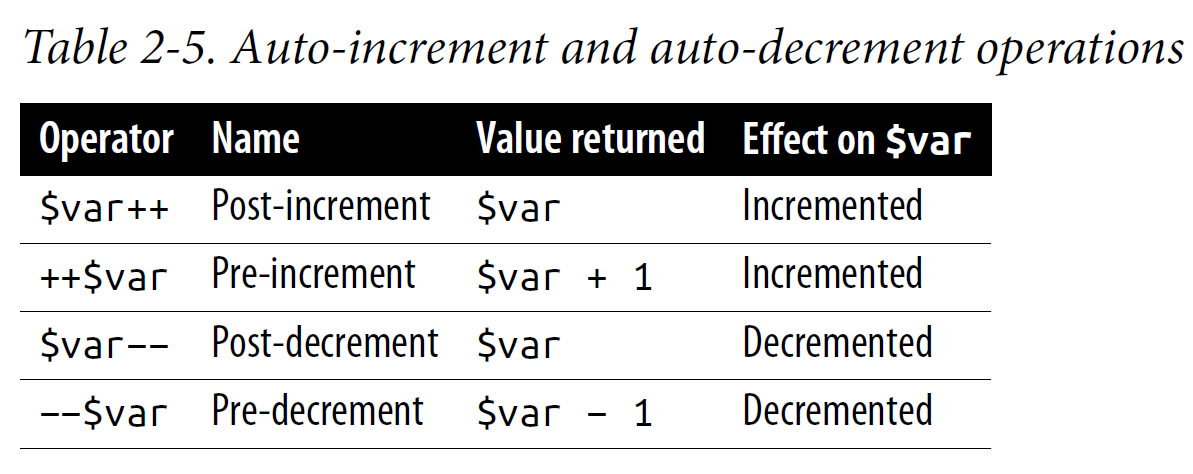
        $s is 'There were 5 ducks'

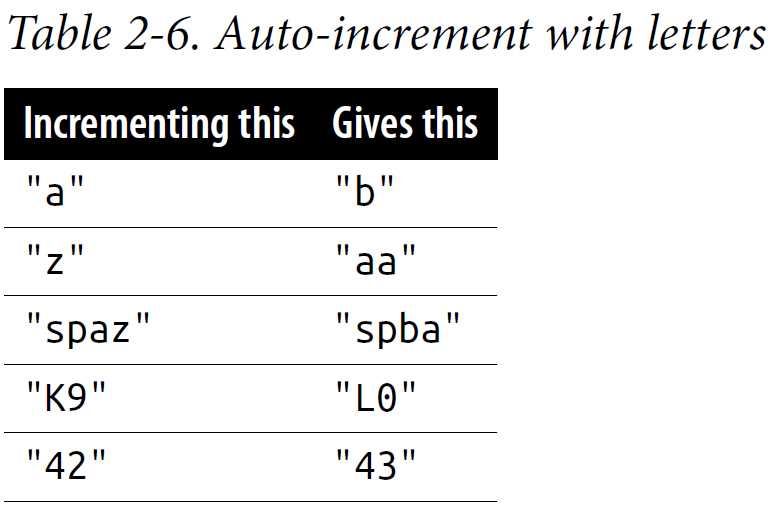
        \*/

The concatenation operator is highly efficient because so much of PHP boils down to string concatenation.

**Auto-Increment and Auto-Decrement Operators**

IN programming, one of the most common operations is to increase or decrease the value of a variable by one. The unary auto-increment (++) and auto-decrement (--) operators provide shortcuts for these common operations. These operators are unique in that they work only on variables; the operators change their operands’ values and return a value.

There are two ways to use auto-increment or auto-decrement in expressions. If you put the operator in front of the operands, it returns the new value of the operands (incremented or decremented). If you put the operator after the operand, it returns the original value of the operand (before the increment or decrement). Table 2-5 lists the different operations.

These operators can be applied to strings as well as numbers. Incrementing an alphabetic character turns it into the next letter in the alphabet. As illustrated in Table 2-6, incrementing “z” or “Z” wraps it back to “a” or “A” and increments the previous character by one (or inserts a new “a” or “A” if at the first character of the string), as through the characters were in a base-26 number system.

**Comparison Operators**

As their name suggests, comparison operators compare operands. The result is always either true if the comparison is truthful or false otherwise.

Operands to the comparison operators can be bother numeric, both string, or one numeric and one string. The operators check for truthfulness in slightly different ways based on the types and values of the operands, whether using strictly numeric comparisons or using lexicographic (textual) comparisons. Table 2-7 outlines when each type of check is used.

A black and white text

Description automatically generated

One important thing to note is that two numeric strings are compared as if they were numbers. IF you have two strings that consist entirely of numeric characters and you need to compare them lexicographically, use the strcmp() function.

The comparison operators are:

**Equality (==)**

If both operands are equal, this operator returns true: otherwise, it returns false.

**Identity (===)**

If both operands are equal and are of the same type, this operator returns true; otherwise, it returns false. Note that this operator does not do implicit type casting. This operator is useful when you don’t know if the values you’re comparing are of the same type. Simple comparison may involve value conversion. For instance, the strings “0.0” and “0” are not equal. The == operator says they are, but === says they are not.

**Inequality (!= or <>)**

If the operands are not equal, this operator returns true; otherwise, it returns false.

**Not identical (!==)**

If the operands are not equal, or they are not of the same type, this operator returns true; otherwise, it returns false.

**Greater than (>)**

If the lefthand operand is greater than the righthand operand, this operator returns true; otherwise, it returns false.

**Greater than or equal to (>=)**

If the lefthand operand is greater than or equal to the righthand operand, this operator returns true; Otherwise, it returns false.

**Less than (<)**

If the lefthand operand is less than the righthand operand, this operator returns truel otherwise, it returns false.

**Less than or equal to (<=)**

If the lefthand operand is less than or equal to the righthand operand, this operator returns true; otherwise, it returns false.

**Spaceship (< = >)**

When the lefthand and righthand operands are equal, this operator returns 0: when the lefthand operand is less than the righthand, it returns -1; and when the lefthand operand is greater than the righthand, it returns 1.

$var1= 5;

        $var2= 65;

        echo $var1 <=> $var2;   //outputs -1

        echo $var2 <=> $var1;   //outputs 2

**Null coalescing operator (??)**

This operator evaluates to the righthand operand if the lefthand operand is NULL;

Otherwise, it evaluates to the lefthand operand.

  $var1= 5;

        $var2= 65;

        echo $var1 <=> $var2;   //outputs -1

        echo $var2 <=> $var1;   //outputs 2

        $var1 = null;

        $var2 = 31;

        echo $var1 ?? $var2; //outputs 31

**Bitwise Operators**

The bitwise operators act on the binary representation of their operands. Each operand is first turned into a binary representation of the value, as described in the bitwise negation operator entry in the following list. All the bitwise operators work on numbers as well as strings, but they vary in their trement of string operands of different lengths. The bitwise operators are:

**Bitwise negation (~)**

The bitwise negation operator changes 1s to 0s and 0s to 1s in the binary representations of the operands. Floating-point values are converted to integers before the operation takes place. If the operand is a string, the resulting value is a string the same length as the original, with each character in the string negated.

**Bitwise AND (&)**

The bitwise AND operator compares each corresponding bit in the binary representation of the operands. If both bits are 1, the corresponding bit in the result is 1; otherwise, the corresponding bit is 0. For example, 0755 & 0671 is 0651. This is easier to understand if we look at the binary representation. Octal 0755 is binary 111101101, and octal 0671 is binary 110111001. We can then easily see which bits are in both numbers and visually come up with the answer:

        111101101

       & 110111001

       - - - - - -

         110101001

The binary number is 110101001. It is octal 0651. (Here’s a tip: split the binary number into three groups—6 is binary 110, 5 is binary 101, and 1 is binary 001;

thus, 0651 is 110101001.) You can use the PHP functions bindec(), decbin(), octdec(), and decoct() to convert numbers back and forth when you are trying to understand binary arithmetic.

If both operands are strings, the operator returns a string in which each character is the result of a bitwise AND operation between the two corresponding characters in the operands. The resulting string is the length of the shorter of the two operands; trailing extra characters in the longer string are ignored. For example, “wolf” & “cat” is “cad”.

**Bitwise OR (|)**

The bitwise OR operator compares each corresponding bit in the binary representations of the operands. If both bits are 0, the resulting bit is 0; otherwise, the resulting bit is 1. For example, 0755 | 020 is 0775.

If both operands are strings, the operator returns a string in which each character is the result of a bitwise OR operation between the two corresponding characters in the operands. The result string is the length of the longer of the two operands, and the shorter string is padded at the end with binary 0s. For example, ‘’pussy’ | “car” is “suwsy”.

**Bitwise XOR (^)**

The bitwise XOR operator compares each corresponding bit in the binary representation of the operands. If either of the bits in the pair, but not both, is 1, the resulting bit is 1; otherwise, the resulting bit is 0. For example, 0755 ^ 023 is 776. If both operands are strings, this operator returns a string in which each character is the result of a bitwise XOR operation are different lengths, the resulting string is the length of the shorter operand, and extra trailing characters in the longer string are ignored. For example, “big drink” ^ “AA” is “#(”.

**Left shift (<<)**

The left-shift operator shifts the bit in the binary representation of the lefthand operand left by the number of places given in the righthand operand. Both operands will be converted to integers if they aren’t already. Shifting a binary number to the left inserts a ) as the rightmost bit of the number and moves all other bits to the left one place. For ex, 3 << 1 (or binary 11 shifted one place left) results in 6 (binary 110).

Note that each place to the left where a number is shifted results in doubling the numbers. The result of left shifting is multiplying the lefthand operand by 2 to the power of the righthand operand.

**Right shift ( >> )**

The right-shift operator shifts the bits in the binary representation of the lefthand operand right by the number of places given in the righthand operand. Both operands will converted to integers if they aren’t already. Shifting a positive binary number to the right inserts a 0 as the leftmost bit of the number and moves all other bits to the right one place. The rightmost bit is discarded. For ex, 13 >> 1 (or binary 1101) shifted one bit to the right, results in 6 (binary 110).

**Logical Operators**

Logical operators provide ways for you to build complex logical expressions. Logical operators treat their operands as Boolean values and return a Boolean value. There are both punctuation and English versions of the operators (|| and or are the same operator). The logical operators are:

**Logical AND (&&, and)**

The result of the logical AND operation is true if and only if both operands are true; otherwise, it is false. If the value of the first operand is false, the logical AND operator knows that the resulting value must also be false, so the righthand operand is never evaluated. This process is called short-circuiting, and a common PHP idiom uses it to ensure that a piece of code is evaluated only if something is true. For ex, you might connect to a database only if some flag is not false:

$result = $flag and mysqlconnect();

The && and ‘and’ operators differ only in their precedence: && comes before ‘and’.

**Logical OR (||, or)**

The result of the logical OR operation is true if either operand is true; otherwise, the result is false. Like the logical AND operator, the logical OR operator is short-circuited. If the lefthand operator is true, the result of the operator much be true, so the right hand operator is never evaluated. A common PHP idiom uses this to trigger an error condition if something goes wrong. For example:

$result = fopen($filename) or exit();

The || and or operators differ only in their precedence.

**Logical XOR (xor)**

The result of the logical XOR operation is true if either operand, but not both, is true; otherwise, it is false.

**Logical negation**

The logical negation operator returns the Boolean value true if the operand evaluates to false, and false if the operand evaluates to true.

**Casting Operators**

Although PHP is a weakly typed language, there are occasions when it’s useful to consider a value as a specific type. The casting operators (int), (float), (string), (bool), (object), and (unset), allows you to force a value into a particular type. To use a casting operator, put the operator to the left of the operand. Table 2-8 lists the casting operators, synonymous operators, and the type to which the operator changes the value.

Table 2-8. PHP casting Operators

A table with black text

Description automatically generated

Casting affects the way other operators interpret a value rather than changing the value in a variable. For example, the code:

$a = "5";

         $b = (int)$a;

Assigns $b the integer value of $a; $a remains the string “5”. To cast the value of the variable itself, you must assign the result of a cast back to the variable:

$a = "5";

         $b = (int)$a; // now $a holds an integer

Not every cast is useful. Casting an array to a numeric type gives 1 (if the array is empty, it gives 0), and casting an array to a string gives “Array” (Seeing this in your output is a sure sign that you’ve printed a variable that contains an array).

Casting an object to an array builds an array of the properties, the mapping property names to values:

class Person{

            var $name = "Fred";

            var $age = 35;

         }

         $o = new Person;

         $a = (array)$o;

         print\_r($a);

         Array ([name] => Fred [age] => 35)

You can cast an array to an object to buil;d an object whose propertiers correspond to the array;s keys and values. For example:

$a = array('name' => "Fred", 'age' => 35, 'wife' => "Wilma");

         $o = (object) $a;

         echo $o->name;

         //output Fred

Keys that are not valid identifiers are invalid property names and are inaccessible when an array is cast to an object but are restored when the object is cast back to an array.

**Assignment**

The basic assignment operator (=) assigns a value to a variable. The lefthand operand is always a variable. The righthand operand can be any expression-any simple literal, variable, or complex expression. The righthand operand’s value is stored in the variable named by the lefthand operand.

Because all operators are required to return a value, the assignment operator returns the value assigned to the variable. For example, the expression $a = 5 not only assigns 5 to $a, but also behaves as the value 5 if used in a larger expression. Consider the following expressions:

$a = 5;

         $b = 10;

         $c = ($a = $b);

The expression $a=$b is evaluated first, because of the parentheses. Now, both $a and $b haveteh same value, 10. Finally, $c is assigned the result of the expression $a = $b, which is the value assigned to the lefthand operand (in this case, $a). When the full expression is done evaluating, all three variables contain the same value 10.

**Assignment with operation**

In addition to the basics assignment operator, there are several assignment operators that are convenient shorthand. These operators consist of binary operators followed directly by an equals sign, and their effect is the same as performing the operation with the full operands, then assigning the result value to the lefthand operand. These assignment operators are:

Puls-equals (+=)

Adds the righthand operand to the value of the lefthand operand, then assigns the result to the lefthand operand. $a += 5 Is the as $a=$a+5.

Minus-equals(-=)

Subtracts the righthand operand fgrom the value of the lefthand operand, then assigns the result to the lfesthand operand.

Divide-equals(/=)

Divides the value of the lefthand operand by the righthand operand, then assigns the result to the lefthand operand.

Multiply-equals(\*=)

Multiplies the righthand operand by the value of the lefthand operand, then assigns the result to the lefthand operand.

Modulud-equals(%=)

Preforms the modulus operation on the value of the lefthand operand and the righthand operand, then assigns the results to the lefthand operand.

Bitwise-XOR-equals(^=)

Preforms a bitwise AND on the value of the lefthand operand and the righthand operand, then assigns the result to the lefthand operand.

BitWise-AND-equals(&=)

Preforms a bitwise AND on the value of the lefthand operand and the righthand operand, then assigns the result to the lefthand operand.

Bitewise-OR-equals(|=)

Preforms a bitwise OR on the value of the lefthand operand and the righthand operand, then assigns the result to the lefthand operand.

Concatenate-equales(.=)

Concatenates the righthand operand to the value of the lefthand operand, then assigns the result to the lefthand operand.

**Miscellaneous Operators**

The remaining PHP operators are for error suppression, executing an external command, and selecting values:

Error suppression(@)

Some operators or sunctions can generate error messages. The error suppression operator, discussed in full in Chatper 17, is used to prevent theses messages from being created.

Execution(‘…’)

The backtick operator executes the string contained between the backticks as a shell command and returns the output

. For example:

$listing = 'ls -ls /tmp';

         echo $listing;

Conditional (? :)

The conditional operator is, depending on the code you look at , either the most overused or most underused operator. It is the only ternary (three-operand) operator and is therefore sometimes just called the ternary operator. The conditional operator evaluates the expression before the ?. If the expression is true, the operator evalutes the expression before the ?. If the expression is true, the operator returns the value of the expression between the ? and :; otherwise, the operator returns the value of the expression after the :. For instance:

a href="<? echo $url; ?>"><? echo $linktest ? $linktest : $url; ?> </a>

If text for the link $url is present in the variable $linktest, it is used as the test for the linkl otherwise, the URL itself is displayed.

Type(instanceof)

The instanceof operator tests whether a variable is an instantiated object of a given class or implements an interface (see Chapter 6 for more information on object and interfaces):

$a = new Foo;

         $isAFoo = $a instanceof Foo; //true

         $isABar = $a instanceof Bar; // false

**Flow-Control Statements**

PHP supports a number of traditional programming constructs for controlling the flow of execution of a program. Conditional statements, such as if/else and switch, allow a program to execute different pieces of code, or none at all, depending on some condition. Loops, such as while and for, support the repeated execution of particular segments of code.

**If**

The if statement checks the truthfulness of an expression and , if the expression is true, evaluates a statement. An if statement looks like:

if (expression) statement

To specify an alternative statement to execute when the expression is false, use the else keyword:

  if(expression)

          statement

          else statement

For example:

if($user\_validate)

          echo "Welcome!";

          else echo "Access Forbidden!";

To include more than one statement within an iof statement, use a block-a set of statements enclosed by curly braces:

 if($user\_validate)

          {

            echo "Welcome!";

            $greeted = 1;

          }

          else {

            echo "Access Forbidden!";

exit;

          }

PHP provides another syntax for blocks in tests and loops. Instead of enclosing the block of statements in curly braces, end the if line with a colon (:) and use a specific keyword to end the block (endif, in this case). For example:

if($user\_validate):

            echo "welcome!";

            $greeted = 1;

          else:

            echo "Access Forbidden!";

            exit;

          endif;

Other statements described in this chapter also have similar alternate syntax styles (and ending keywords); they can be useful if you have large blocks of HTML inside your statements. For example:

<?php if ($user\_validated) : ?>

            <table>

                <tr>

                    <td>First Name</td><td>sophia</td>

                </tr>

                <tr>

                    <td>Last Name:</td><td>Lee</td>

                </tr>

            </table>

            <?php else: ?>

                Please log in.

                <?php endif ?>

Because if is a statement, you can chain(embed) more than one. This is also a good example of how the blocks can be used to help keep things organized:

if($good){

            print("Dandy!");

        }

        else{

            if($error){

                print("Oh, no!");

            }

            else{

                print("I'm ambivalent...");

            }

        }

Suchj chains of if statements are common enough that PHP provides an easier syntax: the elseif statement. For example, the previous code can be rewritten as:

if($good){

            print("Dandy!");

        }

        elseif($error){

            print("Oh,no!");

        }

        else {

            print("I'm ambivalent...");

        }

The ternary conditional operator (? :) can shorten simple true/false tests. Take a common situation, such as checking to see if a given variable is true and printing something if it is. With a normal if/else statement, it looks like this:

<td><?php if($active) {echo "yes"} else {echo "no";} ?></td>

With the ternary conditional operator, it looks like this:

<td><?php echo $active ? "yes" : "no";?></td>

Compare the syntax of the two:

if(expression){true\_statement} else {false\_statement}

(expression) ? true\_expression : false\_expression

The main difference here is that the conditional operator is not a statement at all. This means that it is used on expression, and the result of a completed ternary expression is itdel an expression. In the previous example, the echo statement is inside the if condiotion, while when iused with the ternary operator, it precedes the expression.

**Switch**

The value of a single variable may determine one of a number of different choices (e.g., the variable holds the username and you want to do something different for each user). The switch statement is designed for just this situation.

A switch statement is given an expression and compares its value to all cases in the switch; all statements in a matching case are executed up to the first break keyword it finds. If none match, and a default is given, all statements following the default keyword are executed, up to the first break keyword encountered. For example, suppose you have the following:

if ($name == 'ktatrow'){

            // do something

        }else if($name == 'dawn'){

            // do something

        }else if($name == 'petermac'){

            // do something

        }else if($name == 'bobk'){

            // do something

        }

You can replace that statement with the following switch statement

switch($name){

            case 'ktatrow':

                // do something

                break;

            case 'dawn':

                // do something

                break;

            case 'petermac':

                 // do something

                 break;

            case 'bobk':

                 // do something

                 break;

        }

The alternative syntax for this is:

switch($name):

            case 'ktatrow':

                // do something

                break;

            case 'dawn':

                // do something

                break;

            case 'petermac':

                 // do something

                 break;

            case 'bobk':

                 // do something

                 break;

            endswitch;

Because statements are executed from the matchinf case label to the next break keyword, you can combine several cases in a fall-through. In the following example, “yes” is printed when $name is equal to sylvie or Bruno:

switch($name){

                case'sylvie':// Fall-through

                case'bruno':

                print("yes");

                break;

                default:

                print("no");

                break;

            }

Commentating the fact that you are using a fall-through case in a switch is a good idea, so someonw doesn’t come along at some point and add a break thinking you had forgotten it. You can specify an optional number of levels for the break keyword to break out of. In this way, a break statement can break out of several levels of nested switch statements. An example of using break in this manner is shown in the next section.

**While**

The simplest form of loop is the while statement

while(expression)statement

if the expression evaluates to true, the statement is executed and then the expressuion is re-evaluated (if it is still true, the body of the loop is executed again, and so on). The loop exits when the expression is no longer true (i.e evaluates to false).

As an example, here’s some code that adds the whole numbers from 1 to 10:

$total = 0;

            $i = 1;

            while($i <= 10){

                $total += $i;

                $i++;

            }

The alternative syntax for while has this structure:

while(expr):

                statement;

                more statements;

            endwhile;

For example:

  $total =0;

            $i = i;

            while($i<= 10):

                $total +=$i;

                $i++;

            endwhile;

You can prematurely exit a loop with the break keyword. In the following code, $i never reaches a value of 6, because the loop is stopped once it reaches 5:

 while($i <= 10){

                if($i == 5){

                    break;// breaks out of the loop

                }

                $total +=$i;

                $i++;

            }

Optionally, you can put a number after the break keyword indicating how many levels of loop structures to break out of. In this way, a statement buried deep in nested loops can break out of the outermost loop. For example:

 $i = 0;

            $j = 0;

            while($i <10){

                while($j <10){

                    if($j == 5){

                        break 2;// breaks out of two while loops

                    }

                    $j++;

                }

                $i++

            }

            echo"{$i}, {$j}";

            //OUTPUT 0,5

The continue statement skips ahead to the next test of the loop condition. As with the break keyword, you can continue through an optional number of levels of loop structure:

 while($i < 10){

                $i++;

                while($j < 10){

                    if($j ==5){

                        continue 2; // continues through two levels

                    }

                    $j++;

                }

            }

In this code, $j never has a value above 5, but $i goes through all values from 0 1- 9. PHP also supports a do/while loop, which takes the following form:

do

             statement

            while (expression)

Use a do/while loop to ensure that the loop body is executed at least once (the first time):

$total = 0;

            $i = 1;

            do{

                $total += $i++;

            }while ($i <= 10);

You can use break and continue statements in a do/while statement just as in a normal while statement.

The do/while statement is sometimes used to break out of a bloc of code when an error condition occurs. For example:

do{

                // do some stuff

                if($errorCondition){

                    break;

                }

                // do some other dtuff

            }while (false);

Because the condition for the loop is false, the loop is executed only once, regardless of what happens inside the loop. However, if an error occurs, the code after the break is not evaluated.

**For**

The for statement is similar to the while statement, except it adds counter initializarion and counter manipulation expressions, and is often shorter and easier top read than the equivalent while loop.

Here’s a while loop that counts from 0 to 9, printing each number:

$counter = 0;

            while($counter< 10)

            {

                echo"Counter is {$counter} <br/>";

                $counter++;

            }

Heres the corresponding, more concise for loop:

for($counter = 0; $counter < 10; $counter++){

                echo "Counter is $counter <br/>";

            }

The structure of a for statement is:

for(start; condition; increment){statement(s);}

The expression start is evaluated once, at the beginning of the for statement. Each time through the loop, the expression condition is tested. If it is true, the body of the look is executed if it is false, the loop ends. The expression increment is evaluated after the loop body runs.

The alternative syntax of a for statement is:

for(expr1; expr2; expr3):

                statement;

                ...;

            endfor;

This program adds the numbers from 1 to 10 using a for loop:

for($i = 1; $i<=10; $i++){

                $total +=$i;

            }

Here’s the same loop using the alternate syntax:

for ($i = 1; $i <= 10; $i++):

            $total += $i;

            endfor;

You can specify multiple expressions for any of the expressions in a for statement by separating the expressions with commas. For example:

$total = 0;

            for($i=0, $j=1; $i<=10; $i++,$j\*=2){

                $total =0;

            }

You can also leave an expression empty, signaling that nothing should be done for that phase. In the most degenerate form, the for statement becomes an infinite loop. Yoy probably don’t want to run this example as it never stops printing:

for(;;){

                echo"Can't stop me <br />";

            }

In for loops, as in while loops, you can use the break and continue keywords to end the loop or current iteration.

**Foreach**

The foreach statement allows you to iterate over elements in an array. The two forms of the forwach statement are further discussed in Chapter 5, where we talk in more depth about arrays. To loop over an array, accessing the value at each key use:

foreach($array as $current){

                //...

            }

The alternate syntax is

foreach($array as $current):

                //...

            endforeach;

To loop over an array, accessing both key and value, use:

foreach($array as $key => $value)

            {

                //...

            }

The alternate syntax is

foreach($array as $key => $value):

                //...

            endforeach;

**try…catch**

The try…catch construct is not so much a flow-control structure as it is a more graceful way to handle system srrors. For example, if you want to ensure that your web supplication has a valid connection to a datebase before continuing, you could write code like this:

try{

                $dbhandle = new PDO('mysql:host=localhost; dbname=library', $username, $pwd);

                doDB\_Work($dbhandle);// call function on gaining a connection

                $dbhandle = null;

            }

            catch (PDOExceptation $error){

                print "Error!: " . $error->getMessage() . "<br/>";

                die();

            }

Here the connection is attempted with the try portion of the construct and if there are any errors with it, the flow of the code automatically falls into the catch portion, where the PDO exception class is instaniated into the $error variable. It can then be displayed on the screen, and the code can “gracefully” fail, rather than making an abrupt end. You can even redirect to try connecting to an alternate database, or respond to the error any other way you wish within the catch portion.

**Declare**

The declare statement allows you to specify execution directives for a block of code. The structure of a declare statement is:

declare (directive) statement

Currently, there are only three declare forms: the ticks, encoding, and strict\_types directives. You can use the ticks directive to specify how frequently (measured roughly in number of code statments) a tick function is registered when register\_tick\_function() is called. For example:

register\_tick\_function("someFunction");

            declare(ticks = 3){

                for($i-0; $i <10; $i++){

                    //do somthing

                }

            }

In this code, someFunction() is called every third statement within the block is executed. You can use the encoding directive to specify a PHP script’s output encoding. For example:

declare(encoding="UTF-8");

This form of the declare statement is ignored unless you compiler PHP with the –enable-zend-multibype option. Finally you can use the strict\_type directive to enforce the use of a strict data type when defining and using variables.

**exit and return**

As soon as it is reached the exit statement ends the script’s execution. The return statement returns from a function or, at the top level of the program, from the script. The exit statement takes an sptional value. If this is a number, it is the exit status of the process. IF it is a string, the value is printed before the process terminates. The function die() is an alias for this form of the exit statement

$db = mysql\_connect("localhost", $USERNAME,$PASSWORD);

            if(!$db){

                die("Could not connect to database");

            }

This is more commonly written as:

$db = mysql\_connect("localhost",$USERNAME,$PASSWORD) or

            die("could not connect to database");

See Chapter 3 for more information on using the return statement in functions.

**Goto**

The goto statement allows execution to “jump” to another place in the program. You specify execution points by adding a label which is an identifier followed by a colon (:). You when jump to the label from another location in the script via the goto statement:

for($i=0; $i < $count; $i++)

            {//oops, fount an error

                if($error){

                    goto cleanup;

                }

            }

            cleanup:

            // do some cleanu

You can only goto a label within the same scope as the goto statement itself, and you can’t jump into a loop or switch. Generally, anywhere you might use a goto (or multilevel break statement, for that matter), you can rewrite the code to be cleaner without it.

**Including Code**

PHP provides two constructs to load code and HTML from another module: require and include. Both load a file as a PHP script runs, work in conditionals and loops, and complain if the file being loaded cannot be found. Files are located by an included file path as part of the directive in the user of the function, or based on the setting of include\_path in the php.ini file. The include\_path can be overridden by the set\_include\_path() function. If all these avenues fail, PHP’s last attempt is to try to find the file in the same directory as the calling script. The main difference is that attempting to require a nonexistent file is a fatal error, while attempting to include such a file produces a warning but does not stop script execution.

A common use of include is to separate page-specific content from general site design. Common elements such as headers and footers fo in separate HTML files, and each page then looks like:

<?php include "header.html";?>

            content

            <?php include "footer.html";?>

We use include because it allows PHP to continue to process the page event if there’s an error in the site design file(s). The required construct is less forgiving, and it is more suited to loading code libararies when the page cannot be displayed if the libraries do not load. For example:

require"codelib.php";

            mysub(); // defined in codelib.php

A marginally more efficient way to handle headers and footers is to load a single file and then call functions to generate the standardized site elements:

<?php require "design.php"; header();?>

            content

            <?php footer();

If PHP cannot parse some part of a file added by include or require, a warning is printed and executed continues. You can silence the warning by prepending the call with the silence operator(@)-for example, @include. If the allow\_url\_fopen option is enabled through PHP’s configuration file, php.ini, you can include files from a remote site by providing a URL instead of a simple local path:

include "http://www.example.com/codelib.php";

If the filename begins with http://, https://, or ftp://, the file is retrieved from a remote site and loaded. Files included with include and require can be arbitrarily named. Common extensions are .php,.php5,and .html.

NOTE: that remotely fetching a file that ends in .php from a web server that has PHP enabled fetches the output of that PHP script. It executes the PHP code in that file.

If a program uses include or require to include the same file twice (mistakenly done in a loop, for example), the file is loaded, and the code is run, or the HTML is printed twice. This can result in errors about the redefinition of functions or multiple copies of headers or HTML being sent. To prevent these errors from occurring, use the include\_once and require\_once constructs. They behave the same as include and require the first time a file is loaded, but quietly ignore subsequent attempts to load the same file. For example, many page elements, each stored in separate files, need to know the current user’s preferences. The element libraries should load the user preferences libaray with require\_once. The page designer can then include a page element without worrying about whether the user preference code has already been loaded.

Code in an included file is imported at the scope that is in effect where the include statement is found, so the included code can see and alter your cod’s variables. This can be useful-for instance, a user-tacking library might store the current user’s name in the global $user variable:

// main page

            include "userprefs.php";

            echo"Hello, {$user}.";

The ability of libraries to see and change your variables can also be a problem. You have to know every global variable used by a library to ensure that you don’t accidentally try to use one of them for your own purposes, thereby overwriting the libaray’s value and disrupting how it works.

If the include or require construct is in a function, the variables in the included file become function-scope variables for that function.

Because include and require are keywords, not real statements, you must always enclose them in curly braces in conditional and loop statements:

for($i=0; $i<10; $i++){

            include "repeated\_element.html";

        }

Use the get\_included\_files() function to learn which files your script has included or required. It returns any array containing the full system path filnames of each included or requitred file. Files that did not parse are not included in this array.

**Embedding PHP in Web Pages**

Although it is possible to write and run standalone PHP programs, most PHP code is embedded in HTML or XML files. This is, after all, why it was created in the first place/ Processing such documents involves replacing each chunk of PHP source code with the output it produces when executed.

Because a single file usually contains PHP and non-PHP source code, we need a way to identify the regions of PHP code to be executed. PHP provides four different ways to do this.

As you’ll see, the first and preferred method looks like XML. The second method looks like SGML. The third method is based on ASP tags. The fourth method uses the standard HTML <script> tag; this makes it easy to edit pages with enabled PHP using a regular HTML editor.

**Standard (XML) Style**

Because of the advent of the eXtensible Markup Language (XML) and the migration of HTML to an XML language (XHTML), the currently preferred technique for embedding PHP uses ZML-compliant tags to denote PHP instructions.

Coming up with tags to demark PHP commands in XML was easy, because XML allows the definition of new tags. To use this style, surround your PHP code with <?php and ?>. Everything between these markers is interpreted as PHP, and anything outside the markers is not. Although it is not necessary to include spaces between the markers and the enclosed test, doing so improves readability. For example, to get PHP to print “Hello, world,” you can insert the following line in a web page:

<?echo "Hello, world"; ?>

The trailing semicolon on the statement is optional, because the end of the block also forces the end of the expression. Embedded in a complete HTML file, this looks like:

<!DOCTYPE html>

<html>

<head>

    <meta charset='utf-8'>

    <meta http-equiv='X-UA-Compatible' content='IE=edge'>

    <title>This is my first PHP program</title>

    <meta name='viewport' content='width=device-width, initial-scale=1'>

    <link rel='stylesheet' type='text/css' media='screen' href='main.css'>

    <script src='main.js'></script>

</head>

<body>

    <p>

        Look, ma! It's my first PHP program:<br/>

        <?php echo "Hello, world";?><br/>

        how cool is that?

    </p>

</body>

</html>

Notice that there’s no trace of the PHP course code from the original file. The user sees only its output. Also, notice that we switched between PHP and non-PHP, all in the space of a single line. PHP instructions can be put anywhere in a file, even within valid HTML tags. For example:

<input type="text" name="first\_name" value="<?php echo Peter"; ?>"/>

When PHP is done with this text, it will read:

<input type="text" name="first\_name" value="Peter"/>

The PHP code within the opening and closing markers does not have to be on the same line. If the closing marker of a PHP instruction is the last thin on a line, the line break following the closing tag is removed as well. Thus, we can replace the PHP instructions in the “Hello, world” example with:

<?php

        echo "Hello, world";

        ?><br/>

With no change in the resulting HTML.

**SGML Style**

Another style of embedding PHP comes from SQML instruction processing tags. To use this method, simply enclose the PHP in <? And ?>. Here’s the “Hello, world” example again:

<? echo "Hello, world";?>

This style, known as short tags, is off by default. You can turn on support for short tags by building PHP with the –enable-short-tags option or enable short\_open\_tag in the PHP configuration file. This is discouraged as it depends on the state of this setting; if you export your code to another platform, it may or may not work. The short echo tag, <?= … ?>, is available regardless of the availability of short tags.

**Echoing Content Directly**

Perhaps the single most common operation within a PHP application is displaying data to the user. In the context of a web application, this means inserting into the HTML document information that will become HTML when viewed by the user. To simplify this operation, PHP provides a special version of the SQML tags that automatically take the value inside the tag and insert it into the HTML page. To use this feature, add an equal sign (=) to the opening tag. With this technique, we can rewrite our form example as:

<input type="text" name="first\_name" value="<?= "Dawn"; ?>">

**What’s Next**

Now that you have the basics of the language under your belt-a foundational understanding of what variables are and how to name them, what data types are, and how code flow control works-we will move on to some finer details of the PHP language. Next we’ll cover three topics that are so important to PHP that they each have their own dedicated chapters: how to define functions (Chapter 3), manipulate strings (Chapter 4), and manage arrays (Chapter 5).

**CHAPTER 3**

**Functions**

A function is a named block of code that performs a specific task, possibly acting upon a set of values given to it, aka parameters, and possibly returning a single value or set of values in an array. Functions save on compile time-no matter how many times you call them, functions are compiled only once for the page. They also improve reliability by allowing you to fix any bugs in one place rather than everywhere you perform a task, and they improve readability by isolating code that performs specific tasks.

This chapter introduces the syntax of functions called and function definitions and discusses how to manage variables in functions and pass values to functions (including pass-by-value and pass-by-reference). It also covers variable functions and anonymous functions.

**Calling a Function**

Functions in a PHP program can be built in (or, by being in an extension, effectively built-in)or user-defined. Regardless of their source, all functions are evaluated in the same way:

$someValue = function\_name([parameter, ...]);

The number of parameters a function requires differs from function to function (and, as we’ll see later, many even vary from the same function). The parameters supplied to the function may be any valid expression and must be in the specific order expected by the function. If the parameter are given out of order, the function may still run by a fluke, but it’s basically a case of “garbage in = garbage out.” A function’s documentation will tell you what parameters the function excepts and what value(s) you can expect to be returned. Here are some examples of functions:

$someValue = function\_name([parameter, ...]);

        // strlen() is a PHP built-in function that returns the length of a string

        $length=strlen("PHP");// $length is now 3

        // sin() and asin() are the sine and arcsine math functions

        $result = sin(asin(1));// $result is the sine of arcsin(1), or 1.0

        // unlink() deletes a file

        $result=unlink("functions.txt");

        // $results = true or false depending on success or failure

In the first example, we give an argument, “PHP”, to the function strlen(), which gives us the number of characters in the provided string. In this case, it returns 3, which is assigned to the variable $length. Thiis is the simplest and most common way to use a function.

The second example passes the results of asin(1) to the sin() function. Since the sine and arcine functions are inverses, taking the sine of the arcsine of any value will always return that same value. Here we see that a function can be called within another function. The returned value of the inner call is subsequently sent to the outer function before the overall result is returned and stored in the $result variable.

In the final example, we give a filename to the unlink() function, which attempts to delete the file. Like many functions, it returns false when it fails. This allows you to use another built-in function, die(), and the short-circuiting property of the logic operators. Thus, this example might be rewritten as:

$result = unlink("functions.txt") or die("Operation failed!");

The unlink() function, unlike the other two examples, affects something outsode of the parameters given to it. In this case, it deletes a file from the filesystem. All such side effects of a function should be carefully documented and carefully considered.

PHP has a huge array of functions already defined for you to use in your programs. These extensions preform tasks such as accessing databases, creating graphics, reading and writing XML files, grabbing files from remote systems, and more. Not all functions return a value. They can perform an action like sending an email and then just return controlling behavior to the calling code; having completed their task, they have nothing to “say”.

**Defining a Function**

To define a function, use the following syntax:

function[&] function\_name([parameter[, ...]]){

            statement list

        }

The statement list can include HTML. You can declare a PHP function that doesn’t contain any PHP code. For instance, the column() function simply gives a convenient short name to HTML code that may be needed many time throughout the page:

<?php function column()

        {?>

         </td></td> <?php

        }

The function name can be any string that starts with a letter or underscore followed by zero or more letters, underscores, and digits. Function names are case-insensitive; that is, you can call the sin() function as sin(1), SIN(1), SiN(1), and so on, because all these names refer to the same function. By convention, built-in PHP functions are called with all lowercase.

Typically, functions return some value. To return a value from a function use the return statement: put return expr inside your function. When a return statement is encountered during execution, control reverts to the calling statement, and the evaluated results of expr will be returned as the value of the function. You can include any number of return statements in a function (for example, if you have a switch statement to determine which of several values to return).

Let’s take a look at a simple function. Example 3-1 takes two strings, concatenates them, and then returns the result (in this case, we’ve created a slightly slower equivalent to the concatenation operator, but bear with us for the sake of the example).

<html>

    <head>

        <title> 3\_1\_StringConcatenation</title>

    </head>

    <body>

        <?php

        function strcat($left, $right)

        {

            $combinedString = $left . $right;

            return $combinedString;

        }

        ?>

    </body>

</html>

The function takes two arguments, $left and $right. Using the concatenation operator, the function creates a combined string in the variable $combinedString. Finally, in order to cause the function to have a value when it’s evaluated with our arguments, we return the value $combinedString.

Because the return statement can accept any expression, even comple ones, we can simplify the program as shown here:

 function strcat2($left,$right){

            return $left . $right;

        }

<html>

    <head>

        <title> Look Out World</title>

    </head>

    <body>

        <?php

            function strcat($left,$right)

            {

                return $left.$right;

            }

            $first = "This is a ";

            $second = " complete sentence!";

            echo strcat($first,$second);

        ?>

    </body>

</html>

When this page is displayed, the full sentence is shown.

A screen shot of a computer

Description automatically generated

In this next example a function takes in an integer, doubles it by bit-shifting the original value, and returns the result:

function doubler($value)

        {

            return $value << 1;

        }

Once the function is defined, you can use it anywhere on the page. For example:

<?php echo "A pair of 13s is ". doubler(13); ?>

You can nest function declarations, but with limited effect. Nesting declarations do not limit the visibility of the inner-defined function, which may be called from anywhere in your program. The inner function does not automatically get the outer function’s arguments. And. Finally, the inner function cannot be called until the outer function has been called, and also cannot be called from code parsed agter the outer function:

function outer ($a){

            function inner ($b){

                echo "There $b";

            }

            echo "$a, hello";

        }

        //outputs "Well, hello there reader"

        outer("well");

        inner("reader");

**Variable Scope**

If you don’t use functions, any variable you create can be used anywhere in a page. With functions, this is not always true. Functions keep their own sets of variables that are distinct from those of the page and of other functions.

The variables defined in a function, including its parameters, are not accessible outside the function, and, by default, variables defined outside a function are not accessible inside the function. The following example illustrates this:

<?php

        // This code is broken on purpose

            $a=3;

            function foo()

            {

                $a += 2;

            }

            foo();

            echo $a;

        ?>

The variable $a inside the function foo() is a different variable than the variable $a outside the function; even though foo() uses the add-and -assign operator, the value of the outer $a remains 3 throughout the life of the page. Inside the function, $a has the value 2.

As we discussed in Chapter2, the extent to which a variable can be seen in a program is called the scope of the variable. Variables created within a function are inside the scope of the function (i.e, have function-level scope). Variables created outside of functions and objects have global scope and exist anywhere outside of those functions and objects. A few variables provided by PHP have both function-level and global scope (often referred to as super-global variables).

At first glance, even an experienced programmer may think that in the previous example $a will be 5 by the time the echo statement is reached, so keep that in mind when choosing names for your variables.

**Global Variables**

If you want a variable in the global scope to be accessible from within a function, you can use the global keyword. Its syntax is:

global var1, var2,...;

Changing the previous example to include a global keyword, we get:

<?php

        $a=3;

        function foo()

        {

            global $a;

            $a += 2;

        }

        foo();

        echo $a;

        ?>

Instead of creating a new variable called $a with function-level scope, PHP uses the global $a within the function. Now, when the value of $a is displayed, it will be 5.

You must include the global keyword in a function before any uses of the global variable or variables you want to access. Because they are declared before the body of the function, function patameters can never be global variables.

Using global is equivalent to creating a reference to the variable in the $GLOBALS variable. That is both of the following declarations create a variable in the functions’s scope that is a reference to the same value as the variable $var in the global scope:

global $var;

        $var = & $GLOBALS['var'];

**Static Variables**

Like C, PHP supports declaring function variables as static. A static variable retauins its value between all cakks to the function and is initialized during a script’s execution only the first time the function is called. Use the static keyword at the first use of a function variable to declare it as static. Typically, the first use of a static variable assigns an initial value:

static var [= value][,...];

In Example 3\_5, the variable $count is incremented by one each time the function is called.

<?php

        function counter()

        {

            static $count = 0;

            return $count++;

        }

        for($i =1; $i <= 5; $i++){

            print counter();

        }

        ?>

When the function is called for the first time, the static variable $count is assigned a value of 0. The value is returned and $count is incremented. When the function ends, $count is not destroyed like a nonstatic variable, and its value remains the same until the next time counter() is called. The for loop displayed the number from 0 to 4.

**Function Parameters**

Functions can expect an arbitrary number of arguments, declared by the function definition. There are two different ways to pass parameters to a function. The first, and more common, is by value. The second is by reference.

**Passing Parameters by Value**

In most cases, you pass parameters by value. The argument is any valid expression. That expression is evaluated, and the resulting value is assigned to the appropriate variable in the function. In all of the examples so far, we’ve been passing arguments by value.

**Passing Parameters by Reference**

Passing by reference allows you to override the normal scoping rules and give a function direct access to a variable. To be passed by reference, the argument must be a variable; you indicate that a particular argument of a function will be passed by reference by preceding the variable name in the parameter list with an ampersand (&). Example 3\_6 revisits our doubler() function with a slight change.

<?php

            function doubler(&$value){

                $value = $value << 1;

            }

            $a = 3;

            doubler($a);

            echo $a;

        ?>

Because the function’s $value parameter is passed by reference, the actual value of $a, rather than a copy of that value, is modified by the function. Before, we had to return the double value, but now we change the caller’s variable to be the doubled value.

This is another place where a function has side effects: since we passed the variable $a into doubler() by reference, the value of $a is at the mercy of the function. In this case, doubler() assigns a new value to it.

Only variables-and not constants-can be supplied to parameters declared as passing by reference. Thus, if we included the statement <?php echo double(7); ?> in the previous example, it would issue an error. However, you may assign a default value to parameters passed by reference (in the same manner as you provide default values for parameters passed by value).

Even in cases where your function does not affect the given value, you may wany a parameter to be passed by reference. When passing a value, PHP must copy the value. Particularly for large strings and objects, this can be an expensive operation. Passing by reference removes the need to copy the value.

**Default Parameters**

Sometimes a function may need to accept a particular parameter. For example, when you call a function to get the preferences for a site, the function may take in a parameter with the name of the preference to retrieve. Rather than using some special keyword to designate that you want to retrieve all of the preferences, you can simply not supply any argument. This behavior works by using default arguments.

To specify a default parameter, assign the parameter value in the function declaration. The value assigned to a parameter as a default value cannot be a complex expression, only a scalar value:

function getPreferences($whichPreference = 'all'){

                // If $whichPreference is "all", return all prefs;

                //otherwise, get the specific preference requested...

            }

When you call getPreferences(), you can choose to supply an argument. If you do, it returns the preference matching the string you give it; if not, it returns all preferences.

NOTE: A function may have any number of parameters with default values. However, these defaulted parameters must be listed after all parameters that do not have default values

**Variable Parameters**

A function may require a variable number of arguments. For example, the getPreferences() example in the previous section might return the preferences for any number of names, rather than for just one. To declare a function with a variable number of arguments, leave out the parameter block entirely:

function getPreferences()

            {

                // some code

            }

PHP provides three functions you can use in the function to retrieve the parameters passed to it. func\_get\_args() returns an array of all parameters provided to the function; func\_num\_args() returns the number of parameters provided to the function; and func\_get\_arg() returns a specific argument from the parameters. For example:

$array = func\_get\_arg();

            $count = func\_num\_args();

            $value = func\_get\_arg(argument\_number);

In Example 3-7, the count\_list() function takes in any number of arguments. It loops over those arguments and returns the total of all the values. If no parameters are given, it returns false.

 <?php

        function countList()

        {

            if(func\_num\_args() == 0)

            {

                return false;

            }

            else{

                $count = 0;

                for($i = 0; $i < func\_num\_args(); $i++)

                {

                    $count += func\_get\_arg($i);

                }

                return $count;

            }

        }

        echo countList(1,5,9);// outputs "15"

        ?>

The result of any of these functions cannot directly be used as a parameter to another function. Instead, you must first set a variable to the result of the function, and then use that in the function call. The following expression will not work:

foo(func\_get\_arg());

Instead, use:

$count = func\_get\_arg();

        foo($count);

**Missing Parameters**

PHP lets you be as lazy as you want-when you call a function, you can pass any number of arguments to the function. Any parameters the function expects that are not passed to it remain unset, and a warning is issued for each of them: NOTE THIS IS INCORRECT. THIS IS BROKEN

function takesTwo($a, $b){

            if(isset($a))

            {

                echo " a is set\n";

            }

            if(isset($b)){

                echo " b is set\n";

            }

        }

        echo "With two arguments:\n";

        takesTwo(1,2);

        echo "With one argument:\n";

        takesTwo(1);

**Type Hinting**

When defining a function, you can add type hinting—that is, you can require that a parameter be an instance of a particular class (including instances of classes that extend that class), an instance of a class that implements a particular interface, an array or a callable. To add type hinting to a parameter, include the class name, array, or callable before the variable name in the function’s parameter list. For example

<?php

        class Entertainment{}

        class Clown extends Entertainment{}

        class Job{}

        function handleEntertainment(Entertainment $a, callable $callback = Null)

        {

            echo "Handling " . get\_class($a) . " fun\n";

            if($callback !== NULL){

                $callback();

            }

        }

        $callback = function()

        {

            //do something

        };

        handleEntertainment(new Clown); // works

        handleEntertainment(new Job, $callback); // runtime error

        ?>

A type-hinted parameter must be NULL, an instance of the given class or a subclass of the class, an array, or callable as a specified parameter. Otherwise, a runtime error occues. You can define a data type for a property in a class.

**Return Values**

PHP functions can return only a single value with the return keyword:

function returnOne(){

            return 42;

        }

To return multiple values, return an array:

 function returnTwo()

        {

            return array("Fred",35);

        }

If no return value is provided by a function, the function returns NULL instead. You can set a return data type by declaring it in the function definition. For example, the following code will return an integer of 50 when it is executed:

By default, values are copied out of the function. To return a value by reference, prepend the function name with & both when declaring it and when assigning the return value to a variable:

$names = array("Fred","Barney", "Wilma", "Betty");

        function &findOne($n){

            global $names;

            return $names[$n];

        }

        $person =& findOne(1); //Barney

        $person = "Barnetta";  // changes $names[1]

        ?>

In this code, the findOne() function returns an alias for $names[1] instead of a copy of its value. Because we assign by reference, $person is an alias for $names[1], and the second assignment changes the value in $name[1]. This technique is sometimes used to return large string or array values efficiently from a function. However, PHP implements copy-on-wrote for variable values, meaning that returning a reference from a function is typically unnecessary. Returning a reference to a value is slower than returning the value itself.

**Variable Functions**

As with variable variables where the expression refers to the value of the variable whose name is the value held by the apparent variable (the $$ construct), you can ass parentheses after a variable to call the function whose name is the value held by the apparent variable- for example, $variable(). Consider this situation, Where a variable is used to determine which of three functions to call:

switch($which)

        {

            case 'first':

            first();

            break;

            case 'second':

            second();

            break;

            case 'third'

            third();

            break;

        }

In this case, we could use a variable function call to call the appropriate function. To make a variable function call, include the parameters for a function in parentheses after the variable. To rewrite the previous example:

$which(); // if $which is "first", the function first() is called, etc...

If no function exists for the variable, a runtime error occurs when the code is evaluated. To prevent this, before calling the function you can use the built-in function\_exists() function to determine whether a function exists for the value of the variable:

$yesOrNo = function\_exists(function\_name);

For example:

if(function\_exists($which))

        {

            $which(); // If $Which is "First", the function first() is called, etc...

        }

Language constructs such as echo() and issest() cannot be called through variable functions:

$which = "echo";

        $which("hello, world"); // does not work

**Anonymous Functions**

Some PHP functions do part of their work by using a function you provide to them. For example, the usort() function uses a function you create and pass to it as a parameter to determine the sort order of the items in an array.

Although you can define a function for such purposes, as shown previously, these functions tend to be localized and temporary. To reflect the transient nature of the callback, create and use an anonymous function (also know as a closure).

You can create an anonymous function using the normal function definition syntax, but assign it to a variable or pass it directly.

<?php

        $array = array("really long string here, boy", "this", "middling length", "larger");

        usort($array, function($a, $b){

            return strlen($a)-strlen($b);

        });

        print\_r($array);

        ?>

The array is sorted by usort() using the anonymous function, in order of string length. Anonymous functions can use the variables defined in their enclosing scope using the use syntax. For example

<?php

            $array = array("really long string here, boy", "this", "middling length", "larger");

            $sortOption = 'random';

            usort($array, function($a,$b) use ($sortOption){

                if($sortOption == 'random'){

                    // sort randomly by returning (-1,0,1) at random

                    return rand(0,2)-1;

                }

                else{

                    return strlen($a)-strlen($b);

                }

            });

            print\_r($array);

        ?>

Note that incorporating variables from the enclosing scope is not the same as using global variables—global variables are always in the global scope, while incorporating variables allows a closure to use the variables defined in the enclosing scope. Also note that this is not necessarily the same as the scope in which the closure is called. For example:

<?php

            $array = array("really long string here, boy", "this", "middling length", "larger");

            $sortOption = 'random';

            function sortNonrandom($array){

                $sortOption = false;

                usort($array, function($a,$b) use ($sortOption){

                    if($sortOption == "random"){

                        // sort randomly by returning (-1, 0, 1) at random

                        return rand(0,2)-1;

                    }

                    else{

                        return strlen($a)-strlen($b;)

                    }

                });

                print\_($array);

            }

            print\_r(sortNonrandom($array))

        ?>

In this example, $array is sorted normally, rather than randomly—The value of $sortOption indide the closure is the value of $sortOption in the scope of sortNonrandom(), not the value of $sortOption in the global scope.

**What’s Next**

User-defined functions can be confusing to write and complex to debug, so be sure to test them well and to try to limit them to performing one task each. In the next chapter we’ll be looking at strings and everything that they entail, which is another complex and potentially confusing topic. Don’t get discouraged: remember that we are building strong foundations for writing good, solid, succinct PHP code. Once you have a firm grasp of the key concepts of functions, strings, arrays, and objects, you’ll be well on your way to becoming a good PHP developer.

**Chapter 4**

**Strings**

Most data you encounter as you program will be sequences of characters or strings. Strings can hold people’s names, passwords, addresses, credit card numbers, links to photographs, purchase histories and more. For that reason, PHP has an extensive selection of functions for working with strings .\

This chapter shows the many ways to create strings in your programs, including the sometomes tricky subject of interpolation (placing a variable’s value into a string), then covers functions for changing, quoting, manipulating, and searching strings. By the end of this chapter, you’ll be a string-handling expert.

**Quoting String Constants**

There are four ways to write a string literal in your PHP code: using single quotes, double quotes, the here document (heredoc) format derived for the Unix shell, and its “cousin” now document (nowdoc). Theses methods differ in whether they recognize special exceapte sequences that let you encode other character or interpolate variables.

**Variable Interpolation**

When you define a string literal using double quotes or a heredoc, the string is subject to variable interpolation. Interpolation is the process of replacing variable names in the strings with their contained values. There are two ways to interpolate variables into strings.

The simpler of the two ways is to put the variable named in a double-quoted string or in a heredoc:

<?php

        $who = 'Kilroy';

        $where = 'here';

        echo "$who was $where";

        //output Kilroy was here

        ?>

The other way to surround the variable bneing interpolated with curly braces. Using this syntax ensures the correct vbariable is interpolated. The classic uuse of curly braces its to disambiguate the variable named from any surrounding text:

$n = 12;

        echo "You are the {$n}th person";

        //output You are the 12th person

Without the cury braces, PHP would try to print the value of the $nth variable.

Unlike in some shell environments, in PGP, strings are not repeatedly processed for interpolation. Instead, any interpolations in a double-quoted string are processed first and the result is used as the value of the string:

  $bar = 'this is not printed';

        $foo = '$bar'; // single quotes

        print("$foo");

**Single-Quoted Strings**

Single-quoted strings and nowdocs do not interpolate variables. Thus, the variable named in the following string is not expanded because the string literal in which it occurs is single-qouted:

$name = 'Fred';

        $str = 'Hello, $name';// single-quoted

        echo $str;

        //Output Hello, $name

The only escape sequences that work in single-quoted strings are \’, which puts a single quote in a single-quoted string, and \\, which puts a backslash in a single quoted string. Ant other occurrence of a backslash is interpreted simply as a backslash:

echo $name;

        $path = 'C:\\WINDOWS'; // escaped backslash

        echo $nope;

        //Output Luka O' Bostick

        //c:\WINDOWS

**// \n**

**Double-Quoted Strings**

Double-quoted strings interpolate variables and expand the many PHP escape sequences. Table 4-1 lisits the escape sequences recognized by PHP in double-quoted strings.

A table with text and a list of characters

Description automatically generated

If an unknown escape sequence (i.e, a backslash followed by a character that is not one of those in table 4-1) is found in a double-quoted string literal, it is ignored (if you have the warning level E\_Notice set, a warning is generated for such unknown escape sequences):

**Here Documents**

You can easily put multiline strings into your program with a heredoc, as follows:

$clerihew = <<< EndOfQuote

    Sir Humphrey Davy

    Abominated gravy.

    He lived in the odium

    of having discovered sodium.

    EndOfQuote;

    /\*  OutPut

    Sir Humphrey Davy

    Abominated gravy.

    He lived in the odium

    of having discovered sodium.

    \*/

    echo $clerihew;

The <<< identifier token tells the PHP parser that you’re writing a heredoc. You get to pick the identifier (“EndOfQoute” in this case), and you can put it in double quotes if you wish (e.g “EndOfQuote”). The next line starts the text being quoted by the heredoc, which continues until it reaches a line containing only the identifier. To ensure the quoted text is displayed in the output area exactly as you’ve laid it out, turn on plain-text mode by adding this command at the top of your code file:

header('Content-Type: text/plain;');

Alternately, if you have control of your server settings, you could set default\_mime type to plain in the php.ini file:

default\_mimetype = "text/plain"

This is not recommended, however, as it puts all output from the server in plain-text mode, which would affect the layout of most of your webcode.

If you do not set plain-text mode for your heredoc, the default is typically HTML mode, which simply displayed the output all on one line.

When using a heredoc for a simple expression, you can put a semicolon after the terminating identifier to end the statement (as shown in the first example). If you are using a heredoc in a more complex expression, however, you’ll need to continue the expression on the next line, as shown here:

printf(<<< Template %s is %d years old. Template, "Fred", 35);

Single and double quotes in a heredoc are preserved:

$dialogue = <<< NoMore

    "It's not going to happen!" she fumed.

    He raised an eyebrow. "Want to bet?"

    NoMore;

    echo $dialogue;

    //"It's not going to happen!" she fumed.

    //He raised an eyebrow. "Want to bet?"

As it whitespace:

$ws = <<< Enough

    boo

    hoo

    Enough;

    //$ws = "boo\n hoo";

New to PHP 7.3 is the indentation of the heredoc terminator. This allows for more legible formatting in the case of embedded code, as in the following function:

<?php

    function sayIt(){

        $ws = <<< "StufftoSay"

        the quick brown fox

        Jumps over the lazy dog.

        StufftoSay;

        return $ws;

    }

    echo sayIt();

    /\*

    The quick brown fox

    Jumps over the lazy dog.

    \*/

    ?>

The newline before the trailing terminator is removed, so these two assignments are identical:

$s = 'Foo';

    // same as

    $s = <<< EndOfPointlessHeredoc

    foo

    EndOfPointlessHeredoc;

If you want a newline to end your heredoc-quoted string, you’ll need to add one yourself:

$s = <<< End

    foo

    End;

**Printing Strings**

There are four ways to send output to the browser. The scho construct lets you print many values at once, while print() prints only one value. The printf() function builds a formatted string by inserting values into a template. The print\_r() function is useful for debugging; it prints the contents of arrays, objects, and other things in a more or less human-readable form.

**Echo**

To put a string into the HTML of a PHP-generated page, use echo. While it looks and for the most part behaves – like a function, echo is a language construct. This means that you can omit the parentheses, so the following expression are equivalent:

echo "Printy";

       echo("Printy"); //also valid

You can specify multiple items to print by separating them with commas:

echo "First", "second", "third";

       //output Firstsecondthird

It is a parse error to use parentheses when trying to echo multiple values:

// this is a pares error

       echo("Hello", "world");

Because echo is not a true function, you can’t use it as part of a larger expression:

// parse error

       if(echo("test")){

        echo("It worked!");

       }

You can easily remedy such errors by using the print() or printf() functions.

**Print()**

The print() function sends one value(its argument) to the browser:

if(print("test\n")){

                print("It worked!");

            }

            //Output

            //test

            //It worked!

**Printf()**

The printf() function outputs a string built by substituting values into a template (the format string). It is derived from the function of the same name in the standard C library. The first argument to printf() is the format string. The remaining arguments are the values to be substituted. A % character in the format string indicates a substitution.

**Format modifiers**

Each substitution marker in the template consists of a percent sign (%), possible followed by modifiers from the following list, and ends with a type specifier. (Use %% to get a single percent character in the output.) The modifiers must appear in the order in which they are listed here:

1. A padding specifier denoting the character to use to pad the results to the appropriate string size. Specify 0, a space, or any character prefixed with a single quote. Passing with spaces is the default.

2. A sign. This has a different effect on strings than on numbers. For strings, a minus (-) here forces the string to be left-justified (the default is right-justified). For numbers, a plus (+) here forces positive numbers to be printed with a leading plus sign (e.g, 35 will be printed as +35).

3. The minimum number of character that this element should contain. If the result would be less than this number of characters, the sign and passing specifier govern how to pad to this length.

4. For floating-point numbers, a precision specifier consisting of a period and a number; this dictates how many decimal digits will be displayed. For types other than double, this specifier is ignored.

**Type Specifiers**

The type specifier tells printf() what type of data is being substituted. This determines the interpretation of the previously listed modifiers. There are eight types, as listed in Table 4-2.

A screenshot of a computer

Description automatically generated

The printf() function looks outrageously complex to people who aren’t C programmers. Once you get used to it, you’ll find it a powerful formatting tool. Here are some examples.

* A floating-point number to two decimal places:

printf('%.2f', 27.452);

            //27.45

* Decimal and hexadecimal output:

printf('The hex value of %d is %x', 214,214);

            // The hex value of 214 is d6

* Padding an integer to three decimal places:

printf('Bond. James Bond. %03d.',7);

            //Bond. James Bond. 007.

* Formatting a date:

printf('%02d/%02d/%04d', $month, $day, $year);

            // 7/25/2023

* A percentage:

printf('%.2f%% Complete',2.1);

            // 2.10% Complete

* Padding a floating-point number:

printf('You\'ve spent $%5.2f so far',4.1);

            //You've spend $ 4.10 so far

The sprintf() function takes the same arguments as printf() but returns the built-up string instead of printing it. This lets you save the string in a variable for later use:

$date = sprintf("%02d/%02d/%04d", $month, $day, $year);

            // now we can interpolate $date wherever we need a date

**print\_r() and var\_dump()**

The print\_r() function intelligently displays what is passed to it, rather than casting everything to a string, as echo and print() do. Strings and numbers are simply printed. Arrays appear as parenthesized lists of keys and values, prefaced by Array:

$a = array('name' => 'Fred', 'age' => 35, 'wife'=> 'Wilma');

        print\_r($a);

        /\*

        Array(

            [name]=> Fred

            [age] => 35

            [wife]=> Wilma)\*/

Using print\_r() on an array moves the internal iterator to the position of the last element in the array. See Chapter 5 for more on iterators and arrays. When you print\_r() an object, you see the word Object, followed by the initialized properties of the object displayed as an array:

class p{

                var $name = 'nat';

                // ...

            }

            $p = new p;

            print\_r($p);

            /\*\*

             \*  Object ([name] => nat)

             \*

             \*/

Boolean values and Null are not meaningfully displayed by print\_r():

  print\_r(true); // prints "1"

             print\_r(true); // prints ""

             print\_r(true); // prints ""

For this reason, var\_dump() is preferred over print\_r() for debugging. The var\_dump() function displays any PHP values in a human-readable format:

var\_dump(true); //1

             var\_dump(false);//2

             var\_dump(null);//3

             var\_dump(array('name' => "Fred", 'age' => 35));//4

             var\_dump($p);//5

             /\*\* bool (true) 1

              \* bool (false) 2

              \* bool (null)  3

              \* array(2){    4

              \* ["name"]=> string(4) "Fred"

              \* ["age]=> int(35)

              \* }

              \*

              \*object(p)(1){ 5

              \* ["name"]=>

              \* string(3) "Nat"

              \*  }\*/

**Accessing Individual Characters**

The strlen() function returns the number of characters in a string:

$string = 'Hello, world';

        $length = strlen($string); //$length is 12

You can use the string offset syntax on a string to address individual characters:

 $string = 'Hello';

        for ($i = 0; $i < strlen($string); $i++) {

        printf("the %dth character is %s\n", $i, $string[$i]);

            }

**Cleaning Strings**

Often, the strings we get from files or uses need to be cleaned up before we can use them. Two common problems with raw data are the presence of extraneous whitespace and incorrect capitalization (uppercase versus lowercase).

**Removing Whitespace**

You can remove leading or trailing whitespace with the trim(), ltrim(), and rtrim() functions:

$trimmed = trim(string [, charlist]);

        $trimmed = ltrim(string [, charlist]);

        $trimmed = rtrim(string [, charlist]);

trim() returns a copy og string with whitespace removed from the beginning and the end. ltrim() (the l is for left) does the same, but removes whitespace only from the start of the string. rtrim() (the r is for right) removes whitespace only from the end of the string. The optional charlist argument is a string that specifies all the characters to strip. The default characters to strip are given in Table 4-3.

A screenshot of a computer

Description automatically generated

For example:

$title = "Programming PHP \n";

        $str1 = ltrim($title); // $str1 is "Programming PHP \n"

        $str2 = rtrim($title); // $str2 is " Programming PHP"

        $str2 = trim($title); // $str2 is "Programming PHP"

Given a line of tab-separated data, use the charlist argument to remove leading or trailing whitespace without deleting the tabs:

$record = " Fred\tFlinstone\t35\tWilma\t \n";

        $record = trim($record, "\r\n\0\x0B");

        // $record is "Fred\tFlintstone\t35\tWilma"

**Changing Case**

PHP has several functions for changing the case of strings: strtolower() and strtoupper() operate on entire strings, ucfirst() operates only on the first character of the string, and ucwords() operates on the first character of each word in the string. Each function takes a string to operate on as an argument and returns a copy of that string, appropriately changed. For example:

$string1 = "FRED flintstone";

        $string2 = "barney rubble";

        print(strtolower($string1));

        print(strtoupper($string1));

        print(ucfirst($string2));

        print(ucwords($string2));

        /\*\*Output

         \* fred flintstone

         \* FRED FLINTSTONE

         \* Barney rubble

         \* Barney Rubble

         \*/

If you’ve got a mixed-case string that you want to convert to “title case”, where the first letter of each word is in uppercase and the rest of the letters are in lowercase (and you’re not sure what case the string is in to begin with), use a combination of strtolower() and ucwords():

print(ucwords(strtolower($string1)));

         //Fred Flintstone

**Encoding and Escaping**

Because PHP programs often interact with HTML pages, web addresses (URLs), and databases, there are functions to help you work with those types of data. HTML, web addresses, and database commands are all strings, but they each require different characters to be escaped in different ways. For instance, a space in a web address must be written as %20, while a literal less-than sign (<) in an HTML document must be written as &lt;. PHP has a number of built-in functions to convert to and from these encodings.

**HTML**

Special characters in HTML are represented by entities such as &amp; (&) and &lt; (<). There are two PHP functions that turn special characters in a string into their entities: one for removing HTML tags, and one for extracting only meta tags.

**Entity-quoting all special characters**

The htmlentities() function changes all characters with HTML entity equivalents into those equivalents (with the exception of the space character). This includes the less-than sign (<), the greater-than sign (>), the ampersand (&), and accented characters. For example:

$string = htmlentities("Einstürzende Neubauten");

        echo $string;

        //Output

        // Einstürzende Neubauten

The entity-escaped version, &uuml; (seen by viewing the source), correctly displays as ü in the rendered web page. As you can see, the space has not been turned into &nbsp;.

The htmlentities() function actually takes up to three arguments:

$output = htmlentities(input,flags,encoding);

The encoding parameter, if given, identifies the character set. The default is “UTF-8.” The flags parameter controls whether single and double quotes, ENT\_QUOTES converts both types of quotes, and ENT\_NOQUOTES converts neither. There is no option to convert only single quotes. For example

 $input = <<< End

        "Stop pulling my hair!" Jane's eyes flashed. <p>

        End;

        $double = htmlentities($input);

        // &quot;Stop pulling my hair!&quot; Jane's eyes flashed.&lt;p&gt;

        $both = htmlentities($input, ENT\_QUOTES);

        // &quot;Stop pulling my hair!&quot; Jane&#039;s eyes flashed.&lt;p&gt;

        $neither = htmlentities($input, ENT\_NOQUOTES);

        //"Stop pulling my hair!" Jane's eyes flashed.&lt;p&gt;

**Entity-quoting only HTML syntax characters**

The htmlspecialchars() function converts the smallest set of entities possible to generate valid HTML. The following entities are converted:

* Ampersands (&) are converted to &amp;
* Double quotes(“) are converted to &quot;
* Single quotes (‘) are converted to &#039; (if ENT\_QUOTES is on, as described for htmlentities())
* Less-than signs (<) are converted to &lt;
* Greater-than signs (>) are converted to &gt;

If you have an application that displays data that a user has entered in a form, you need to run that data through htmlspecialchars() before displaying or saving it. If you don’t, and the user enters a string like “angle < 30” or “sturm & drang”, the browser will think the special characters are HTML, resulting in a garbled page.

Like htmlentities(), htmlspecialchars() can take up to three arguments:

$output = htmlspecialchars(input,[flags, [encoding]]);

The flags and encoding arguments have the same meaning that they do for htmlentities().

There are no functions specifically for converting back from the entities to the original text, because this is rarely needed. There is a relatively simple way to do this, though. Use the get\_html\_translation\_table() function to fetch the translation table used by either of these functions in a given quote style. For example, to get the translation table that htmlentities() uses, do this:

$table = get\_html\_translation\_table(HTML\_ENTITIES);

To get the table for htmlspecialchars() in ENT\_NOQUOTES mode, use:

$table = get\_html\_translation\_table(HTML\_SPECIALCHARS, ENT\_NOQUOTES);

A nice trick is to use this translation table, flip it using array\_flip(), and feed it to strtr() to apply it to a string, thereby effectively doing the reverse of htmlentities():

$str = htmlentities("Einstürzende Neubauten");// now it is encoded

        $table = get\_html\_translation\_table(HTML\_ENTITIES);

        $revTrans = array\_flip($table);

        echo strtr($str, $revTrans);

        //OUTPUT: Einstürzende Neubauten

You can, of course, also fetch the translation table, add whatever other translations you want to it, and then do the strtr(). For example, if you wanted htmlentities() to also encode each space to &nbsp;. You would do:

$table = get\_html\_translation\_table(HTML\_ENTITIES);

        $table[' '] = '&nbsp;';

        $encoded = strtr($original, $table);

**Removing HTML tags**

The strip\_tags() function removes HTML tags from a string:

$input = '<p>Howdy, &quot;Cowboy&quot;</p>';

            $output = strip\_tags($input);

            //$output is 'Howdy, &quot;Cowboy&quot;'

The function may take a second argument that specifies a string of tags to leave in the string. List only the opening forms of the tags. The closing forms of tags listed in the second parameter are also preserved:

$input = 'The <b>bold</b> tags will <i>stay</i><p>';

            $output = strip\_tags($input, '<b>');

            // $output is 'The <b>bold</b> tags will stay'

Attributes in preserved tags are not changed by strip\_tags(). Because attributes such as style and onmouseover can affect the look and behavior of web pages, preserving some tags with strip\_tags() won’t necessarily remove the potential for abuse.

**Extracting meta tags**

The get\_meta\_tags() function returns an array of the meta tags for an HTML page, specified as a local filename or URL. The name of meta tag (keywords, author, description, etc) becomes the key in the array and the content of the meta tag becomes the corresponding value:

$metaTags = get\_meta\_tags('http://www.example.com/');

        echo "Web page made by {$metaTags['author']}";

        //output Web page made by John Doe

The general form of the function is:

$array = get\_meta\_tags(filename [, use\_include\_path]);

Pass a true value for use\_include\_path to let PHP attempt to open the file using the standard include path.

**URLs**

PHP rovides functions to convert to and from URL encoding, which allows you to build and decode URLs. There are actually two types of URL encoding, which differ in how they treat spaces. The first (specified by RFC 3986) treats a space as just another illegal character in a URL and encodes it as %20. The second (implementing the application/x-www-form-urlencoded system) encodes a space as a + and it used in building query strings.

Note that you don’t want to use these functions on a complete URL, such as <http://www.example.com/hello>, as they will escape the colons and slashes to produce

http%3A%2F%2Fwww.example.com%2Fhello

Endode only partial URLs (the bit after <http://www/example.com/>) and add the protocol and domain name later.

**RFC 3986 encoding and decoding**

To encode a string according to the URL conventions, use rawurlencode();

$output = rawurlencode(input);

This function takes a string and returns a copy with illegal URL characters encoded in the %dd conventation.

If you are dynamically generating hypertext references for links in a page, you need to convert them with rawurlencode();

$name = "Programming PHP";

        $output = rawurlencode($name);

        echo "http://localhost{$output}";

        //Output http://localhost/Programming%20PHP

The rawurldecode() function decodes URL-encoded strings:

$encoded = 'Programming%20PHP';

        echo rawurldecode($encoded);

        //Output Programming PHP

**Query-string encoding**

The urlencode() and urldecode() functions differ from their raw counterparts only in that they encode spaces as plus signs(+) instead of as the sequence %20. This is the format for builfing query strings and cookie values. These functions can be useful in supplying form-like URLs in the HTML. PHP automatically decodes query strings and cookie values, so you don’t need to use these functions to process those values. The functions are useful for generating query strings:

$baseUrl = 'http://www.google.com/q=';

        $query = 'PHP sessions -cookies';

        $url = $baseUrl . urlencode($query);

        echo$url;

        //Output: http://www.google.com/q=PHP+sessions+-cookies

**SQL**

Most database systems require that string literals in your SQL queries be escaped. SQL’s encoding scheme is pretty simple-single quotes, double quotes, NUL-bytes, and backslashes need to be preceded by a backslash. The addslashes() function adds these slashes, and the stripslashes() function removes them:

$string = <<< EOF

        "It's never going to work," she cried,

        as she hit the backslash (\) key.

        EOF;

        $string = addslashes($string);

        echo $string;

        echo stripslashes($string);

        /\*\* Output

         \* \"It\'s never going to work,\" she cried,

         \* as she hit the backslash (\\) key.

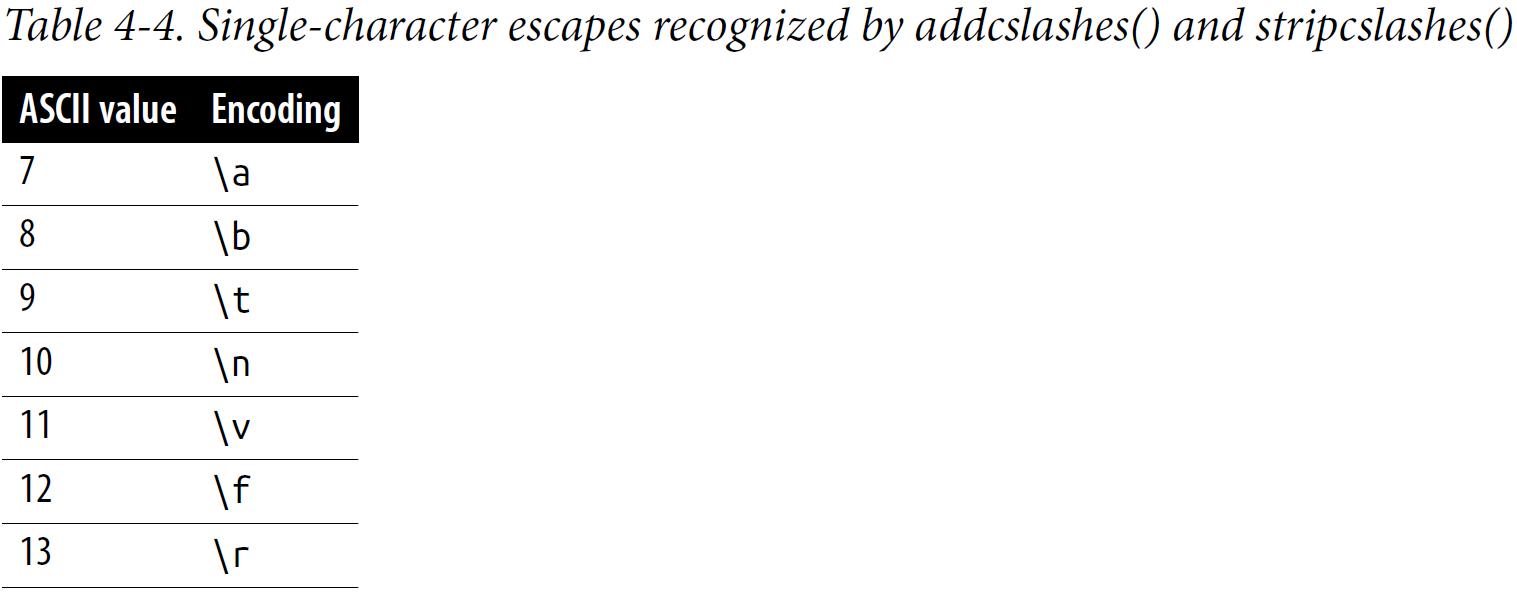
         \* "It's never going to work," she cried, as

         \*  she hit the backslash (\)key.

         \*/

**C-String Encoding**

The addslashes() function escapes arbitrary characters by placing backslashes before them. With the exception of the characters in Table4-4, characters with ASCII values less than 32 or above 126 are encoded with their octal values (e.g., “\002”). The addcslashes() and stripceslashes() functions are used with nonstandard database systems that have their own ideas of which characters need to be escaped.



Call addcslashes() with two arguments – the string to encode and the characters to escape:

$escaped = addcslashes(string,charset);

Specify a range of characters to escape with “..” construct:

echo addcslashes("hello\tworld\n", "\x00..\x1fz..\xff");

        //Output: hello\tworld\n

Beware of specifying ‘0’, ‘a’, ‘b’, ‘f’, ‘n’, ‘r’, ‘t’, or ‘v’ in the character set, as they will be turned into ‘\0’, ‘\a’, and so on. These escapes are recognized by C and PHP and may cause confusion.

Stripcslashes() takes a string and returns a copy with the escapes expanded:

$string = stripcslashes(escaped);

For example:

$string = stripcslashes('hello\tworld\n');

        //Output $string is "hello\tworld\n"

**Comparing Strings**

PHP has two operators and six functions for comparing strings to each other.

**Exact Comparisons**

You can compare two strings for equality with the == and === operators. These operators differ in how they deal with nonstring operands. The == operantor casts string operands to numbers, so it reports that 3 and “3” are equal. Due to the rules for casting strings to numbers, it would also report that 3 and “3b” are equal, as only the portion of the string up to a non-number character is used in casting. The === operator does not cast, and returns false if the data types of the argument differ:

$o1 = 3;

        $o2 = "3";

        if($o1 == $o2){

            echo("== returns true<br>");

        }

        if($o1 === $o2){

            echo("=== returns true<br>");

        }

        // == returns true

The comparison operators (<,<=,>,>=) also work on strings:

$him = "Fred";

        $her = "Wilma";

        if($him < $her){

            print "{$him} comes before {$her} in the alphabet. \n";

        }

        //Output Fred comes before Wilma in the alphabet

However, the comparison operators give unexpected results when comparing strings and numbers:

$string = "PHP Rocks";

        $number = 5;

        if($string < $number){

            echo("{$string} < {$number}");

        }

        //Output PHP Rocks < 5

When one argument to a comparison operator is a number, the other argument is cast to a number. This means that “PHP Rocks” is cast to a number, given 0 (Since the string does not start with a number). Because 0 is less than 5, PHP prints “PHP Rocks < 5”.

To explicitly compare two strings as strings, casting numbers to strings if necessary, use the strcmp() function:

$relationship = strcmp(string\_1, string\_2);

The function returns a number less than 0 if string\_1 sorts before string\_2, greater than 0 if string\_2 sorts before string\_1, or 0 if they are the same:

$n = strcmp("PHP Rocks", 5);

        echo($n);

        //output: 1

A variation on strcmp() is strcasecmp(), which converts strings to lowercase before comparing them. Its arguments and return values are the same as those for strcmp();

$n = strcasecmp("Fred", "frED"); // $n is 0

Another variation on string comparison is to compare only the first few characters of the string. The strncmp() and strncasecmp() functions take an additional argument, the initial number of characters to use for the comparisons:

$relationship = strncmp(string\_1, string\_2, len);

        $relationship = strncasecmp(string\_1, string\_2, len);

The final variation on these functions is natural-order comparison with strnatcmp() and strnatcasesmp(), which take the same arguments as strcmp() and returns the same kind of values. Natural-order comparison identifies numeric portions of the strings being compared and sorts the string parts separately from the numeric parts.

Table 4-5 shows strings in natural order and ASCII order:

A screenshot of a computer

Description automatically generated

**Approximate Equality**

PHP provides several functions that let you test whether two strings are approximately equal—soundex(), metaphone(), similar\_text(), and levenshtein():

$soundexCode = soundex($string);

        $metaphoneCode = metaphone($string);

        $inCommon = similar\_text($string\_1, $string\_2 [, $percentage]);

        $similarity = levenshtenin($string\_1, $string\_2);

        $similarity = levenshtein($string\_1, $string\_2 [, $cost\_ins, $cost\_rep, $cost\_del]);

The Soundex and Metaphone algorithms each yield a string that represents roughly how a word is pronounced in English. To see whether two strings are approximately equal with these algorithms, compare their pronunciations. You can compare Soundex values only to Soundex values and Metaphone values only to Metaphone values. The Metaphone algorithm is generally more accurate, as the following example demonstrates:

$known = "Fred";

        $query = "Phred";

        if(soundex($known) == soundex($query))

        {

            print "soundex: {$known} sounds like {$query} <br>";

        }

        else{

            print "soundex: {$known} doesn't sound like {$query}<br>";

        }

        if(metaphone($known) == metaphone($query))

        {

            print "metaphone: {$known} sounds like {$query}<br>";

        }else {

            print"metaphone: {$known} doesn't sound like {$query}<br>";

        }

        //OutPut

        //soundex: Fred doesn't sound like Phred

        //metaphone: Fred sounds like Phred

The similar\_text() function returns the number of characters that its two string arguments have in common. The third argument, if present, it a variable in which to store the commonality as a percentage:

$string1 = "Rasmus Lerdorf";

        $string2 = "Razmus Lehrdorf";

        $common = similar\_text($string1, $string2, $percent);

        printf("They have %d chars in common (%.2f%%).", $common, $percent);

        //Output: They have 13 chars in common (89.66%)

The Levenshtenin alforithm calculates the similarity of two strings based on how many characters you nust add, substitute, or remove to make them the same. For example instance, “cat” and “cot” have a Levenshtein distance of 1, because you need to change only one character (the “a” to an “o”) to make them the same:

$similarity = levenshtein("cat", "cot");// $similarity is 1

This measure of similarity is generally quicker to calculate than that used by the similar\_text() function. Optionally, you can pass three values to the levenshtein() function to individually weight insertions, deletions, and replacements – for instance, to compare a word against a contraction.

This example excessively weights insertions when comparing a string against its possible contraction, because contractions should never insert characters:

echo levenshtein('would not', 'wouldn\'t', 500,1,1);

**Manipulating and Searching Strings**

PHP has many functions to work with strings. The most commonly used functions for searching and modifying strings are those that use regular expressions to describe the string in question. The functions described in this section do not use regular expressions—they are faster than regular expressions, but they work when you’re looking for a fixed string (for instance if you’re looking for “12/11/01” rather than “any numbers separated by slashes”).

**Substrings**

If you know where the data that you are interested in lies in a larger string, you can copy it out with substr() function:

$piece = substr(string, start[, length]);

The start argument is the position in the string at which to begin copying, with 0 meaning the start ofg the string. The length argument is the number of characters to copy (the default is to copy until the end of the string). For example:

$name = "Fred Flintstone";

        $fluff = substr($name,6,4); //$fluff is "lint"

        $sound = substr($name, 11); //$sound is "tone"

To learn how many time a smaller string occurs within a larger one, use substr\_count();

$number = substr\_count(big\_string, small\_string);

For example:

$sketch = <<< EndOfSketch

        Well, there's egg and bacon; egg sausage and bacon; egg and spam;

        egg bacon and spam; egg bacon sausage and spam; spam bacon sausage

        and spam; spam egg spam spam bacon and spam; spam sausage spam spam

        bacon spam tomato and spam;

        EndOfSketch;

        $count = substr\_count($sketch, "spam");

        print("The wor;d spam occurs {$count} times.");

        //Output The word spam occurs 14 times.

The substr\_replace() function permits many kinds of string modifications:

$string = substr\_replace(original, new, start [, length]);

The function replaces the part of original indicated by the start (0 means the start of the string) and length values with the string new. If no fourth argument is given, substr\_replace() removes the text from start to the end of the string. For instance:

  $greetings "good morning citizen";

        $farewell = substr\_replace($greetings, "bye", 5,7);

        //$farewell is "good bye citizen"

Use a length of 0 to insert without deleting:

$farewell = substr\_replace($farwell, "kind ", 9, 0);

        // $farewell is "good bye kind citizen"

Use a replacement of “” to delete without inserting:

$farewell= substr\_replace($farewell,"", 8);

        // $farewell is "good bye"

Here’s how you can insert at the beginning of the string:

$farewell = substr\_replace($farewell, "now it's time to say ", 0, 0);

        // $farewell is "Now it's time to say good bye"

A negative value for start indicates the number of characters from the end of the string from which to start the replacement:

$farewell = substr\_replace($farewell, "riddance",-3);

        // $farewell is "now it's time to say good riddance"

A negative length indicates the number of characters from the end of the string at which to stop deleting:

$farewell = substr\_replace($farewell, "", -8, -5);

        // $farewell is "now it's time to say good dance"

**Miscellaneous String Functions**

The strrev() function takes a string and returns a revered copy of it:

$string = strrev(string);

For example:

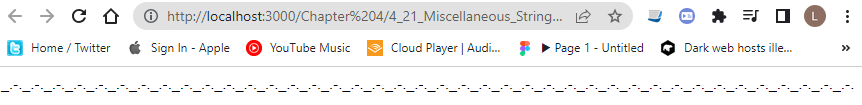
echo strrev("There is no cabal");

        //labac on si erehT

The str\_repear() function takes a string and a count and returns a new string consisting of the argument string repeated count times:

$repeated = str\_repeat(string, count);

For example, to build a crude wavy horizontal rule:



The str\_pad() function pads one string with another. Optionally, you can say what string to pad with, and whether to pad on the left, right, or, both:

$padded = str\_pad(to\_pad, length [, with [, pad\_type]]);

The default is to pad on the right with spaces:

$string = str\_pad('Fred Flintstone', 30);

        echo "{$string}:35:Wilma";

        // Fred Flintstone :35:Wilma

The optional third argument is the string to pad with:

$string = str\_pad('Fred Flintstone', 30, '. ');

        echo "{$string}35";

        //Fred Flintstone. . . . . . . .35

The optional fourth argument can be STR\_PAD\_RIGHT (the default), STR\_PAD\_LEFT, or STR\_PAD\_BOTH (to center). For example:

echo '[' . str\_pad('Fred Flintstone', 30, ' ', STR\_PAD\_LEFT) . "]\n";

echo '[' . str\_pad('Fred Flintstone', 30, ' ', STR\_PAD\_BOTH) . "]\n";

        /\*\*Output

         \* [Fred Flintstone]

         \* [Fred Flintstone]

         \*/

**Decomposing a String**

PHP provides several functions to let you break a string into smaller components. In increasing order of complexity, they are explode(), strtok(), and sscanf().

**Exploding and imploding**

Data often arrives as string, which must be broken down into an array of values. For instance, you might want to split up the comma-separated fields from a string such as “Fred,25,Wilma.” In thses situations, use the explode() function:

$array = explode(separator,string[, limit]);

The first argument, separator, is a string containing the field separator. The second argument, string, is the string to split. The optional third argument, limit, is the maximum number of values to return in the array. If the limit is reached, the last element of the array contains the remainder of the string:

$input = 'Fred,25,Wilma';

        $fields = explode(',', $input);

        // $fields is array('Fred', '25', 'Wilma')

        $fields = explode(',', $input,2);

        // $fields is array('Fred', '25,Wilma')

The implode() function does the exact opposite of explode()—it creates a larger string from an array of smaller strings:

$string = implode(separator, array);

The first argument, separator, is the string to put between the elements of the second argument, array. To reconstruct the simple comma-separated value string, simply say:

$fields = array('Fred','25','Wilma');

$string = implode(',', $fields);// $string is 'Fred,25,Wilma'

The join() function is an alias for implode().

**Tokenizing**

The strtok() function lets you iterate through a stringm getting a new chunk(token) each time. The first time you call it, you need to pass two arguments: the string to iterate over and the token separator. For example:

$firstChunk = strtok(string, separator);

For instance, consider this invocation:

$string = "Fred,Flintstone,35,Wilma";

        $token = strtok($string, ",");

        while($token !== false)

        {

            echo("{$token}<br />");

            $token = strtok(",");

        }

        /\*\* Output

         \* Fred

         \* Flintstone

         \* 35

         \* Wilma

         \*/

The strtok() function returns false when there are no more tokens to be returned. Call strtok() with two arguments to reinitialize the iterator. This restarts the tokenizer from the start of the string.

**sscanf()**

The sscanf() function decomposes a string according to a printf()-like template:

$array = sscanf(string,template);

        $count =sscanf(string, template, var1, ...);

If used without the optional variables, sscanf() returns an array of fields:

$string = "Fred\tFlintstone (35)";

        $a = sscanf($string, "s\t%s (%d)");

        print\_r($a);

        /\*\* Output

         \* Array (

         \* [0] => Fred

         \* [1] => Flintstone

         \* [2] => 35)

         \*/

Pass references to variables to have the fields stored in those variables. The number of fields assigned is returned:

$string = "Fred\tFlintstone (35)";

        $n = sscanf($string, "%s\t%s (%d)", $first, $last, $age);

        echo "Matched {$n} fields: {$first}{$last} is {$age} years old";

        //Output Matched 3 fields: Fred Flintstone is 35 years old

**String-Searching Functions**

Several functions find a string or character within a larger string. They come in three families: strops() and strrpos(), which return a position; strstr(), strchr(), and friends, which return the string they find; and strspn() and strcspn(), which return how much of the start of the string matches a mask.

In all cases, if you specify a number as the “string” to search for, PHP treats that number as the ordinal value of the character to search for. Thus, thses function calls are identical because 44 is ASCII value of the comma:

$pos = strpos($large, ","); // find first comma

$pos = strpos($large, 44);  // also find first comma

All the string-searching functions return false if they can’t find the substring you specified. If the substring occurs at the beginning of the string, the functions return 0. Because false casts to the number 0, always compare the return value with === when testing for failure:

if($pos === false){

                // wasn't found

            }

            else{

                //was found, $pos is offset into string

            }

**Searches Return Position**

The strops() function finds the first occurrence of a small string in a larger string:

$position = strpos(large\_string, small\_string);

If the small string isn’t found, strops() returns false.

The strrpos() function finds the last occurrence of a character in a string. It takes the same arguments and returns the same type of value as strops().

For instance:

$record = "Fred,Flintstone,35,Wilma";

        $pos = strrpos($record, ",");// find last comma

        echo("The last comma in the record is at position {$pos}");

        //The last comma in the record is at position 18

**Searches returning rest of string**

The strstr() function finds the first occurrence of a small string in a larger string and returns from that small string on. For instance:

1 $record = "Fred,Flintstone,35,Wilma";

        $rest = strstr($record, ",");//$rest is ",Flintstone,35,Wilma"

The variations on strstr()are:

* stristr()

Case-insensitive strstr()

* strchr()

Alias for strstr()

* strrchr()

Finds last occurrence of a character in a string

As with strrpos(), strrchr() searches backward in the string, but only for a single character, not for an entire string.

**Searches using masks**

If you thought strrchr() was esoteric, you haven’t seen anything yet. The strspn() and strcspn() functions tell you how many character at the beginning of a string are composed of certain character:

$length = strspn(string, charset);

For example, this function tests whether a string holds an octal number:

 function  isOctal($str){

        return strspn($str,'01234567') == strlen($str);

        }

The c in strcspn() stands for complement—it tells you how much of the start of the string in not composed of the characters in the character set. Use it when the number of interesting characers is greater than the number of uninteresting characters. For example, this function tests whether a string has any NUL-bytes, tabs, or carriage returns:

**Decomposing URLs**

The parse\_url() function returns an array of components of a URL:

$array = parse\_url(url);

For example:

$bits = parse\_url("http://me:secret@example.com/cgi-bin/board?user=fred");

        print\_r($bits);

        /\*\*Array (

         \* [scheme] => http

         \* [host] => example.com

         \* [user] => me

         \* [pass] => secret

         \* [path] => /cgi-bin/board

         \* [query] => user=fred )

         \*/

The possible keys of the hash are scheme, host, port, user, pass, path, query, and fragment.

**Regular Expressions**

If you need more complex searching functionality than the previous methods provide, you can use a regular expression\_\_ a string that represents a pattern, The regular expression functions compare that pattern to another string and see if any of the string matches the pattern. Some function tell you whether there was a match, while other make changes to the string.

There are three uses for regular expressions: matching, which can also be used to extract information from a string; substituting new text for matching text; and splitting a string into an array of smaller chunks. PHP has functions for all. For instance, preg\_match() does a regular expressions match.

Perl has long been condifered the benchmark for powerful regular expressions. PHP uses a C library called pcre to provide almost complete support for Per’s arsebal of regular expressions fearures. Perl regular expressions act on arbitrary binary data, so you can safely match patterns or strings that contain the NUL-byte(\x00).

**The Basics**

Most characters in a regular expression are literal characters, meaning that they match only themselves. For instance, if you search for regular expression “/cow/” in the string “dave as a cowhand”, you get a match because “cow” occurs in that string.

Some characters have special meanings in regular expressions. For instance, a caret (^) at the beginning of a regular expression indicates that it must match the beginning of the string (or, more precisely, anchors the regular expression to the beginning of the string):

preg\_match("/^cow/", "Dave was a cowhand"); // returns false

preg\_match("/^cow/", "cowabunga!");// returns true

Similarly, a dollar sign ($)at the end of a regular expression means that it must match the end of the string(i.e, anchors the regular expression to the end of the string):

preg\_match("/cow$/", "Dave was a cowhand"); // returns false

preg\_match("/cow$/", "Don't have a cow");// returns true

A period(.) in a regular expression matches any single character:

preg\_match("/c.t/", "cat"); // returns true

        preg\_match("/c.t/", "cut"); // returns true

        preg\_match("/c.t/", "c t"); // returns true

        preg\_match("/c.t/", "bar"); // returns false

        preg\_match("/c.t/", "ct");  // returns false

If you want to match one of these special characters (called a metacharacter), you have to escape it with a backslash:

preg\_match("/\$5.00/", "Your bill is $5.00 exactly");// returns true

preg\_match("/$5.00/", "Your bill is $5.00 exactly");// returns false

Regular expressions are case-sensitive by default, so the regular expression “/cow/” doesn’t match the string “COW”. If you want to perform a case-insensitive match, you specify a flag to indicate that (as you’ll see later in this chatper).

So far, we haven’t done anything we couln;t have done with string functions we’ve akready seen, like strstr(). The real power of regular expressions comes from their ability to specify abstract patterns that can match many different character sequences. You can specify three basic types of abstract patterns in a regular expression:

* A set of acceptable characters that can appear in the string (e.g, alphabetic characters, numeric character, specific punctuation characters)
* A set of alternatives for the string (e.g, “com”, “edu”, “net”, or “org”)
* A repeating sequence in the string (e.g, at least one but not more than five numeric characters)

These three kinds of patterns can be combined in countless ways to create regular expressions that match such things as valid phone numbers and URLs.

**Character Classes**

To specify a set of acceptable characters in your pattern, you can wither build a character class yourself or use a predefined one. You can build your own character class by enclosing the acceptable characters in square brackets:

preg\_match("/c[aeiou]t/","I cut my hand");// returns true

        preg\_match("/c[aeiou]t/","This crust cat");//returns true

        preg\_match("/c[aeiou]t/","What cart");//returns false

        preg\_match("/c[aeiou]t/","14ct gold");//returns false

The regular expression engine finds a “c”, then checks that thenext character is one of “a”,”e”,”I”.”o”, or “u”. IF it isn’t a vowel, the match fails and the engine goes back to look for another “c”. IF a vowel is found, the engine checks that the next character is a “t”. If it is, the engine is at the end of the match and returns true. If the next character isn’t a “t”, the engine goes back to look for another “c”.

You can negate a character class with a caret (^) at the start:

  preg\_match("/c[^aeiou]/", "I cut my hand");//return false

        preg\_match("/c[^aeiou]/", "Roboot cthon");//return true

        preg\_match("/c[^aeiou]/", "14ct gold");//return false

In this case, the regular expression engine is looking for a “c” followed by a character that isn’t a vowel, followed by a “t”.

You can define a range of characters with a hyphen (-). This simplifies character classes like “all letters” and “all digits”:

preg\_match("/[0-9]%/", "we are 25% complete");//returns true

preg\_match("/[0123456789]%/", "we are 25% complete");//returns true

preg\_match("/[0-9]%/", "11th");//returns false

preg\_match("/[0-9]%/", "cat");//returns true

preg\_match("/[0-9]%/", "PIT");//returns false

preg\_match("/[0-9]%/", "11!");//returns false

preg\_match("/[0-9]%/", "stop!");//returns true

When you are specifying a character class, some special character lose their meaning, while other take on new meanings. Inparticular, the $ anchor and the period lose their meaning in a character class, while the ^ character is no longer an anchor but negates the character class if it is the first character after the open bracket. FCor instance, [^\]] matches any non closing bracket character, while [$.^] matches any dollar sign, period, or caret.

**Alternatives**

You can use the vertical pipe (|) character to specify alternatives in a regular expression:

preg\_match("/car|dog/", "the cat rubbed my legs");// returns true

preg\_match("/car|dog/", "the dog rubbed my legs");//returns true preg\_match("/car|dog/","the rabbit rubbed my legs");//returns false

The precedence of alternation can be a surprise: “/^car|dog$/” selects from “^cat” and “dog$”, meaning that it matches a line that either starts with “cat” or ends with “dog”. If you want a line that contains just “cat” or “dog”, you need to use the regulare expression “/^(cat|dog)$/”.

You can combine character classes and alternation to, for example, check for string that don’t start with a capital letter:

preg\_match("/^([a-z]|[0-9])/", "the quick brown fox");// returns false

preg\_match("/^([a-z]|[0-9])/", "jumped over");//returns true

preg\_match("/^([a-z]|[0-9])/","10 lazy dogs");//returns true

**Repeating Sequences**

To specify a repeating pattern, you use a quantifier. The quantifier goes after the pattern that’s reeated and says how many times to repeat that pattern. Table 4-6 shows the quantifiers that are supported by PHP’s regular expression.

**A black and white text with black text

Description automatically generated**

To repeat a single character, simply put the quantifier after the character:

preg\_match("/ca+t/". "caaaaaaat");// retuns true

        preg\_match("/ca+t/", "ct");//returns false

        preg\_match("/ca?t/","caaaaaaat");//returns false

        preg\_match("/ca+\*t/","ct");//returns true

When quantifiers and character classes we can actually do something useful, like matching valid US telephone number:

preg\_match("/[0-9]{3}-[0-9]{3}-[0-9]{4}/","303-555-1212");//returns true

preg\_match("/[0-9]{3}-[0-9]{3}-[0-9]{4}/","64-9-555-1234");//returns false

**Subpatterns**

You can use parentheses to group bits of a regular expression together tobe treated as a single unit called a subpattern:

preg\_match("/a (very )+big dog/", "it was a very very big dog"); //returns true

        preg\_match("/^(cat|dog)$/", "cat"); //returns true

        preg\_match("/^(cat|dog)$/", "dog"); //returns true

The parentheses also cause the substring that matches the subpattern to be captured. If you pass an array as the third argument to a match function, the array is populated with any captured substring:

preg\_match("/[0-9]+)/", "you have 42 magic beans", $captured);

//returns true and populates $captured

The zeroth element of the array is set tot the entire string being matched against. The first element is the substring that matched the first subpattern (if there is one), the second element is the substring that matched the second subpattern, and so on.  
**Delimiters**

Pearl-style regular expressions emulate the Perl syntax for patterns, which means that each pattern must be enclosed in a pair of delimiters. Traditionally, the forward slash (/) character is used; for example, /pattern/. However, any nonalphanumeric character other than the backslash character (\) can be used to delimit a Perl-style pattern./ This is useful for matching strings containing slashes, such as filenames. For example the following are equivalent:

 preg\_match("/\/use\/local\//", "/usr/local/bin/perl"); // returns true

 preg\_match("#/usr/local/#", "/usr/local/bin/perl"); // returns true

Parentheses (()), curly braces ({}), square brackets ([]), and angle brackets (<>) can be used as pattern delimiters:

preg\_match("{/usr/local/}", "/usr/local/bin/perl");//returns true

The section “Trailing Options” on page {tbd} discusses the single-character modifiers you can put after the closing delimiter to modify the behavior of the regular expression engine. A very useful one is x, which makes the regular expression before matching. These two patterns are the same, but one is much easier to read:

'/([[:alpha:]]+)\s+\1/'

        '/( # start capture

        [[:alpha:]]+ # a word

        \s+ # whitespace

        \1 # the same word again

        ) # end capture

        /x'

**Match Behavior**

The period(.) matches any character except for a newline (\n). The dollar sign ($) matches at the end of the string or, if the string ends with a newline, just before that newline:

preg\_match("/is (.\*)$/","the key is in my pants", $captured);

        // $captured[1] is 'in mt pants'.

**Character Classes**

As shown in Table 4-7, Perl-compatible regular expressions define a number of named sets of characters that you can use in character classes. The expansions in Table 4-7 are for English. The actial letters varu from, locale to locale.

aEach [: something :] classes can used in place of a character in a character class. For instance, to find any character that’s a digit, an uppercase letter, or an “at” sign (@), use the following regular expression:

 [@[:digit:][:upper:]]

However, you can’t use a character class at the endpoint of a range:

 preg\_match("/[A-[:lower:]]/", "string");// invalid regular expression

Some locales consider certain character sequences as if they were a single character these are called collating sequences. To match one of these multicharacter sequences in a character class, enclose it with [. and .]. For example, if your locale has the collating sequence ch, you can match s, t, or ch with character class:

[st[.ch.]]

The final extension to character classes is the equivalence class, which you specify by enclosing the character within [= and =]. Equivalence classes match characters that have the same collating order, as defined in the current locale. For example, a locale may define a, á, and ä as having the same sorting precedence. TO match any one of them, the equivalence class is [=a=].

**A screen shot of a computer

Description automatically generated**

**Anchors**

An anchor limits a match to a particular location in the string (anchors do not match actual characters in the target string). Table 4=8 lists the anchors supported by regular expressions.

A black and white text box

Description automatically generated

A black and white text

Description automatically generated

A word boundary is defined as the point between a whitespace character and an identifier (alphanumeric or underscore) character:

preg\_match("/[[:<:]]gun[[:>:]]/", "the Burgundy exploded");//returns false

preg\_match("/gun/", "the Burgundy exploded");// returns true

Note that the beginning and end of a string also qualify as word boundaries.

**Quantifiers and Greed**

Regular expression quantifiers are typically greedy. That is, when faced with a quantifier, the engine matches as much as it can while still satisfying the rest of the pattern. For instance:

preg\_match("/(<.\*>)/","do <b>not</b> press the button",$match);

        // $match[1] is '<b>not</b>'

The regular expression matches from the first less-than sign to the last greater-than sign. In effect, the .\* matches everything after the first less-than sign, and the engine backtracks to make it match less and less until finally there’s a greater-than sign to be matched.

This greediness can be a problem. Sometimes you need minimal (nongreedy)

Match-ing this is, quantifiers that match as few times as possible to satisfy the resr of the pattern. Perl provides a parallel set of quantifiers that match minimally. They’re easy to remember because there the same as the greedy quantifiers, but with a question mark(?) Appended. Table 4-9 shows the corresponding greedy and nongreedy quantifiers supported by Perl-style regular expressions.

A screenshot of a computer

Description automatically generated

Here’s how to match a tag using a nongreedy quantifier:

 preg\_match("/(<.\*?>)/","do <b>not</b> press the button",$match);

        // $match[1] is '<b>not</b>'

Another, faster way us to use a character class to match every non-greater-than character up to the next greaser-than sign:

preg\_match("/(<[^>]\*>)/","do <b> not </b> press the button", $match);

        // $match[1] is '<b>'

**Noncapturing Groups**

If you enclose a part of a pattern in parentheses, the text that matches the subpattern is captured and can be accessed later. Sometimes, though, you want to create a subpattern without capturing the matching text. In Perl-compatible regular expressions, you can do this using the (?: subpattern) construct:

preg\_match("/(?:ello)(.\*)/", "jello biafra", $match);

        // $match[1] is " biafra"

**Backreferences**

You can refer to text captured earlier in a pattern with a backreference: \1 refers to the contents of the first subpattern, \2 refers to the second, and so on. If you nest subpatterns, the first begins with the first opening parenthesis, the second begins with the seconds opening parenthesis, and so on.

For instance, this identifiers double words:

preg\_match("/([[:alpha:]]+)\s+\1/","Paris in the the spring", $m);

        // returns true and $m[1] is "the"

The preg\_match() function captures at most 99 subpatterns; subpatterns after the 99th are ignored.

**Trailing Options**

Perl-style regular expressions let you put single-letter options (flags) after the regular expression pattern to modify the interpretation, or behavior, of the match. For instance, to match case-insensitively, simply use the i flag:

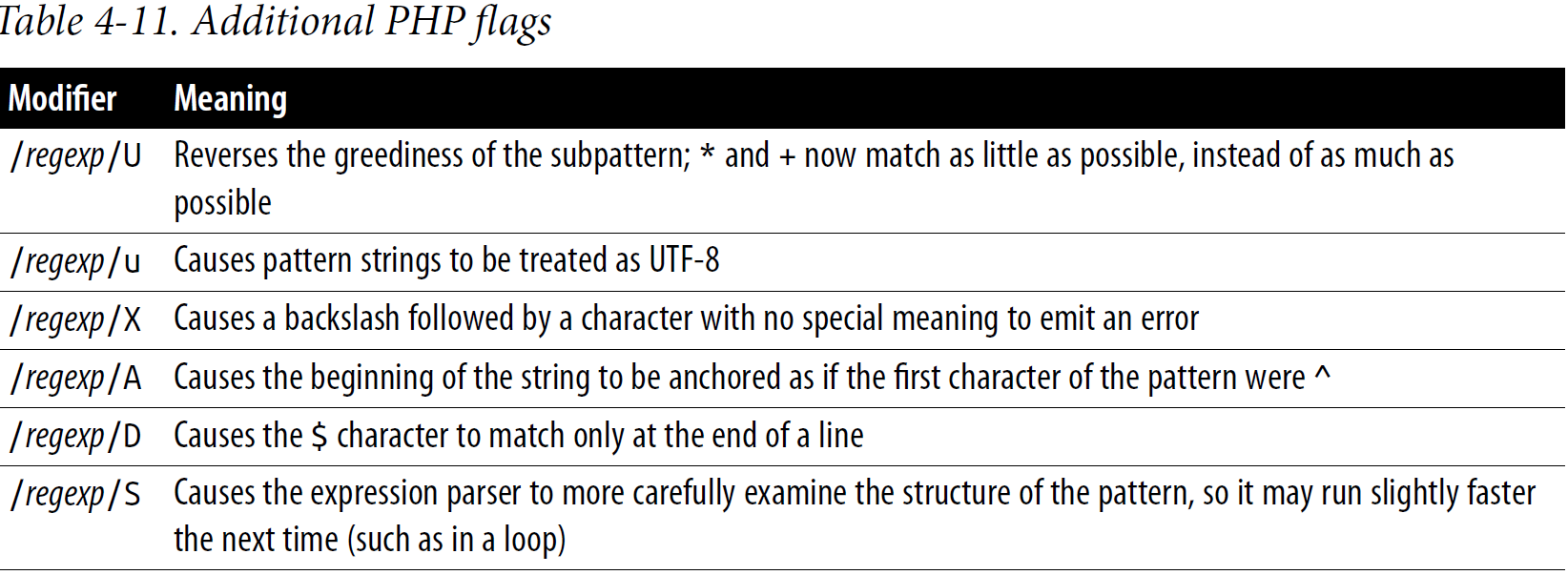
 preg\_match("/cat/i","Stop, Catherine!"); // return true

Table 4-10 shows which Perl modifiers are suppored in Perl-compatible regular expressions.

A screenshot of a computer

Description automatically generated

PHP’s Perl-compatible regular expression functions also support other modifiers the aren’t supported by Perl as listed in Table 4-11.



It’s possible to use more than one option in a single pattern, as demonstrated in the following example:

preg\_match("/cat/i","Stop, Catherine!"); // return true

        $message = <<< END

        To: you@youcorp

        From: me@mecorp

        Subject: pay up

        Pay me or else!

        END;

        preg\_match("/^subject: (.\*)/im",$message, $match);

        print\_r($match);

        //Output Array ([0] => SubjectL pay up [1] => pay up)

**Inline Options**

In addition to specifying pattern-wide options after the closing pattern delimiter, you can specify options within a pattern to have them apply only to part of the pattern. The syntax for this is:

(?flags:subpattern)

For example, only the word “PHP” is case-insensitive in this example:

echo preg\_match(‘/I like (?i:PHP)/’, 'I like pHp', $match);

        print\_r($match);

        //returns true (echo: 1)

        //$match[0] is 'I like pHp'

The I,m,s,U,x and X options can be applied internally in this fashion. You can use multiple oprions at once:

preg\_match('/eat (?ix:foo d)/', 'eat FoOD'); // returns true

Prefix an option with a hyphen (-) to turn it off:

echo preg\_match('/I like (?-i:PHP)/', 'I like pHp', $match);

        print\_r($match);

        //returns false (echo: 0)

        // $match[0] is ''

An alternative form enables or disables the flags until the end of the enclosing subpattern or pattern:

preg\_match('/I like (?i)PHP/', 'I like pHp'); // returns true

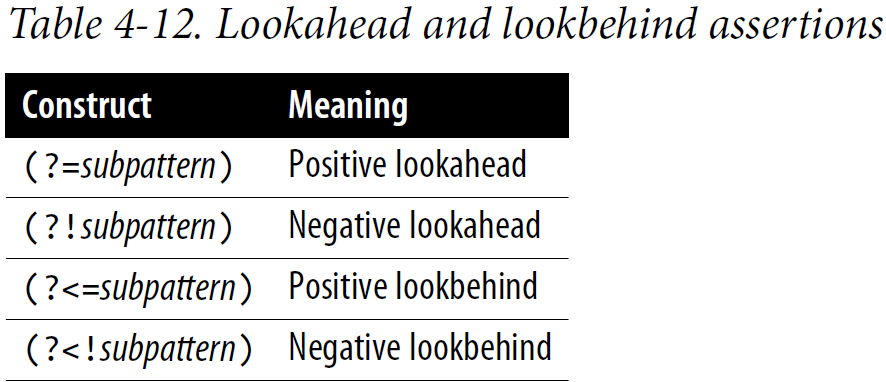
preg\_match('/I (like (?i)PHP) a lot/', 'I like pHp a lot', $match);

        //$match[1] is 'like pHp'

Inline flags do not enable capturing. You need an additional set of capturing parentheses to do that.

**Lookahead and Lookbehind**

In patterns, it’s sometimes useful to say “match here if this is next.” This is particularly common when you are splitting a string. The regular expression describes the separator, which is not returned. You can use lookahead to ensure (without matching it, thus preventing it from being returned) that there’s more data after the separator. Similarly, lookbehind checks the preceding text.



A simple use of negative lookbehind is to extract quoted strings that contain quoted delimiters. For instance, here’s how to extract a single-quoted string (note that the regular expression is commented using the x modifier):

$input = <<< END

        name = 'Tim O\'Reilly';

        END;

        $pattern = <<< END

        ' # opening quote

        ( # begin capturing

         .\*? # the string

         (?<!\\\\) # skip escaped quotes

        ) # end capturing

        ' # closing quote

        END;

        preg\_match("($pattern)x", $input,$match);

        echo $match[1];

        //OUTPUT Tim O\'Reilly

The only tricky part is that to get a pattern that looks behind to see if the last character was a backslash, we need to escape the backslash to prevent the regular expression engine from seeing \), which would mean a literal close parenthesis. In other words, we have to backslash that backslash: \\). But PHP’s string-quoting rules say that \\ produces a literal single backslash, so we end up requiring four backslashes to get one through the regular expression! This is why regular expressions have a reputation for being hard to read.

Perl limits lookbehind to constant-width expression. That is, the expressions cannot contain quantifiers, and if you use alternation, all the choices must be the same length. The Perl-compatible regular expression engine also forbids quantifiers in lookbehind, but does permit alternatives of different length.

**Cut**

The rarely used once-only subpattern, or cut, prevents worst-case behavior by the regular expression engine on some kinds of patterns. The subpattern is never backed out of once matched.

The common use for the once-only subpattern is when you have repeated expression that may itself be repeated:

/(a+|b+)\*\.+/

This code snippet take several seconds to report failure:

$p = '/(a+|b+)\*\.+$/';

        $s = 'abababababbabbbabbaaaaaabbbbabbababababababbba..!';

        if(preg\_match($p, $s)){

            echo "Y";

        }else {

            echo "N";

        }

This is because the regular expression engine tries all the different places to start the match, but has to backtrack out of each one, which takes time. If you know that once something is matched it should never be backed out of, you should mark it with (?>subpattern);

$p = '/(?>a+|b+)\*\.+$/';

The cut never changes the outcome of the match it simply makes it fail faster.

**Conditional Expressions**

A conditional expression is like an if statement in a regular expression. The general form is:

(?(condition)yespattern)

            (?(condition)yespattern|nopattern)

If the assertion can be one of two types: either a backreference, or a lookahead or lookbehind match. To reference a previously matched substring, the assertion is a number from 1 to 99 (the most backreferences available). The condition uses the pattern in the assertion only if the backreference was matched. If the assertion is not a backrederence, it must be a positive or negative lookahead or lookbehind assertion.

**Functions**

There are five classes of functions that work with Perl-compatible regular expressions: matching, replacing, splitting, filtering, and a utility function for quoting text.

**Matching**

The preg\_match() function performs Perl-style pattern matching on a string. It’s the equivalent of the m// operator in Perl. The preg\_match\_all() functions takes the same arguments and gives the same return value as the preg\_match() function, except that it takes a Perl-style pattern instead of a standard pattern:

$found = preg\_match(pattern, string [, captured]);

For example:

preg\_match('/y.\*e$/','Sylvie');// returns true

preg\_match('/y(.\*)e$/', 'Sylvie.$m')// $m is array('ylvie', 'lvi')

While there’s a preg\_match() function to match case-insensitiviely, there;s no preg\_matchi() function. Instead, use the I flag on the pattern:

preg\_match('y.\*e$/i', 'Sylvie',$m);//returns true

The preg\_match\_all() function repeatedly matches from where the last match ended, until no more matches can be made:

$found = preg\_match\_all(pattern, string, matches [, order ])

The order value, either PREG\_PATTERN\_ORDER or PREG\_SET\_ORDER, determines the layout of matches. We’ll look at both, using this code as a guide:

$string = <<< END

        13 dogs

        12 rabbits

        8 cows

        1 goat

        END;

        preg\_match\_all('/(\d+) (/S+)/', $string, $m1, PREG\_PATTERN\_ORDER);

        preg\_match\_all('/(\d+) (/S+)/', $string, $m1, PREG\_SET\_ORDER);

With PREG\_PATTERN\_ORDER (the default), each element of the array corresponds to a particular capturing subpattern. So $m1[0] is an array of all the substrings that matched the pattern, $m1[1] is an array of all the substrings that matched the first subpattern (the numbers), and $m1[2] is an array of all the substrings that matched the second subpattern (the words). The array $m1 has one more elemnt then it has subpatterns.

With PREF\_SET\_ORDER, each element of the array corresponds to the next attempt to match the whole patter. So $m2[0] is an array of the first set of matches (’13 dogs’, ‘13’, ‘dogs’), $m2[1] is an array of the seonds set of matches (’12 rabbits’, ‘12’, ’rabbits’), and so on. The array $m2 has as many elements as there were successful matches of the entire pattern.

Example fetches the HTML at a particular web address into a string and extracts the IR:s from the HTML. For each URL, it generates a link back to the program that will display the IRLs at that address.

    /\*\*Example 4-1 Extracting URLs from an HTML page \*/

    <?php

    $url = '';

    if ($\_SERVER['REQUEST\_METHOD'] === 'POST') {

        // Check if the 'url' key exists in the $\_POST array before accessing it

        if (isset($\_POST['url'])) {

            $url = $\_POST['url'];

        }

    } else {

        // Check if the 'url' key exists in the $\_GET array before accessing it

        if (isset($\_GET['url'])) {

            $url = $\_GET['url'];

        }

    }

?>

    <form action="<?php echo $\_SERVER['PHP\_SELF']; ?> " method="POST">

    <p>URL: <input type="text" name="url" value="<?php echo $url ?>"/><br />

    <input type="submit">

    </form>

    <?php

        if($url){

            $remote = fopen($url, 'r'); {

                $html = fread($remote, 1048576); // read up to 1 MB of HTML

            }

            fclose($remote);

            $urls='(http|https|telnet|gopher|file|wais|ftp)';

            $ltrs ='\w';

            $gunk ='/#~:.?+=&%@!\-';

            $punc='.:?\-';

            $any="{$ltrs}{$gunk}{$punc}";

            preg\_match\_all("{

                \b # start at word boundary

                {$urls}: # need resource and a colon

                [{$any}] +? # followed by one or more of any value

                #characters-but be conservative

                #and take only what you need

                (?= # the match ends at

                [{$punc}]\* # punctuation

                [^{$any}] # followed by a non-URL character

                | # or

                \$ #the end of the string

                )

                }x",$html,$matches);

            printf("I found %d URLs<P>\n", sizeof($matches[0]));

            foreach($matches[0] as $u){

                $link = $\_SERVER['PHP\_SELF']. '?url=' . urlencode($u);

                echo "<a href=\"{$link}\">{$u}</a><br />\n";

            }

        }

    ?>

A screenshot of a computer

Description automatically generated

**Replacing**

The preg\_replace() function behaves like the search-and-replace operation in your text editor. It finds all occurrences of a pattern in a string and changes those occurrences to something else:

$new = preg\_replace(pattern, replacement, subject [, limit ]);

Pass an array of strings as subject to make the substitution on all the them. The new strings are returned from preg\_replace():

names = array('Fred Flintstone',

        'Barney Rubble',

        'Wilma Flintstone',

        'Betty Rubble');

        $tidy = preg\_replace('/(\w)\w\* (/w+)/', '\1 \2', $names);

        // $tidy is array ('F Flintstone', 'B Rubble', 'W Flintstone', 'B Rubble')

To perform multiple substitutions on the same string or array of strings with one call to preg\_replace(), pass arrays of patterns and replacements:

$contractions = array("/don't/i", "/won't/i", "/can't/i");

        $expansions = array('do not', 'will not', 'can not');

        $string = "{;ease don;t yell - I can;t jump while you won't speak";

        $longer = preg\_replace($contractions, $expansions, $string);

        // $longer is 'Please do not yell - I can not j8ump while you will not speak';

If you give fewer replacements than patterns, text matching the extra patterns is deleted. This is a handy way to delete a lot of things at once:

$htmlGunk = array('/<.\*?>/', '/&.\*?;/');

        $html = '&eacute; : <b>very</b> cute';

        $stripped = preg\_replace($htmlGunk, array(), $html);

        // stripped is ' : very cute'

If you give an array of patterns but a single string replacement, the same replacement is used for every pattern:

$stripped = preg\_replace($htmlGunk, '',$html);

The replacement can use backreferences. Unlike backreferences in patterns, though, the preferred syntax for backreferences in replacements is $1,$2,$3, and so on. For example:

echo preg\_replace('/(\w)\w+\s+(\w+)/', '$2, $1', 'Fred Flintstone')

        //output Flintstone, F.

The /e modifier makes preg\_replace() treat the replacement string as PHP code that returns the actual string to use in the replacement. For example, this converts every Celsius temperature to Fahrenheit:

$string = 'It was 5C outside, 20C inside';

        echo preg\_replace('/(\d+)C\b/e', '$1\*9/5+32', $string);

        //OutputIt was 41 outside, 68 inside

This more complex example expands variables in a string:

$name = 'Fred';

        $age = 35;

        $string = '$name is $age';

        preg\_replace('/\$(\w+)/e', '$$1', $string);

Each match isolates the name of a variable ($name, $age). The $1 in the replacement refers to those names, so the PHP code actually executes is $name and $age. That code evaluates to the value of the variable, which is what’s used as the replacement. Whew!

A variation on preg\_replace() is preg\_replace\_callback(). This calls a function to get the replacement string. The function is passed an array of matches (the zeroth element is all the txt that matched the pattern, the first is the contents of the first captured subpattern, and so on). For example:

$string = 'goodbye cruel world';

        $new = preg\_replace\_callback('/\w+/', 'titlecase', $string);

        echo $new;

        //output Goodbye Cruel World

**Splitting**

Whereas you use preg\_match\_all() to extract chunks of a string when you know what those chunks are, use preg\_split() to extract chunks when you know what separates the chunks from each other:

$chunks = preg\_split(pattern,string[, limit [, flags]]);

The pattern matches a separator between two chunck. By default, the separators are not returned. The optional limit specifies the maximum number of chunck to return (-1 is the default, which means all chunks). The flags argument is a bitwise OR combination of the flags PREG\_SPLIT\_NO\_EMPRY (empty chunks are not returned).

For example, to extract just the operands from a simple numeric expression, use:

$ops = preg\_split('{[+\*/-]}', '3+5\*9/2');

        //output: $pos is array ('3','5','9','2')

To extract pattern matches at every boundary between characters in the string, and at the start and end of the string. This lets you split a string into an array of characters:

$array = preg\_split('//', $string);

**Filtering an array with a regular expression**

The preg\_grep() function returns those elements of an array that match a given pattern:

$matching = preg\_grep(pattern, array);

For instance, to get only the filenames that end in .txt, use:

 $textfiles = preg\_grep('/\.txt$/', $filenames);

**Quoting for regular expressions**

The preg\_quote() function creates a regular expression that matches only a given string:

$re =preg\_quote(string [, delimiter]);

Every character in string that has special meaning inside a regular expression (e.g, \* or $) is prefaced with a backslash:

echo preg\_quote('$5.00 (five bucks)');

        //output \$5\.00\(five bucks\)

The oprional second argument is an extra character to be quoted. Usually, pass your regular expression delimiter here:

**Differences from Perl Regular Expressions**

Although very similar, PHP’s implementation of Perl-style regular expressions has a few minor differences from actual Perl regular expressions:

* The NULL character (ASCII 0) is not allowed as a literal character within a pattern string. You can reference it in other ways, however (\000,\x00,etc).
* The \E, \G, \L, \l, \Q, \u, and \U options are not supported.
* The (?{some perl code}) construct is not supported.
* The /D, /G, /U, /u, /A, and /X modifiers are supported
* The vertical tab \v counts as a whitespace character
* Lookahead and lookbehind assertions cannot be repeated using \*, +, or ?.
* Parenthesize submatches within negative assertions are not remembered.
* Alternation branches within a lookbehind assertion can be of different lengths.

**What’s Next**

Now that you know everything there is to know about strings and working with them, the next major part of PHP we’ll focus on is arrays. These compound data types will challenge you, but you need to get well acquainted with them, as PHP works with them in many areas. Learning how to add array elements, sort arrays, and deal with multidimensional forms of arrays is essential to being a good PHP developer.

**Chapter 5**

**Arrays**

AS we discussed in Chapter 2, PHP supports both scalar and compound data types. IN this chaopter, we’ll discuss one of the compound types: arrays. An array is a collection of data calues organized as an ordered collection of key-value pairs. It may help to think of an array, in loose terms, like an egg carton. Each compartment of an egg carton can hold an egg, but it travels around as one overall container. And, just as an egg carton can hold an egg, but it travels around as one overall container. And, just as an egg carton doesn’t have to contain only eggs (you can put anything in there like rocks, snowballs, four-leaf clovers, or nuts and bolts), so too an array is not limited to one type of data. It can hold strings, integers, Booleans, and so on. Plus, array compartments can also contain other arrays\_\_ but more on that later.

This chapter talks about creating an array, adding and removing elements from an array, and looping over the contents of an array. Because arrays are very common and useful, there are many built-in functions that work with them in PHP. For example, if you want to send emails to more than one email; address, you’ll store the email address in an array and then loop through the array, sending the message to the current email address. Also, if you have a form that permits multiple selections, the items the user selected are returned in an array.

**Indexed Versus Associative Arrays**

There are two kinds of arrays in PHP. Indexed and associative. The keys of an indexed array are integer, beginning at 0. Indexed arrays are used when you identify things by their position. Associative arrays have strings as keys and behave more like two-column tables. The first column is the key, which is used to access the value.

PHP internally stortes all arrays as associative arrays; the only dfifferenmce between associative and indexed arrays is what the keys happen to be. Some array features are provided mainly for use with indexed arrays because they assume that you have or want keys that are consecutive integers beginning at 0. In bother cases, the keys are unique. In other words, you can’t have two elements with the same key, regardless of whether the key is a string or an integer.

PHP arrays have an internal order to their elements that is independent of the keys and values, and there are functions that you can use to traverse the arrays based on this internal order. The order is normally that in which values were inserted into the array, but the sorting functions described later in this chapter let you change the order to one based on keys, values, or anything else you choose.

**Identifying Elements of an Array**

Before we look at creating an array, let’s look at the structure of an existing array. You can access specific values from an existing array using the array variable’s name, followed by the element’s key, or index, within square brackets:

$age['fred']

$shows[2]

The key can be either a string or an integer. String values that are equivalent to integer numbers (without leading zeros) are treated as integers. Thus, $array[3] and $array[‘3’] reference the same element, but $array[‘03’] references a different elemet. Negative numbers are valid keys, but they don’t specify positions from the end of the array as they do in Perl.

You don’t have to quote single-word strings. For instance, $age[‘fred’] is the same as $age[fred]. However, It’s consider good PHP style to always use quotes because quoteless keys are indistinguishable from constants. When you use a constant as an unquoted index, PHP uses the value of the constant as the index and emits a warning. This will throw an error in future versionss of PHP:

$person = array("name"=> 'Peter');

        print "Hello, {$perosn[name]}";

        //output: Hello, Peter

        // this 'works' but emits this warning as will

        /\*\* Warning: Use of undefined constant name

         \* - assumed 'name'

         \* (this will throw an Error in a future verson of PHP)

         \*

         \*/

You must use quotes if you’re using interpolation to build the array index:

$person = array("name"=> 'Peter');

        print "Hello, {$perosn[name]}";

        //output: Hello, Peter

        // this 'works' but emits this warning as will

        /\*\* Warning: Use of undefined constant name

         \* - assumed 'name'

         \* (this will throw an Error in a future verson of PHP)

         \*

         \*/

You must use quotes if you’re using interpolation to build the array index:

$person = array("name" => 'Peter');

         print"Hello, {$person["name"]}";

Although it’s technically optional, you should also quote the key if you’re interpolating an array lookup to ensure that you get the value you expect. Consider this example:

define('NAME', 'bob');

         $person = array("name" => 'Peter');

         echo "Hello, {$person['name']}";

         echo "<br/>" ;

         echo NAME ;

         //OUTPUT:

         /\*\*

          \* Hello, Peter

          \* Hello, NAME

          \* bob

          \*/

**Storing Data in Arrays**

Storing a value in an array will create the array if it doesn’t already exist, but trying to retrieve a value from an array that hasn’t been defined won’t create the array. For example:

//$addresses not defined before this point

          echo $addresses[0];// prints nother

          echo $addresses; // prints nothing

          $addresses[0] = "spam@cyberpromo.net";

          echo $addresses; // prints "Array"

Using simple assignment to initialized an array in your program can lead to code like this:

$addresses[0] = "apam@cyberpromo.net";

          $addresses[1] = "abuse@example.com";

          $addresses[2] = "root@example.com";

That’s an indexed array, with integer indices beginning at 0. Here’s associative array:

$price['gasket'] = 15.29;

          $price['wheel'] = 75.25;

          $price['true'] = 50.00;

An easier way to initialize an array is to the the array() construct, which builds an array from its arguments. This builds an indexed array, and the index values (starting at 0) are created automatically:

$addresses = array("spam@cyberpromo.net",

          "abuse@example.com"

          ,"root@example.com");

To create an associative array with array(), use the => symbol to separate indices (keys) from values:

$price = array(

            'gaskey' => 15.29,

            'wheel' => 75.25,

            'tire' => 50.00

          );

Notice the use of whitespace and alignment. We could have bunched up the code, but it wouldn’t Have been as easy. to read (this is equivalent to the previous code sample), or as easy to add or remove values:

$price = array('gasket' => 15.29, 'wheel' => 57.25, 'tire' => 50.00);

You can also specify an array using a shorter, alternate syntax:

$price = ['gasket' => 15.29, 'wheel' => 75.25, 'tire'=> 50.00];

To construct an empty array, pass no arguments to array():

$addresses = array();

You can specify an initial key with => and then a list of values. The values are inserted into the array starting with that key, with subsequent values having sequential keys:

$days = array(1=> "Mon", "Tue", "Wed", "Thu","Fri","Sat","Sun");

       // 2 is tue, 3 is Wed, etc.

If the initial index is a non-numeric string subsequent indices are integers beginning at 0, Thus, the following code is probably a mistake:

$whoops = array('Fri' => "Black", "Brown", "Green");

       // same as

       $whoops = array('Fir' => "Black", 0 => "Brown", 1 => "Green");

**Appending Values to an Array**

To add more values to the end of an existing indexed array, use the [] syntax:

$family = array("Fred", "Wilma");

       $family[] = "Pebbles"; // $family[2] is "Pebbles"

This construct assumes the array’s indices are numbers and assigns elements into the next available numeric index, starting from 0. Attempting to append to an associative array without appropriate keys is almost always a programmer mistake, but PHP will give the new elements numeric indices without issuing a warning:

**Assigning a Range of Values**

The range() function creates an array of consecutive integer or character values between and including the two values you pass to it as arguments. For example:

numbers = range(2,5); // $numbers = arrays(2,3,4,5);

        $letters = range('a','z');

        $reversedNumbers = range(5,2); // $reversedNUmbers = array(5,4,3,2);

Only the first letter of a string argument is used to build the range:

 range("aaa","zzz"); // same as range('a','z')

**Getting the Size of an Array**

The count() and the sizeof() functions are identical in use and effect. They return the number of elements in the array. There is no stylistic preference about which function you use. Here’s an example:

$family = array("Fred","Wilma","Pebbles");

        $size = count($family);// $size is 3

This function counts only array values that are actually set:

$family = array("Fred","Wilma","Pebbles");

        $size = count($family);// $size is 3'

        $confusion = array(10 => "ten", 11 => "eleven", 12 => "twelve");

        $size = count($confusion); // $size is 3

**Padding an Array**

To create an array with values initialized to the same content, use array\_pad(). The first argument to array\_pad() is the array, the second argument is the minimum number of elements you want the array to have, and the third argument is the value to give any elements that are created. The array\_pad() function returns a new padded array, leaving its argument (source) array alone.

Here’s array\_pad() in action:

$scores = array(5,10);

$padded = array\_pad($scores,5,0)// $padded is now array(5,10,0,0,0)

Notice how the new values are appended to the array. If you want the new values added to the start of the array, use a negative second argument:

$padded = array\_pad($scores,-5,0);// $padded is now array(5,10,0,0,0)

If you pad an associative array, existing keys will be preserved. New elements will have numeric keys starting at 0.

**Multidimensional Arrays**

The values in an array can themselves be arrays. This lets you easily create multidimensional arrays:

  $row0 = array(1,2,3);

        $row1 = array(4,5,6);

        $row2 = array(7,8,9);

        $multi = array($row0,$row1,$row2);

You can refer to elements of multidimensional arrays be appending more square brackets, []:

$value = $multi[2][0]; // row 2, colum 0. $value = 7

To interpolate a lookup of a multidimensional array, you must enclose the entire array lookup in curly braces:

**Extracting Multiple Values**

To copy all of an array’s values into variables, use the list() construct:

list($variable, ...) = $array;

The array’s values are copied into the listed variables in the array’s internal order. By default that’s the order in which they were inserted, but the sort functions described later let you change that. Here’s an example:

$person = array("Fred", 35, "Betty");

        list($name, $age,$wife) = $person;

        // $name is "Fred", $age is 35, $wife is "Betty"

The use of the list() function is a common practive for picking up values from adatabase selection where only one row is returned This automatically loads the data from the simple query into a series of local variables. Here is an example of selecting two opposing teams from a sports scheduling database:

$resultArray = [];

        while ($stmt->fetch()) {

            $resultArray[] = ['HomeTeam' => $hometeam, 'AwayTeam' => $awayteam];

        }

        // Now you can use list to extract the values

        foreach ($resultArray as $result) {

            list($homeTeam, $awayTeam) = array\_values($result);

            echo "Home Team: " . $homeTeam . "<br>";

            echo "Away Team: " . $awayTeam . "<br>";

        }

There is more coverage on databases in Chapter 9

If you have more values in the array than in the list(), the extra values are ignored:

$person = array("Fred", 35, "Betty");

list($name, $age) = $person; // $name is "Fred", $age is 35

If you have more values in the list() than in the array the extra values are set to NULL:

 $values = array("hello", "world");

 list($a,$b,$c)= $values; //$a is "hello", $b is "world", $c is NULL

Two or more consecutive commas in the list() skip values in the array:

**Slicing an Array**

To extract only a subset of the array, use the array\_slice() function:

$subset = array\_slice(arrau, offset, length);

The array\_slice() function returns a new array consisting of a consecutive series of valiues from the original array. The offset parameter identifies the initial element to copy (0 represents the first element in the array), and the length parameter identifies the number of values of copy. The new array has consecutive number key starting at 0. For example:

$people = array("Tom","Dick", "Harriet","Brenda","Jo");

$middle = array\_slice($people, 2,2); // $middle is array("Harriet","Brenda")

It is generally only meaningful to use array\_slice() on indexed arrays (i.e., those with consecutive integer indices starting at 0):

// this use of array\_slice() makes no sense

$person = array('name' => "Merry", 'age'=> 35, 'wife' => "Betty");

$subset = array\_slice($person,1,2); // $subset is array(0 => 35, 1 => "Betty")

Combine array\_slice() with list() to extract only some values to variables:

$order = array("Tom", "Dick", "Harriet","Brenda","Jo");

list($seond, $third) = array\_slice($order,1,2);

// $second is "Dick", $third is "Harriet"

**Splitting an Array into Chunks**

To divide an array into smaller, evenly sized arrays, use the array\_chunk() function:

$chunks = array\_chunk(array, size [, preserve\_keys]);

The function returns an array of the smaller arrays. The third argument, preserve\_keys, is a Boolean value that determines whether the elements of the new arrays have the same keys as in the original (useful for associative array) or new numeric key starting from 0 (useful for indexed arrays). The default it to assign new keys, as shown here:

Array(

          [0]=> Array (

          [0] => 1

          [1] => 2

          [2] => 3

          )

          [1] => Array (

          [0] => 4

          [1] => 5

          [2] => 6

          )

          [2] => Array (

          [0] => 7

           )

          )

**Keys and Values**

The array\_keys() function returns an array consisting of only the keys in the array in internal order:

$arrayOfKeys = array\_keys(array);

Here’s an example:

$person = array('name' => "Fred", 'age' => 35, 'wife' => "Wilma");

$keys = array\_keys($person); // $keys is array("name", "age", "wife")

PHP also provides a (generally less useful) function to retrieve an array of just the values in an array, array\_values()

$arrayOfValues = array\_values(array);

AS with array\_keys(), the values are returned in the array’s internal order:

$values = array\_values($person);// $values is array("Fred",35,"Wilma")

**Checking Where an Element Exists**

To see if an element exists in an array, use the array\_key\_exists() function:

if(array\_ley\_exists(key, array)){...}

The function returns a Boolean value that indicates whether the first argument is a valid key in the array given as the second argument.

It’s not sufficient to simply say:

if($person['name']){...}// this can be misleading

Even if there is an element in the array with the key name, its corresponding value might be false (i.e 0, NULL, or the empty string). Instead, use array\_key\_exists(), as follows:

$person['age'] = 0; //unborn?

        if($person['age']){

            echo "true!\n";

        }

        if(array\_key\_exists('age', $person)){

            echo "exists!\n";

        }

        //exists!

Many people use the isset() function, which returns true if the element exists and is not NULL:

$a = array(0,NULL,'');

        function tf($v){

            return $v ? 'T' : 'F';

        }

        for ($i=0; $i<4; $i++){

            printf("%d: %s $s\n", $i, tf(isset($a[$i])), tf(array\_key\_exists($i, $a)));

        }

        /\*\* Output

         \* 0: T T

         \* 1: F T

         \* 2: T T

         \* 3: F F

         \*/

**Removing and Inserting Elements in an Array**

The array\_splice() function can remove or insert elements in an array and optionally create another array from the removed elements:

$removed = array\_splice(array, start [, length [, replacment]]);

We’ll look at array\_splice() using this array:

$subjects = array("physics", "chem", "math", "bio","cs", "drama", "classics");

We can remove the “math”. “bio”, and “cs” elements by telling array\_splice() to start at positton 2 and remove 3 elements:

 $remove = array\_splice($subjects, 2, 3);

        //$removed is array("math", "bio", "cs")

        // $subjects is array("physics", "chem", "drama", "classics")

If you simply want to delete elements from the source array and you don’t care about retaining their values, you don’t need to store the results of array\_splice():

array\_splice($subjects, 2);

        // $subjects is array("physics", "chem");

To insert elements where other were removed, use the fourth argument:

 $new = array("law", "business", "IS");

        array\_splice($subjects, 4, 3, $new);

        //$subjects is array("physics", "chem", "math", "bio", "law", "business", "IS")

The size of the array replacement array doesn’t have to be the same as the number of elements you delete. The array grows or shrinks as needed:

$new = array("law", "business", "IS");

        array\_splice($subjects, 4, 3, $new);

        //$subjects is array("physics", "chem", "math", "bio", "law", "business", "IS")

To insert new elements into the array while pushing existing elements to the right, delete zero elements:

$subjects = array("physics", "chem", 'math');

$new = array("law", "business");

array\_splice($subjects, 2,0,$new);

// $subjects is array("physics", "chem", "law", "business", "math")

Although the examples so far have used an indexed array, array\_splice() also works on associative arrays:

$capitals = array(

            'USA' =>"Washington",

            'Great Britain'=> "London",

            'New Zealand'=> "Wellington",

            'Australia'=> "Canberra",

            'Italy'=> "Rome",

            'Canada'=> "Ottawa",

        );

        $downUnder = array\_splice($capitals, 2,2);// remove New Zealand and Australia

        $france = array('France' => "Paris");

        array\_splice($capitals,1,0,$france); // insert France between USA and GB

**Converting Between Arrays and Variables**

PHP provides two functions, extract() and compact(), that coverts between arrays and variables. The names of the variables correspond to keys in the array, and the values of variables become the values in the array. For instance, this array

$person = array('name' => "Fred", 'age' => 35, 'wife' => "Betty");

Can be converted to, or built from, these variables:

$person = array('name' => "Fred", 'age' => 35, 'wife' => "Betty");

        $name="Fred";

        $age = 35;

        $wife = "Betty";

**Creating Variables from an Array**

The extract() functions automatically creates local variables from an array. The indices of the array elements become the variable names:

extract($person);//$name, $age, and $wife are now set

If a variable created by the extraction has the same name as an existing one, the existing variable’s value is overwritten with the one from the array.’

You can modify extract()’s behavior by passing a second argument. The Appendix describes the possible values for the second argument. The most useful value is EXTR\_PREFIX\_ALL, which indicates that the third argument to extract() is a prefix for the variable names that are created. This helps ensure that you create unique variable names when you are extract(). It is good PHP style to always use EXTR\_PREFIX\_ALL, as shown here:

extract($person);//$name, $age, and $wife are now set

        $shape = "round";

        $array = array('cover' => "bird", 'shape' => "rectangular");

        extract($array, EXTR\_PREFIX\_ALL, "book");

        echo "Cover: {$book\_cover}, Book Shape: {$book\_shape}, Shape: {$shape}";

        //OutPut Cover: bird, Book Shape: rectangular,Shape: round

**Creating an Array from Variables**

The compact() function is the reverse of extract(); you pass it the variable names to compact either as separate parameters or in an array. The compact() function creates an associative array whose keys are the variable names and whose values are the variable’s values. Any names in the array that do not correspond to actual variables are skipped. Here’s an example of compact() in action:

$color = "indigo";

        $shape = "curvy";

        $floppy = "none";

        $a = compact("color", "shape", "floppy");

        //or

        $names = array("color", "shape", "floppy");

        $a = compact($names);

**Traversing Arrays**

The most common task with arrays is to do something with every element—for instance, sending mail to each element of an array address, updating each file in an array of filenames, or adding up each element of an array of prices. There are several ways to traverse arrays in PHP, and the once you choose will depend on data and the task you’re preforming.

**The foreach Construct**

The most common way to loop over elements of an array is to use the foreach construct:

$addresses = array("spam@cyberpromo.net","abuse@exmaple.com");

        foreach($addresses as $value)

        {

            echo "Processing {$value}\n";

        }

        //OutPut Processing spam@cyberpromo.net

        //Processing abuse@example.com

PHP executes the body of the loop (the echo statement) once for each element of $addresses in turn, with $value set to the current element. Elements are processed by their internal order. An alternative form for foreach gives you access to the current key:

$person = array('name' => "Fred", 'age' => 35, 'wife' => "Wilma");

        foreach($person as $key => $value){

            echo "Fred's {$key} is {$value}\n";

        }

        /\*\* Fred's name is Fred

         \* Fred's age is 35

         \* Fred's wife is Wilma

         \*/

In this case, the key for each element is placed in $key and the corresponding value is placed in $value.

The foreach construct does not operate on the array itself, but rather on a copy of it. You can insert or delete elements in the body of a foreach loop, safe in the knowledge that the loop won’t attempt to process the deleted or inserted elements.

**The Iterator Functions**

Every PHP array keeps track of the current element you’re working with; the pointer to the current element is know as the iterator. PHP has functions to set, move, and rest this iterator. The iterator functions are:

* current()

Returns the element currently pointed at by the iterator.

* reset()

Moves the iterator to the first element in the array and returns it.

* next()

Moves the iterator to the next element in the array and returns it.

* prev()

Moves the iterator to the next element in the array and returns it.

* end()

Moves the iterator to the last element in the array and returns it.

* each()

Returns the key and value of the current element as an array and moves the iterator to the next element in the array,

* key()

Returns the key of the current element.

The each() function is used to loop over the elements of an array. It processes elements according to their internal order:

function printAddresses($addresses) {

            foreach ($addresses as $key => $value) {

                echo "{$key} is {$value}<br />\n";

            }

        }

        $addresses = array(

            0 => 'spam@cyberpromo.net',

            1 => 'abuse@example.com'

        );

        printAddresses($addresses);

This approach does not make a copy of the array, as foreach does. This is useful for very large arrays when you want to conserve memory.

The iterator functions are useful when you need to consider some parts of the array separately from other. Example 5-1 shows code that builds a table, treating the first index and value in an associative array as table column headings.

A screen shot of a computer

Description automatically generated

**Using a for Loop**

If you know that you are dealing with an indexed array, where the keys are consecutive integers beginning at 0, you can use a for loop to count though the indices. The for loop operates on the array itself, not on a copy of the array, and processes elements in key order regardless of their internal order.

Here’s how to print an array using for:

**Calling a Function for Each Array Element**

PHP provides a mechanism, array\_walk(), for calling a user-defined function once per element in an array:

array\_walk(array,callable);

The function you define takes in two or, optionally, three arguments: the first is the element’s value, the second is the element’s key, and the third is a value supplied to array\_walk() when it is called. For instance, here’s another way to print table columns made of the values from an array:

//array\_walk(array,callable);

        $printRow = function ($value, $key)

        {

            print("<tr><td>{$key}</td><td>{$value}</td></tr\n");

        };

        $person = array('name' => "Fred", 'age' => 35, 'wife' => "Wilma");

        echo"<table border=1>";

        array\_walk($person, $printRow);

        echo"</table>";

A variation of this example specifies a background color using the oprional third argument to array\_walk(). This parameter gives us the flexibility we need to print many tables, with many background colors:

function printRow($value, $key, $color)

        {

            echo"<tr>\n<td bgcolor=\"{$color}\">{$value}</td>";

            echo "<td bgcolor=\"{$color}\"{$key}</td>\n</tr>\n";

        }

        $person = array('name'=> "Fred", 'age' => 35, 'wife' => "Wilma");

        echo "<table border=\"1\">";

        array\_walk($person, "printRow", "lightblue");

        echo"</table>";

If you have multiple options you want to pass into the called function, simply pass an array in as a third parameter:

$extraData = array('border' => 2, 'color' => "red");

        $baseArray = array("Ford","Chrysler","Volkswagen","Honda","Toyota");

        array\_walk($baseArray, "walkFunction", $extraData);

        function walkFunction($item, $index, $data)

        {

            echo "{$item} <- item, then border: {$data['border']}";

            echo " color->{$data['color']}<br />";

        }

        /\*\* Output

         \* Ford <- item, then border: 2 color->red

         \* Crysler <- item, then border: 2 color->red

         \* VW<- item, then border: 2 color->red

         \* Honda<- item, then border: 2 color ->red

         \* Toyota<- item, then border: 2 color->red

         \*/

The array\_walk() function processes elements in their internal order.

**Reducing an Array**

A cousin of array\_walk(), array\_reduce() applies a function to each element of the array in turn, to build a single value:

$result = array\_reduce(array,callable[, default]);

The function takes two arguments: the running total, and the current value being processed. It should return the new running total. For instance, to add up the squares of the values of an array, use:

//$result = array\_reduce(array,callable[, default]);

        $addItUp = function ($runningTotal, $currentValue)

        {

            $runningTotal +=$currentValue \* $currentValue;

            return $runningTotal;

        };

        $numbers = array(2,3,5,7);

        $total = array\_reduce($numbers, $addItUp);

        echo $total;

        //Output: 87

The array\_reduce() line makes these function calls:

addItUp(0,2);

        addItUp(4,3);

        addItUp(13,5);

        addItUp(38,7);

The default argument, if provided, it a seed value. For instance, if we change the call to array\_reduce() in the previous example:

$total = array\_reduce($numbers, "addItUp", 11);

The resulting function calls are:

addItUp(11,2);

        addItUp(15,3);

        addItUp(24,5);

        addItUp(49,7);

If The array is empty, array\_reduce() returns the default value. If no default value is given and the array is empty, array\_reduce() returns NULL.

**Searching for Values**

The in\_array() function returns true or false, depending on where the first argument is an element in the array given as the second argument:

if(in\_array(to\_find, array[, strict])){...}

If the optional third argument is true, the types of to\_find and the value in the array must match. The default is to not check the data types.

Here’s a simple example:

        $addresses = array("spam@cyberpromo.net", "abuse@example.com",

        "root@example.com");// $gotSpam is true

        $gotSpam = in\_array("spam@cyberpromo.net", $addresses);//$gotMilk is false

PHP automatically indexes the values in arrays, so in\_array() is generally much faster than a loop checking every value in the array to find the one you want.

Example 5-2 checks whether the user has entered information in all the required fields in a form.

<html>

    <head>

        <title> Look Out World</title>

    </head>

    <body>

        <?php

        function hasRequired($array, $requiredFields){

            $array =

            $keys = array\_keys( $array);

            foreach($requiredFields as $fieldName){

                if(! in\_array ($fieldName,$keys)){

                    return false;

                }

            }

            return true;

        }

        if($\_POST['submitted']){

            $testArray = array\_filter($\_POST);

            echo "<p>You ";

            echo hasRequired ($testArray, array (

                'name',

                'email\_address'

            )) ? "did" : "did not";

            echo " have all the required fields.</p>";

        }

        ?>

        <form action="<?php echo $\_SERVER['PHP\_SELF']; ?>" method="POST">

        <p>

            Name: <input type="test" name="name" /><br /> Email address: <input

            type="text" name="email\_address" /><br /> Age (optional): <input

            type="text" name="age"/>

        </p>

        <p style="text-align: center">

        <input type="submit" value="submit" name="submitted" />

    </p>

    </form>

    </body>

</html>

A screenshot of a computer

Description automatically generated

AA variation on in\_array() is the array\_search() function. While in\_array() returns true if the value is found, array\_search() returns the key of the element, of found:

$person = array('name'=> "Fred", 'age' => 35, 'wife' => "Wilma");

        $k = array\_search("Willa", $person);

        //OUTPUT: Fred'a wife is Wilma

The array\_search() function also takes the optional third strict argument, which requires that the types of the value being searched for and the value in the array match.

**Sorting**

Sorting changes the internal order od elements in an array and optionally rewrites the keys to reflect this new order. For example, you might use sorting to arrange a list of scores from biggest to smallest, to alphabetize a list of names, or to order a set of users based on how many messages they posted.

PHP provides three ways to sort arrays-sorting by keys, sorting by value without changes the keys, or sorting by values and then changing the keys. Each kind of sort can be done in ascending order, descending order, or an order determined by a user-defined function.

**Sorting One Array at a Time**

The functions provided by PHP to sort an array are shown in Table 5-1.

A screenshot of a computer

Description automatically generated

The sort(), rsort(), and usort() functions are designed to work on indexed arrays because they assign new numeric keys to represent the ordering. They’re useful when you need to answer questions such as “What are the top 10 scores?” and “Who’s the third person in alphabetical order?” The other sort functions can be used on indexed arrays, but you’ll only be able to access the stored ording by using traversal constructs such as foreach and next().  
To sort names into ascending alphabetical order, do something like like: