



Getting Started with PHP

*Tamar E. Granor
Tomorrow's Solutions, LLC
Voice: 215-635-1958
Website: www.tomorrowssolutionsllc.com
Email: tamar@tomorrowssolutionsllc.com*

PHP is one of the most commonly used languages for adding code to websites. It allows you to pre-process information and generate HTML, and it has extensions for working with a variety of databases, including SQL Server and MySQL.

In this session, we'll look at the basics of PHP as well as how to use it to add data to websites at runtime.

Introduction

I first used PHP when I needed to update the website I'd created for a non-profit I'm involved with to include something like a "Contact Us" page. I needed a way to collect name and address, along with other information. By then, I'd heard a couple of user group sessions on PHP. As I think is common for those creating websites, I found some examples that were similar to what I wanted to do and modified them for my needs.

A few years later, I needed to build some new data-driven websites and used that as an opportunity to actually learn some PHP (as well as JavaScript, Bootstrap, and more). The first site was a struggle, but the second was much easier.

The goal of this session is to help you get through that initial struggle by sharing what I've learned. I'll assume you've spent at least a little time with HTML and have some idea how to create a static webpage.

What is PHP?

PHP is a programming language designed for web development. The name (in a self-referential fashion typical of a certain corner of the computing world) stands for PHP: Hypertext Processor. It was created in 1994.

Most web hosts make PHP available, though it may have to be enabled.

Getting Started

Because PHP code normally runs on a website, you don't actually have to install it locally, but having it installed and properly configured makes it easier to test. In addition, to start with PHP (and web work more generally), you need to have some kind of IDE.

Installing and configuring PHP

The home page for PHP is <https://www.php.net/>. Click Downloads there to get to the download site. There, you'll find the current version at the top and recent older versions below. However, the downloads on that page are Linux-oriented. Click Windows downloads for the version you want, or simply go to <https://windows.php.net/download/>.

The download page offers four different downloads for each PHP version. The first choice is easy: 64-bit (x64) or 32-bit (x86); choose the appropriate one for your computer. The second choice is more complicated: thread safe or non-thread safe. There's information on the download page about what to choose, but if you're not heavily into web development, it's hard to understand. The simple answer seems to be that it depends what web server you're running (on your local machine). If you're using IIS (Microsoft's web server), take the non-thread safe version. If you're using Apache, take the thread safe version.

Once you've downloaded it, installing PHP is simply a matter of unzipping it into a folder.

However, being able to run PHP code on your local machine is complicated. As the download discussion implies, you need to have a web server installed and running. Beyond that, there is configuration required to make that web server talk to PHP.

There are multiple articles about how to set up IIS and PHP to talk to each other. This one is the most straightforward I could find:

<https://jamesmccaffrey.wordpress.com/2017/01/26/installing-php-on-windows-10-and-iis/>.

If you prefer to use Apache, your best bet is to download a complete package that includes Apache, PHP and some other tools from <https://ampps.com/download>.

Choosing an IDE

In the VFP world, we're spoiled by having a dedicated IDE. When you install VFP, you get an IDE with it, including a debugger. Most other programming languages don't have that tight integration of the IDE with the language, so to start working with them, you need to install both the language and an appropriate IDE. The primary benefit of this separation for web development is that you can often use a single IDE for HTML, CSS, PHP, JavaScript, and whatever other web technologies you're working with.

You can create websites with a simple text editor, even Notepad. But you're likely to be more productive with a tool that provides features like syntax coloring and completion, debugging, and so forth.

There are many, many IDEs for web development available. In the last few years, I've tried three of the free options. I haven't been entirely happy with any of them. I'm currently using Microsoft's Visual Studio Code.

The first major IDE I tried was Eclipse (<https://www.eclipse.org/ide/>). It offers multiple versions, including one for PHP. It has a wide variety of supports for writing web code, including syntax coloring, code completion, matching brackets, code folding, and much more. However, I found that fairly regularly, it would lose many of the keyboard shortcuts for navigation. (That is, I'd press something like CTRL-Home and nothing would happen.) I'd have to shut Eclipse down and restart it and most often, that fixed the problem. But it was slow to start up and obviously, having to do that disrupted my train of thought. I searched online for a solution, but nothing I found seemed to fix it for good.

Next, I tried NetBeans IDE (<https://netbeans.org/>). Like Eclipse, it offered syntax coloring, code completion, matching brackets, code folding, and more. However, I found that virtually every time I used it, early in my session, the IDE would freeze for anywhere from 1 to 3 minutes. When the freeze ended, I'd get a message that "slowness" had been "detected" and would be asked to report what was happening. I did that diligently for some time, but finally got tired of it.

I currently use Microsoft's Visual Studio Code (<https://code.visualstudio.com/>). To work with a particular language or languages, you can load extensions that support that

language. I've installed some extensions, but haven't yet hit on exactly the right set to give me all the features Eclipse and NetBeans offer. However, since I also haven't had the kinds of problems I did in those IDEs, I'm far more productive.

Using PHP on a webpage

PHP was created as a server-side scripting language for websites. This has a couple of consequences. First, of course, PHP code is executed on the web server, not on the machine running the browser. That means PHP code can't make assumptions about what's available based on what's available to the local machine.

The second consequence is that there has to be a way to indicate PHP code within a web page. Blocks of PHP code are bracketed with `<?php` and `?>`. That is, a typical PHP block looks like **Listing 1**. (As the example shows, you indicate comment lines in PHP with `//.`)

Listing 1. Blocks of PHP code are indicated by bracketing the code.

```
<?php
//Your PHP code here
?>
```

There are times when you want to simply evaluate an expression in PHP within HTML. To do that, you put the expression inside `<?=>` and `?>`, as in **Listing 2**.

Listing 2. You can embed a PHP expression in HTML to be evaluated as the page is drawn.

```
<td>This line of HTML includes an evaluated PHP expression: <?=> strval($obj->iID) ?>
```

It's not unusual to have a mix of both styles in a single page.

PHP basics

In learning any new programming language, there's a set of basic things you need to do. How do you divide code statements? How do you specify comments? What, if anything, is case-sensitive? And so on. This section of this paper tackles those fundamentals.

Terminating PHP statements

PHP uses semi-colons as statement terminators. That is, you put a semi-colon at the end of each statement, as in **Listing 3**.

Listing 3. PHP requires a semi-colon to terminate each statement.

```
<?php
    echo "hello";
    echo "world";
?>
```

Because every statement is terminated, you can put more than one statement on a line. So the code in **Listing 4** is equivalent to the code in Listing 3.

Listing 4. You can put more than one statement on a line because every statement is terminated.

```
<?php
    echo "hello"; echo "world";
?>
```

The closing bracket (“?”) can serve as a terminator for the last line of code, so the code in **Listing 5** is also equivalent to the other two examples. (In fact, you can omit the closing bracket if the final statement is terminated with a semi-colon.)

Listing 5. The closing bracket for PHP does double-duty as the final statement terminator.

```
<?php
    echo "hello";
    echo "world"
?>
```

Indicating comments

As noted in “Using PHP on a webpage” above, a pair of slashes indicates a comment in PHP. That notation can be used both at the beginning of a line and to add a comment at the end of a line. PHP allows # for the same purpose.

PHP also supports block comments that begin with /* and end with */. These comments can be spread across as many lines as appropriate. The closing */ can appear anywhere, but many people prefer to put it at the beginning of a new line to make the end of the comment obvious.

Listing 6 (HelloWorld.php in the materials for this session) demonstrates the various ways to indicate comments.

Listing 6. There are multiple ways to indicate comments in PHP.

```
<?php
    // This is a comment
    # So is this
    echo "hello"; //This is an inline comment
    echo "world"; # So is this
    /* This is a block comment. It can be divided across
       as many lines as you want. People often use
       these for headers. People often put the closing
       part on a separate line, but it's not required.
    */
?>
```

Most of the IDEs provide an easy way to toggle comments, so you can quickly comment code out or restore it.

Case-sensitivity

PHP is partly case-sensitive. The language keywords can appear in any case, but variable names are case-sensitive. That means that you can use the ECHO command by typing ECHO or echo or Echo or any other mix. However, the variable names \$NAME, \$Name, and \$name refer to three different variables.

Variables

All variable names in PHP begin with \$. The character following the \$ can be a letter or the underscore character. After that, you can use letters, numbers and underscores. (In other words, except for the \$ at the front, PHP variable names follow the same rules as Visual FoxPro.)

Variables in PHP are loosely typed. That is, you don't declare the type for a variable and the type of a variable can be changed dynamically, as in **Listing 7**.

Listing 7. PHP variables are loosely typed, so you can change the type of a variable simply by assigning a value of a different type.

```
<?php
  $MyVar = 'Tamar';
  $MyVar = 7;
?>
```

PHP variables have local scope, by default. However, any code that's not in a defined function is part of the same scope, even if it's interspersed with HTML in a file and even if it's spread across multiple files. Variables defined at that level are referred to as global, even though they are not available inside functions.

However, you can make such global variables available inside a particular function by adding a global definition for it, inside the function. **Listing 8** demonstrates. (See the section "Creating custom functions," later in this paper, for details on defining functions.)

Listing 8. Variables declared at the global scope are not available inside functions, unless you add a global declaration inside the function.

```
<?php
  $MyVar = 'Tamar';

  function test1() {
    global $MyVar;
    echo $MyVar; //Works
  }

  function test2() {
    echo $MyVar; //Gives an error
  }
```

PHP also offers static variables. These allow you to create a variable inside a function and have it remain in memory for future calls to the same function. Static variables let you do things like count how many times a function is called.

Generating output

As you may have inferred by now, the ECHO command generates output. In a web page, it lets you produce text to be inserted into the page. So, for example, if you saved the code in **Listing 3** to a file, uploaded the file to your webhost, and navigated to it, you'd get a page with just the string "helloworld", as shown in **Figure 1**.

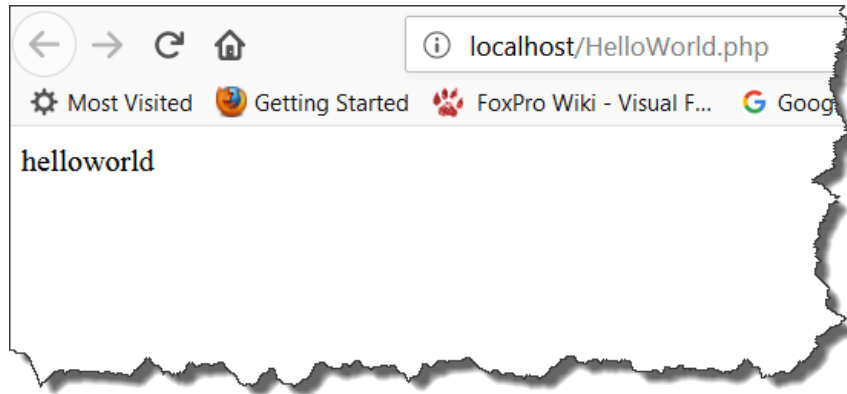


Figure 1. The ECHO command generates output for your web page.

ECHO can actually handle multiple, comma-separated strings, even of different types. (See the section, "Data types," later in this document, for a discussion of PHP data types.) The values passed are concatenated with no space added. (PHP, of course, also has a concatenation operator for strings; it's discussed later in this paper in "Strings.") So the code in **Listing 9** (NameAndBirthdate.php in the materials for this session) produces a single line shown in **Figure 2**; note the inclusion of spaces within the quoted strings in order to get appropriate spacing in the result.

Listing 9. The ECHO command accepts multiple, comma-separated expressions.

```
<?php
    $Name = 'Tamar';
    $BirthYear = 1958;
    echo "My name is ",$Name, ". I was born in ", $BirthYear, "."
?>
```

My name is Tamar. I was born in 1958.

Figure 2. You can build complex strings by combining expressions with ECHO.

ECHO doesn't assume that what you send is meant to be a complete line. Unless you specifically tell it to start a new line, ECHO keeps adding to the current line. So the code in **Listing 10** produces the same result as the code in **Listing 9**.

Listing 10. ECHO doesn't add line breaks unless you tell it to, so this code is equivalent to the previous example.

```
<?php
    $Name = 'Tamar';
    $BirthYear = 1958;
    echo "My name is ";
    echo $Name;
    echo ". I was born in ";
    echo $BirthYear;
    echo ".";
?>
```

Indicating a new line is both simple and tricky. You can include “\n” in the string to start a new line. However, a browser will ignore that character and keep the string on a single line. So, for example, the code in **Listing 11** creates a two-line string, which I can see in VS Code's output window, as in **Figure 3**. However, in the browser, the output looks the same as **Figure 2**.

Listing 11. Including “\n” or “\r\n” in the string to output produces a new line, but it doesn't affect the browser.

```
<?php
    $Name = 'Tamar';
    $BirthYear = 1958;
    echo "My name is ",$Name, ".\nI was born in ", $BirthYear, "."
?>
```

```
My name is Tamar.
I was born in 1958.
```

Figure 3. Because the output included “\n”, in the VS Code output window, two lines are visible.

There are a couple of ways to get a new line in the browser. One is to include the appropriate HTML tag in the string rather than “\n”. The code in **Listing 12** uses the
 tag to break the text into two lines, giving a result like that in **Figure 3**. (However, in this case, the VS Code Output window gets it wrong.)

Listing 12. You can include HTML tags in the expressions you pass to ECHO.

```
<?php
    $Name = 'Tamar';
    $BirthYear = 1958;
    echo "My name is ",$Name, "<br>I was born in ", $BirthYear, "."
?>
```

The second option is to use PHP's nl2br() function to convert the new line character or characters into the
 tag, as in **Listing 13** (EchoWithBreak.php in the downloads for this session).

Listing 13. PHP's `nl2br()` function converts `\n`, `\r` or the combination `\r\n` to the HTML `
` tag.

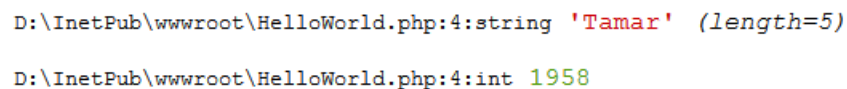
```
<?php
  $Name = 'Tamar';
  $BirthYear = 1958;
  echo "My name is ", $Name, nl2br(".\nI was born in "), $BirthYear, "."
?>
```

One final note related to generating output with ECHO. Single quotes and double quotes are not interchangeable in PHP. That difference, described in “Strings,” later in this paper, affects the results of ECHO.

PHP supports some other ways of generating output, generally used for testing and debugging. `VAR_DUMP()` accepts a list of expressions and produces output showing the type and value for each. For example, if we replace the ECHO command in **Listing 13** with the line shown in **Listing 14** (`VarDump.php` in the materials for this session); the result is shown in **Figure 4**.

Listing 14. The `VAR_DUMP()` function outputs information about each expression you pass it.

```
var_dump($Name, $BirthYear);
```



```
D:\InetPub\wwwroot\HelloWorld.php:4:string 'Tamar' (length=5)
D:\InetPub\wwwroot\HelloWorld.php:4:int 1958
```

Figure 4. `VAR_DUMP()` shows you the type and value of each expression and some additional information.

The `PRINT_R()` function prints (or returns) the value of a variable in a human-readable form. For scalar variables, that doesn't provide anything new, but for arrays (see “Arrays,” later in this paper), it shows the entire array.

Data types

PHP supports four scalar data types: string, integer, float (also known as double), and Boolean. In addition, PHP supports two compound data types: arrays and objects. There are two special data types: resource and NULL.

The PHP documentation refers to callbacks, function calls passed as parameters, and iterables, a generic way to refer to things that can be iterated, as compound data types, but since both are used for specifying parameters and return values, I think they're more notations than data type.

This paper covers the basics about PHP's data types and omits some of their complexities.

Strings

Strings in PHP can be delimited by either single quotes or double quotes, but the two are not interchangeable. Single quotes are the simplest delimiter. When you use them the

string is exactly what's enclosed in the quotes, except that you can insert the single quote character itself using backslash (\) as an escape character. The code in **Listing 15** (SingleQuotes.php in the materials for this session) produces the output shown in **Figure 5**.

Listing 15. Single quotes specify strings that contain exactly what's inside them, except you need to escape single quotes.

```
<?php
    $Name = 'Tamar';
    $Company = 'Tomorrow\'s Solutions, LLC';
    echo $Name,nl2br("\n");
    echo $Company;
?>
```



Figure 5. The simplest way to specify a string is by surrounding it with single quotes.

Double-quotes handle a whole set of escaped characters (**Table 1** shows the most commonly used; see the PHP documentation for the complete set) and evaluate any variables within the string. This is why, in earlier examples, \n was wrapped in double-quotes.

Table 1. When you enclose a string in double-quotes, you can include a number of special characters, including these, and have them handled properly.

Escape sequence	Meaning	ASCII code
\n	Line feed	10
\r	Carriage return	13
\t	Tab	9
\f	Form feed	12
\\	Backslash	92
\\$	Dollar sign	36
\"	Double quote	34

Listing 16 (DoubleQuotes.php in the materials for this session) demonstrates the convenience of evaluating variables on the fly this way. The result is shown in **Figure 6**. This capability makes it easy to put together blocks that include a number of values. You don't have to keep closing quotes, concatenating, and then opening quotes.

Listing 16. When strings are delimited by double-quotes, variables in the string are evaluated.

```
<?php
    $Name = 'Tamar';
    echo "Hello, my name is $Name";
?>
```

```
Hello, my name is Tamar
```

Figure 6. This output was created by evaluating a variable in a double-quoted string.

To concatenate two strings, use the dot operator (that is, the period), as in **Listing 17** (Concatenate.php in the materials for this session). The result is shown in **Figure 7**.

Listing 17. The concatenation operator in PHP is a period.

```
<?php
  $FName = 'Tamar';
  $LName = 'Granor';
  $Name = $FName.' '.$LName;
  echo $Name;
?>
```

```
Tamar Granor
```

Figure 7. You combine strings in PHP with a dot.

Like a number of languages, PHP makes it easy to build up long strings by offering a concatenating assignment operator, combining the dot with an equal sign. When you use this operator, the string on the right-hand side of the operator is concatenated to the string on the left. It's one of a whole set of assignment operators; some of the others are discussed in the next section.

The example in **Listing 17** can be rewritten using this operator, as in **Listing 18** (ConcatenateAssign.php in the materials for this session). While the change probably makes this particular example less readable, when building up long strings, the concatenation assignment operator can make code a lot easier to read.

Listing 18. You can concatenate across multiple assignment statements using `.=`, the concatenating assignment operator.

```
<?php
  $FName = 'Tamar';
  $LName = 'Granor';
  $Name = $FName;
  $Name .= ' ';
  $Name .= $LName;
  echo $Name;
?>
```

Numeric types

PHP supports two numeric data types: integer and float/double. Most of the time, you can use them interchangeably. That is, in most cases, you don't need to think about whether a given value is integer or float. Since PHP is loosely typed, you can assign an integer value to a variable, and follow that up by assigning it a float value. In many cases, conversion

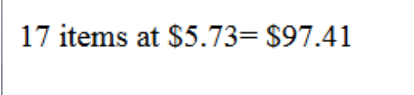
between the two types happens automatically. For example, multiplying an integer by a float results in a float, even if you assign the result to the variable holding the integer.

As in other languages, float values have limited precision and doing arithmetic with float values can lead to rounding errors.

Listing 19 (Numbers.php in the materials for this session) demonstrates both an integer and a float value and the result of multiplying them. **Figure 8** shows the output.

Listing 19. You can use both integer and float values in PHP, and mix them together without explicitly converting.

```
<?php
    $qty = 17;
    $cost = 5.73;
    echo $qty, ' items at $', $cost, '= $', $qty * $cost;
?>
```



17 items at \$5.73= \$97.41

Figure 8. PHP supports integers and floats.

PHP has the arithmetic operators you'd expect: + for addition, - for subtraction, * for multiplication, and / for division. In addition, it offers % for modulo and ** for exponentiation. **Listing 20** shows each of those at work; it's included in the materials for this session as Arithmetic.php.

Listing 20. PHP has a full set of arithmetic operators.

```
<?php
    echo 5+3, '<br>'; // 8
    echo 100-37, '<br>'; // 63
    echo 17*15, '<br>'; // 255
    echo 27/3, '<br>'; // 9
    echo 60 % 8, '<br>'; // 4
    echo 4 ** 3, '<br>'; // 64
?>
```

The + and - operators also convert numeric strings to numbers. The newly created number is either integer or float, based on whether the string has decimals or not. Use + to simply convert the string to a number; use - to invert its sign at the same time.

As mentioned in the previous section, there's a whole set of arithmetic assignment operators. Each of them consists of the arithmetic operator followed by the equal sign. They're shown in **Table 2**. In each case, the variable on the left-hand side is treated as the first operand, as well as being assigned the result, while the variable or value on the right is used as the second operand. Since these operators make the most sense when looping through values, I'll wait to show an example until we see some loop commands.

Table 2. PHP supports a whole set of arithmetic assignment operators that make it easier to accumulate results.

Operator	Operation
+=	Addition
-=	Subtraction
*=	Multiplication
/=	Division
%=	Modulo

PHP also has special operators for incrementing and decrementing a variable. Use ++ to increment by one and -- to decrement by one. **Listing 21** (Incrementing.php in the materials for this session) demonstrates; the result is shown in **Figure 9**.

Listing 21. PHP has special operators for incrementing and decrementing numeric variables.

```
<?php
  $Age = 5;
  echo $Age;
  $Age++;
  echo ' ', $Age++;
?>
```

```
5 6
```

Figure 9. Use the ++ and -- operators to quickly add 1 or subtract 1.

Booleans

As in other languages, Booleans are logical values. They can be either TRUE or FALSE. (The constants are not case-sensitive, so you can write TRUE, true, True, or even tRuE.) Internally, TRUE = 1 and FALSE = 0. When you use ECHO with a Boolean value, the result depends on the value. For true, ECHO prints 1; for false, ECHO prints nothing at all. **Listing 22** (Booleans.php in the materials for this session) demonstrates. The output is shown in **Figure 10**; note that there are only 5 values and all are 1.

Listing 22. The Boolean constants are case-insensitive. Internally, they're represented by 1 (TRUE) and 0 (FALSE)

```
<?php
  $b1 = TRUE;
  $b2 = True;
  $b3 = true;
  $b4 = tRuE;
  $b5 = FALSE;
  $b6 = TRUE;

  echo $b1, $b2, $b3, $b4, $b5, $b6;
?>
```

```
11111
```

Figure 10. When you ECHO Booleans, only the true values are printed (as 1).

PHP offers the usual operators for Boolean values, as well as one that not every language offers. There are two ways to write the And and Or operators, but they have different precedence. `&&` for And and `||` for Or are evaluated before the assignment operators. The actual keywords AND and OR have the lowest precedence. In each of the cases, the And operator has higher precedence than the Or operator.

In addition to those operators, PHP uses `!` for Not and XOR for exclusive OR. The Not operator has very high precedence, while XOR falls between AND and OR.

Comparison operators

PHP has a full set of comparison operators, with a couple of twists. The equal sign is used only for assignment (and in the assignment operators). To compare two values, you use two or three equal signs together. Two equal signs, `==`, is a basic equality comparison; if necessary, data types are coerced to allow the comparison. Three equal signs, `===`, called *identity*, tests for both equality and the same type. **Listing 23** (EqualityTests.php in the materials for this session) demonstrates the difference between them, as well as the coercion of types for the equality test. **Figure 11** shows the results; as noted earlier, ECHO doesn't print anything for False.

Listing 23. The equality operator coerces data types. The identity operators returns True only when types, as well as values, match.

```
<?php
  $Year = 2019;
  $StrYear = '2019';
  echo 'Equality, using ==: ', $Year == $StrYear, nl2br("\n");
  echo 'Identity, using ===: ', $Year === $StrYear, nl2br("\n")
?>
```

```
Equality, using ==: 1
Identity, using ===:
```

Figure 11. PHP offers two operators to test equality. One considers type while the other doesn't.

PHP offers two ways to test for inequality, `!=` and `<>`; they're equivalent. In addition, there are the usual operators for greater than (`>`), greater than or equal (`>=`), less than (`<`) and less than or equal (`<=`).

PHP 7 and later offers one unusual comparison operator, `<=>`, called *spaceship* (probably because of its appearance). It compares two integer values and returns a single value based on the comparison: 1 if the left-hand value is greater, 0 if they're equal, -1 if the right-hand value is greater. **Listing 24** (Spaceship.php in the materials for this session) demonstrates; the result here is -1.

Listing 24. The spaceship operator returns 1, 0 or -1 to indicate how the specified integers compare.

```
<?php
  $Year = 2019;
  $LastYear = 2018;
  echo $LastYear<=>$Year;
?>
```

Operator Precedence

Because PHP has many, many operators (including quite a few not discussed in this paper), operator precedence is complicated. Fortunately, the rule taught in school still applies: parentheses, exponents, multiplication/division, addition/subtraction. But the other operators have to be worked in, as well.

There's another twist as well, called associativity. It determines which direction you work with operators of equal precedence. Because the familiar arithmetic operators all have the same associativity (left), this isn't something we're used to thinking about. That is, $12 / 8 * 4$ is seen as $(12 / 8) * 4$ rather than $12 / (8 * 4)$. But some of PHP's operators have right associativity. One of them is the exponentiation operator, so $2 ** 3 ** 4$ is seen as $2 ** (3 ** 4)$, which is a pretty big number. **Table 3** shows the operators discussed in this paper, in precedence order, and the associativity of each. Use parentheses to change the order of evaluation (or to make your code more readable).

Table 3. To figure out how PHP evaluates an expression, you need to know both the precedence and associativity of the operators it uses.

Operator(s)	Associativity
**	Right
++	Right
--	Right
!	Right
*	Left
/	Left
%	Left
+	Left
-	Left
.	Left
<	Non-associative
<=	Non-associative
>	Non-associative
>=	Non-associative
==	Non-associative
!=	Non-associative
===	Non-associative
!==	Non-associative
<>	Non-associative
<=>	Non-associative
&&	Left
	Left

Operator(s)	Associativity
=	Right
+=	
-=	
*=	
**=	
/=	
.=	
&=	
AND	
XOR	Left
OR	Left

To see the complete set of rules, check the Operator Precedence topic in the PHP manual: <https://www.php.net/manual/en/language.operators.precedence.php>.

Arrays

PHP uses arrays extensively. PHP arrays can contain data of different types and can even contain other arrays. Arrays in PHP are zero-based; that is, the index for the first element is 0, for the second is 1, and so on.

Arrays can be specified as key => value pairs, or as just a list of values. (In fact, within an array, you can mix the two, but it's not a good idea.) When you choose key => value pairs, the key can be numeric or string. An array that uses key => value pairs is called an *associative array*. An array without keys is called an *indexed array*.

You can specify an array using the ARRAY keyword, followed by parentheses that enclose the values, or with no keyword and square brackets. **Listing 25** (DeclareArrays.php in the materials for this session) shows the definition of a couple of arrays; their contents are then displayed using the PRINT_R() function. The output is shown in **Figure 12**.

Listing 25. Arrays in PHP can be created as indexed or associative; associative arrays have a key for each value.

```
<?php
$Organizers = ['Doug', 'Rick', 'Tamar'];

$Person = Array(
    "First" => 'Tamar',
    "Last" => 'Granor',
    "Age" => 60
);

print_r($Organizers);
echo n12br("\n");
print_r($Person);
```



```
?>
```

```
Array ( [0] => Doug [1] => Rick [2] => Tamar )  
Array ( [First] => Tamar [Last] => Granor [Age] => 60 )
```

Figure 12. The `PRINT_R()` function is handy for displaying arrays while testing.

To refer to elements of an indexed array, you use its index (again, counting from 0). For an associative array, you use the element's key. **Listing 26** (`AccessArray.php` in the materials for this session) demonstrates, using the same arrays as in the previous example.

Listing 26. The way you reference an array element depends on whether the array is associative or indexed.

```
<?php  
$Organizers = ['Doug', 'Rick', 'Tamar'];  
  
$Person = Array(  
    "First" => 'Tamar',  
    "Last" => 'Granor',  
    "Age" => 60  
);  
  
echo $Organizers[1], nl2br("\n");  
echo $Person["Last"];  
?>
```

PHP supports multidimensional arrays. In PHP, that term means an array where one or more elements are themselves arrays, as in **Listing 27** (`MultilevelArray.php` in the materials for this session). Note that each contained array can have a different number or even a different set of items. Here, the last contained array has more items than the other two. The result is shown in **Figure 13**; in this case, `VAR_DUMP()` provides more readable output than `PRINT_R()`.

Listing 27. A PHP array can contain other arrays. The contained arrays do not have to have the same or even similar structures.

```
<?php  
$Organizers = array(  
    "Doug" => array(  
        "First" => "Doug",  
        "Last" => "Hennig",  
        "Company" => "Stonefield"  
    ),  
    "Rick" => array(  
        "First" => "Rick",  
        "Last" => "Schummer",  
        "Company" => "White Light"  
    ),  
    "Tamar" => array(  
        "First" => "Tamar",  
        "Last" => "Granor",  
    )  
);
```

```

        "Company" => "Tomorrow's Solutions",
        "Age"     => 60
    )
);

var_dump($Organizers);
?>

```

```

D:\InetPub\wwwroot\HelloWorld.php:21:
array (size=3)
  'Doug' =>
    array (size=3)
      'First' => string 'Doug' (length=4)
      'Last' => string 'Hennig' (length=6)
      'Company' => string 'Stonefield' (length=10)
  'Rick' =>
    array (size=3)
      'First' => string 'Rick' (length=4)
      'Last' => string 'Schummer' (length=8)
      'Company' => string 'White Light' (length=11)
  'Tamar' =>
    array (size=4)
      'First' => string 'Tamar' (length=5)
      'Last' => string 'Granor' (length=6)
      'Company' => string 'Tomorrow's Solutions' (length=20)
      'Age' => int 60

```

Figure 13. VAR_DUMP() provides more readable output for multi-level arrays than PRINT_R().

To reference elements of a multidimensional array, you supply an index or key for each level inside square brackets. So, for example, to find Rick's company or Tamar's age, you'd write the expressions in **Listing 28**.

Listing 28. To reference elements of a multidimensional array, you provide the index or key for each level, each in its own square brackets.

```

$Organizers['Rick']['Company']
$Organizers['Tamar']['Age']

```

You can omit indices or keys from the right in order to refer to an entire contained array. **Listing 29** shows the array of Doug's information being assigned to a new variable.

Listing 29. You can address an entire array within a multidimensional array.

```

$ItsDoug = $Organizers['Doug'];

```

It's possible for some dimensions in a multidimensional array to use indexes while others use keys. For example, the array defined in **Listing 30** (MixedArray.php in the materials for this session) uses an index for the first level, and keys for the second. In this case, that makes things a little confusing, as the result in **Figure 14** shows. Here, the array element with index 0 refers to the data for the number 1, and so on.

Listing 30. Some levels in a multidimensional array can use indexes, while others use keys. (In fact, they can be freely mixed even within a level, but that's likely to lead to confusion.)

```
<?php
$Powers = array(
    array(
        "number" => 1,
        "square" => 1,
        "cube"   => 1
    ),
    array(
        "number" => 2,
        "square" => 4,
        "cube"   => 8
    ),
    array(
        "number" => 3,
        "square" => 9,
        "cube"   => 27
    ),
    array(
        "number" => 4,
        "square" => 16,
        "cube"   => 64
    )
);

var_dump($Powers);
?>
```

```
D:\InetPub\wwwroot\HelloWorld.php:25:
array (size=4)
  0 =>
    array (size=3)
      'number' => int 1
      'square' => int 1
      'cube'   => int 1
  1 =>
    array (size=3)
      'number' => int 2
      'square' => int 4
      'cube'   => int 8
  2 =>
    array (size=3)
      'number' => int 3
      'square' => int 9
      'cube'   => int 27
  3 =>
    array (size=3)
      'number' => int 4
      'square' => int 16
      'cube'   => int 64
```

Figure 14. Different levels in a multidimensional array can make different choices about indexes or keys.

Again, you use the index or key for each dimension to specify a particular element. **Listing 31** shows the way you'd reference the square for 2; since the index starts at 0, the element where number = 2 has the index 1. Obviously, in an application, it's a good idea to define things so that you avoid the cognitive dissonance of this example.

Listing 31. Even when some dimensions use indexes and others use keys, the notation for referencing a particular element is the same.

```
$Powers[1]['square']
```

Objects

PHP is a mixed procedural and object-oriented language. Many built-in capabilities are available in both procedural and object-oriented forms. Creating your own classes in PHP is beyond the scope of this paper, but this section covers enough of PHP's object-oriented capabilities to make it possible to work with classes and objects that are built into the language or available via extensions.

To create a new object from an existing class, use the `NEW` keyword. For example, PHP implements datetimes (and some related concepts) via classes. So, one way to create a datetime variable is to use `NEW`, as in **Listing 32** (`CreateObject.php` in the materials for this session); **Figure 40** shows the result, which indicates the properties of `DateTime`. (However, these properties are protected and cannot be accessed in code.)

Listing 32. In PHP, datetimes are objects, so `NEW` creates a new one.

```
<?php
    $today = new datetime;
    print_r($today);
?>
```

```
DateTime Object ( [date] => 2019-07-03 15:51:07.484639 [timezone_type] => 3 [timezone] => UTC )
```

Figure 15. DateTimes are objects in PHP.

The operator to refer to a property or call a method of a class is `->`. In **Listing 33** (`UseObject.php` in the materials for this session), after creating a `DateTime`, its `format` method is called to generate attractive output. The result is shown in **Figure 40**.

Listing 33. Use `->` to refer to a property or method of an object.

```
<?php
    $today = new datetime;
    echo $today->format('Y-M-j');
?>
```

```
2019-Jul-3
```

Figure 16. The `DateTime` class has a `Format` method to produce attractive output.

The sections “Date and Time creation and conversion functions” and “Date Math functions,” later in this document, cover working with and formatting DateTimes.

Control Structures

PHP has a large set of control structures. Most of them should be familiar to most developers, even if they use a slightly different syntax or format.

All of the control structures that include blocks of code require those blocks to be wrapped in curly braces if they are more than one line. There’s no keyword like ENDIF or ENDDO to indicate the end of a control structure.

Many of the control structures require an expression, as well. Those are wrapped in parentheses.

For example, the usual structure for an IF with an ELSE case is shown in **Listing 34**. For a full discussion of IF, see the next section, “Conditional structures.”

Listing 34. In PHP control structures, conditions to evaluate are enclosed in parentheses, while blocks of code are enclosed in curly braces.

```
if (condition) {
    // commands to execute if condition is true
} else {
    // commands to execute if condition is false
}
```

Conditional structures

PHP has two structures that let you decide which branch of code to execute: IF and SWITCH.

IF has a number of variations. As in other languages, the simplest form takes a condition and executes some code if that condition is true. **Listing 35** shows a simple example (that assumes some earlier code has set the variable \$result).

Listing 35. The simplest IF statement checks a condition and executes some code if it’s true.

```
if (!$result)
    echo "Failure";
```

PHP supports an else case, as well. **Listing 36** expands the previous example to handle both success and failure.

Listing 36. As in other languages, IF has an optional ELSE clause.

```
if ($result)
    echo "Success";
else
    echo "Failure";
```

IF also supports the ELSEIF keyword that lets you lay out multiple cases without having to nest IFs. The code in **Listing 37** (IFwithELSEIF.php in the materials for this session) does what the spaceship operator does as well as providing output.

Listing 37. With ELSEIF, you can test a series of conditions without nesting IFs.

```
if ($Value1 == $Value2) {
    echo "Equal";
    $Result = 0;
}
elseif ($Value1 > $Value2) {
    echo "Larger";
    $Result = 1;
}
else {
    echo "Smaller";
    $Result = -1;
}
```

The SWITCH command lets you act on multiple values of the same expression. The structure is shown in **Listing 38**. There are a few important things to note. First, SWITCH handles only a single expression; you can't evaluate a whole condition in each case. This also means that the expression following the SWITCH keyword is evaluated only once. Second, once you find a matching value, execution continues until the end of the command unless you use BREAK to exit. Third, the DEFAULT case is optional; if you omit it and no cases match, no code is executed.

Listing 38. PHP's SWITCH command is a limited form of CASE statement.

```
switch (expression) {
    case value1:
        //code when expression = value1
        break;
    case value2:
        //code when expression = value2
        Break;
    ...
    default:
        //code when no other case was true
}
```

The code in **Listing 39** checks the variable \$year and determines three things: whether it's a leap year, whether it's a presidential election year and whether it's a congressional election year. (In the US, presidential elections occur in years divisible by 4 and congressional elections in all even years.) However, we can do better than this code. We want the same behavior when the year mod 4 is 1 or 3. With SWITCH, we can combine those cases, as in **Listing 40** (Switch.php in the materials for this session).

Listing 39. SWITCH provides a way to check a series of values for an expression.

```
<?php
```

```
$year = 2018;

switch ($year%4) {
    case 0:
        if ($year%100 <> 0 or $year%400 == 0) {
            $LeapYear = true;
        }
        else {
            $LeapYear = false;
        }

        $Presidential = true;
        $Congressional = true;

        break;
    case 1:
        $LeapYear = false;
        $Presidential = false;
        $Congressional = false;
        break;

    case 2:
        $LeapYear = false;
        $Presidential = false;
        $Congressional = true;
        break;

    case 3:
        $LeapYear = false;
        $Presidential = false;
        $Congressional = false;
}

echo $year, " is ";
if (!$LeapYear)
    echo "not ";
echo "a leap year. It is ";
if (!$Presidential)
    echo "not ";
echo "a presidential election year and is ";
if (!$Congressional)
    echo "not ";
echo "a Congressional election year."

?>
```

Listing 40. If you want the same code for multiple values in a SWITCH, just list all the cases together and then put the code.

```
switch ($year%4) {
    case 0:
        if ($year%100 <> 0 or $year%400 == 0) {
            $LeapYear = true;
        }
        else {
```

```
        $LeapYear = false;
    }

    $Presidential = true;
    $Congressional = true;

    break;

case 1:
case 3:
    $LeapYear = false;
    $Presidential = false;
    $Congressional = false;
    break;

case 2:
    $LeapYear = false;
    $Presidential = false;
    $Congressional = true;
}
}
```

Loops

PHP offers four loop structures: WHILE, DO-WHILE, FOR and FOREACH. Each is useful in different situations.

WHILE evaluates a condition at the top of the loop and continues looping as long as the condition is true. If the condition is initially false, the code inside the loop is never executed.

Listing 41 (While.php in the materials for this session) shows a simple example that figures out when the next Presidential election is starting with a given year. Note that because there's only one line of code inside the loop here, curly braces aren't needed (though they're not prohibited either). **Figure 17** shows the result.

Listing 41. The WHILE command creates a condition-controlled loop with the condition evaluated on the way in.

```
<?php
    $startyear = 2018;
    $year = $startyear;

    while ($year%4 <> 0)
        $year ++;

    echo "The next presidential election year, starting with $startyear, is $year";

?>
```

```
The next presidential election year after 2018 is 2020
```

Figure 17. The code in **Listing 41** produces this output.

DO-WHILE is the same as WHILE except that the condition is evaluated at the end of the loop and thus, the code inside the loop is executed at least once. **Listing 42** (DoWhile.php in the materials for this session) shows the Presidential year calculation using a DO-WHILE loop. The two examples produce the same results, except when \$startyear is already a presidential election year, as in this example. The result is shown in **Figure 18**.

Listing 42. With a DO-WHILE loop, the code inside the loop is executed at least once because the condition isn't tested until the end.

```
<?php
    $startyear = 2016;
    $year = $startyear;

    do
        $year ++;
    while ($year%4 <> 0);

    echo "The next presidential election year after $startyear is $year";

?>
```

The next presidential election year after 2016 is 2020

Figure 18. This is the output from the example in **Listing 42**.

PHP's FOR loop is typically a counted loop, but actually has much more capability than that. You provide a starting expression, an ending expression, and an expression for changing the value of the ending expression, all enclosed in parentheses and separated by semicolons, as shown in **Listing 43**. The starting expression is evaluated once at the beginning of the loop. Then, the ending expression is evaluated. If it's True, the statements inside the loop are executed. Then, the change expression is evaluated, at which point the ending expression is tested again. Execution continues in that manner until the ending expression is False when tested.

Listing 43. A PHP FOR loop is based on three expressions: the first is evaluated only once; the second is tested on the way into the loop and at the beginning of each pass; the third is used to modify conditions, so the loop eventually ends.

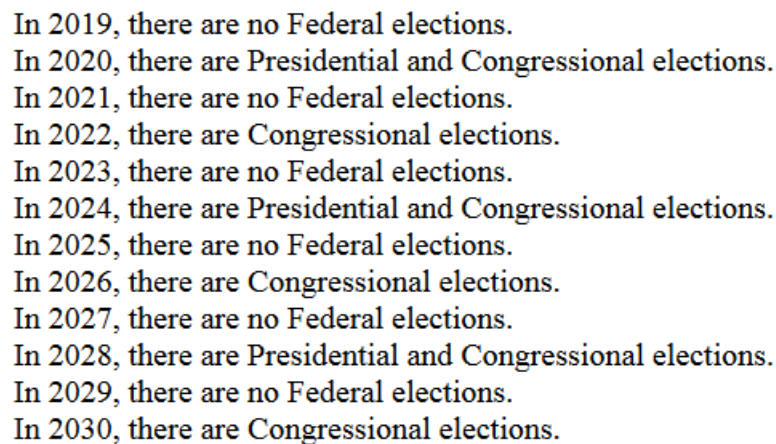
```
for (starting expr;ending expr;change expr) {
    //commands inside loop
}
```

Most often, the change expression increments or decrements a counter created by the starting expression and tested in the ending expression. In **Listing 44** (For.php in the materials for this session), the variable \$year is initialized to 2019 and incremented on each pass through the loop; the loop ends when \$year is more than 2030. The output is shown in **Figure 19**.

Listing 44. The most typical FOR loops initialize a variable, increment or decrement it each time through the loop, and end when a particular value is reached.

```
<?php
for ($year=2019; $year <= 2030; $year++){
    if ($year%4 == 0)
        $status = "Presidential and Congressional elections";
    elseif ($year%2 == 0)
        $status = "Congressional elections";
    else
        $status = "no Federal elections";

    echo nl2br("In $year, there are $status.\n");
}
?>
```



```
In 2019, there are no Federal elections.
In 2020, there are Presidential and Congressional elections.
In 2021, there are no Federal elections.
In 2022, there are Congressional elections.
In 2023, there are no Federal elections.
In 2024, there are Presidential and Congressional elections.
In 2025, there are no Federal elections.
In 2026, there are Congressional elections.
In 2027, there are no Federal elections.
In 2028, there are Presidential and Congressional elections.
In 2029, there are no Federal elections.
In 2030, there are Congressional elections.
```

Figure 19. The code in **Listing 44** uses a loop to figure out the status of a series of years.

It's worth noting that FOR loops can be much more complex. You can actually initialize multiple variables in the starting expression and perform multiple actions in the change expression.

The final control structure is FOREACH, which makes it easy to loop through arrays. The simplest form of the command assigns one value of the array to a variable, which can be used inside the loop; the syntax is shown in **Listing 45**.

Listing 45. FOREACH loops through arrays. In this form, on each pass, the next element of the array specified by `arrayexpression` is assigned to `$value`.

```
foreach (arrayexpression as $value) {
    // commands inside loop
}
```

In **Listing 46** (`ForEach.php` in the materials for this session), the array `$years` contains a list of years. The loop goes through and figures out the status of each listed year in turn. The results are shown in **Figure 20**.

Listing 46. With FOREACH, it's easy to process each element of an array.

```
<?php
$years = array(1958,1963,1977,1982,1986,1995,1999,2008,2017);

foreach ($years as $year){
    if ($year%4 == 0)
        $status = "Presidential and Congressional elections";
    elseif ($year%2 == 0)
        $status = "Congressional elections";
    else
        $status = "no Federal elections";

    echo nl2br("In $year, there are $status.\n");
}
?>
```

```
In 1958, there are Congressional elections.
In 1963, there are no Federal elections.
In 1977, there are no Federal elections.
In 1982, there are Congressional elections.
In 1986, there are Congressional elections.
In 1995, there are no Federal elections.
In 1999, there are no Federal elections.
In 2008, there are Presidential and Congressional elections.
In 2017, there are no Federal elections.
```

Figure 20. The code in **Listing 46** processes a series of values contained in an array.

FOREACH works with multidimensional arrays, too, assigning an entire row to the specified variable. **Listing 47** (ForEachMulti.php in the materials for this session) uses the array of Southwest Fox organizers created in an earlier example to generate some text about who they are; the output is shown in **Figure 21**.

Listing 47. When a FOREACH loop is based on a multidimensional array, it loops through the top dimension, assigning the contained arrays to the variable in turn.

```
<?php
$Organizers = array(
    "Doug" => array(
        "First"    => "Doug",
        "Last"     => "Hennig",
        "Company"  => "Stonefield"
    ),
    "Rick" => array(
        "First"    => "Rick",
        "Last"     => "Schummer",
        "Company"  => "White Light"
    ),
    "Tamar" => array(
        "First"    => "Tamar",
        "Last"     => "Granor",
```

```
        "Company" => "Tomorrow's Solutions",
        "Age"     => 60
    )
);

$OrgNames = '';
foreach ($Organizers as $Org) {
    if ($OrgNames <> '')
        $OrgNames .= ', ';

    $OrgNames .= $Org['First'].' '.$Org['Last'];
};

echo "The organizers of Southwest Fox are: $OrgNames";
?>
```

The organizers of Southwest Fox are: Doug Hennig, Rick Schummer, Tamar Granor

Figure 21. Using a FOREACH loop allows us to collect the list of organizers.

FOREACH can also populate a variable for the key on each pass, along with the variable for the value. **Listing 48** (ForEachKey.php in the materials for this session) shows an example, using the same data as in **Listing 47**; the key is displayed along with the company data for each organizer. **Figure 22** shows the results.

Listing 48. An alternative form of FOREACH provides the key as well as the value of each array item.

```
$Info = '';
foreach ($Organizers as $Key => $Org) {
    $Info .= $Key.': '.$Org['Company'].nl2br("\n");
};

echo $Info;
```

Doug: Stonefield
Rick: White Light
Tamar: Tomorrow's Solutions

Figure 22. You can ask FOREACH to give you the key for each item along with the value.

Functions

PHP has a long list of built-in functions and the ability to define your own functions. In addition, PHP supports extensions that add functions to the language. Later in this paper, we'll look at extensions that support access to databases.

This section looks at the built-in functions you're most likely to need and then shows you how to write your own functions.

Built-in functions

PHP has dozens of built-in functions, but there are a handful that you're likely to use over and over.

String functions

With strings, you're likely to use STRLEN(), SUBSTR(), STRTOUPPER(), STRTOLOWER(), STRPOS() and TRIM() often.

STRLEN() accepts a string and returns its length. As you'd expect, leading and trailing blanks are counted.

The SUBSTR() function lets you extract part of a string, using the syntax shown in **Listing 49**. The start position is zero-based, so to extract the first character in the string, pass 0. If you omit the length parameter, the function returns the rest of the string, beginning with the specified start position.

Listing 49. SUBSTR() returns part of a string, based on the start position and length you pass.

```
$cResult = substr($string, $start, $length)
```

STRTOUPPER() and STRTOLOWER() convert the string to pass to all upper and all lower, respectively. PHP also has functions to capitalize just the first character (UCFIRST()) and the first character of each word (UCWORDS()) in a string.

Use STRPOS() to find one string within another; the syntax is shown in **Listing 50**. The start position and the return value are zero-based. If the second string is not found in the first, the function returns False. (The PHP documentation refers to the first parameter as haystack and the second as needle.) STRPOS() is case-sensitive; for case-insensitive matching, use the otherwise identical STRIPOS() function.

Listing 50. STRPOS() searches for one string within another and returns the position where it's found.

```
$position = strpos($stringtosearchin, $searchtosearchfor, $startposition)
```

The code in **Listing 51** (StrFuncs.php in the materials for this session) demonstrates these five functions.

Listing 51. There are a handful of string functions you're likely to use often.

```
<?php
    $sentence = 'Now is the time for all good men to come to the aid of their
country.';

    $length = strlen($sentence);
    $first5 = substr($sentence, 0, 5);
    $upper = strtoupper($sentence);
    $lower = strtolower($sentence);

    $goodpos = strpos($sentence, 'good');
```

```

echo "Original string is $length characters".nl2br("\n");
echo "First five characters are $first5".nl2br("\n");
echo "Upper case sentence is $upper".nl2br("\n");
echo "Lower case sentence is $lower".nl2br("\n");
echo "Good appears at position $goodpos".nl2br("\n");
?>

```

```

Original string is 69 characters
First five characters are Now i
Upper case sentence is NOW IS THE TIME FOR ALL GOOD MEN TO COME TO THE AID OF THEIR COUNTRY.
Lower case sentence is now is the time for all good men to come to the aid of their country.
Good appears at position 24

```

Figure 23. PHP has lots of functions that operate on strings.

The TRIM() function removes a variety of white space from the beginning and end of a string. The types of white space it removes are shown in **Table 4**.

Table 4. The TRIM() function can return characters other than spaces.

Character	Removes	ASCII value
" "	Spaces	32
"\t"	Tabs	9
"\n"	New line/Linefeed	10
"\r"	Carriage return	13
"\0"	NUL	0
"\x0B"	Vertical tab	11

You can fine tune what the function removes by passing the optional second parameter. Pass a single string listing the characters you want removed. You can actually pass any characters at all, not just the ones in **Table 4**.

Listing 52 (Trim.php in the materials for this session) demonstrates both the basic version of TRIM() and what happens when you pass the second parameter. The examples put a colon before and after the trimmed string to show you where it begins and ends. The output is shown in **Figure 24**.

The first example omits the second parameter, so all kinds of white space are trimmed. In the second example, only spaces are removed; since the original string doesn't have spaces at the beginning or end, nothing is removed. The third example removes the tabs at the beginning of the string, but not the spaces. The last two examples show that the order in which you pass the characters to remove doesn't matter. In all the examples except the first, the CRLF at the end of the string isn't removed, so the second colon appears on a new line.

Listing 52. TRIM() removes lots of kinds of white space by default, but you can specify exactly what to remove.

```

<?php
$messy = "\t\t Something \r\n";

```

```
echo nl2br(":".trim($messy).":\n");
echo nl2br(":".trim($messy," ").":\n");
echo nl2br(":".trim($messy,"\t").":\n");
echo nl2br(":".trim($messy,"\t ").":\n");
echo nl2br(":".trim($messy," \t").":\n");
?>
```

```
:Something:
: Something
:
: Something
:
:Something
:
:Something
:
```

Figure 24. The TRIM() function lets you remove pretty much anything from the beginning and end of a string.

Date and Time creation and conversion functions

There are a number of functions that let you get date and time values and convert them from one type to another. The ones you're likely to use often are listed in **Table 5**. They differ in what type of value they accept as parameters and what type they return.

Table 5. There are a number of ways to get your hands on datetime values.

Function	Purpose
GETDATE()	Returns the current (or a specified) date and time as an array.
DATE()	Returns the current (or a specified) date and time as a string formatted as specified.
DATE_CREATE()	Returns a datetime for the specified date and time string.
DATE_FORMAT()	Returns the specified datetime as a string in the specified format.

GETDATE() fills an array with information about the current date and time, or the datetime you can optionally pass as a parameter. There are 11 elements, ranging from seconds to day of the month to day of the week. All of elements have descriptive keys, except for the last, which contains the number of seconds since the Unix Epoch (that is, midnight January 1, 1970). **Figure 25** shows an example produced by sending the results to VAR_DUMP().

```
D:\InetPub\wwwroot\HelloWorld.php:2:
array (size=11)
  'seconds' => int 23
  'minutes' => int 1
  'hours' => int 14
  'mday' => int 27
  'wday' => int 4
  'mon' => int 6
  'year' => int 2019
  'yday' => int 177
  'weekday' => string 'Thursday' (length=8)
  'month' => string 'June' (length=4)
  0 => int 1561644083
```

Figure 25. A call to GETDATE() produces an array with 11 elements containing many aspects of the current date and time.

DATE() accepts a format string and optionally a datetime and returns either the current date and time or the one you passed in the specified format.

There’s a long list of things you can include in the format string. **Table 6** shows the most common. You can find the full list in the documentation for the DATE() function at <https://www.php.net/manual/en/function.date.php>.

Table 6. PHP gives you many ways to format a datetime. Here are the most common.

Character	Part	Effect	Values or examples
d	Day	Two-digit day of the month (with leading zeroes)	01 to 31
D	Day	Three-character day of the week	Mon through Sun
j	Day	Day of the month without leading zeroes	1 to 31
l (lowercase L)	Day	Full name of day of the week	Monday through Sunday
F	Month	Full name of the month	January through December
m	Month	Two-digit month number (with leading zeroes)	01 to 12
M	Month	Three-character name of the month	Jan through Dec
n	Month	Month number without leading zeroes	1 to 12
Y	Year	Four-digit year	1999, 2019
y	Year	Two-digit year	99, 19
a	Time	Lowercase am or pm	am, pm
A	Time	Uppercase AM or PM	AM, PM
g	Time	12-hour format of hour without leading zeroes	1 through 12
G	Time	24-hour format of hour without leading zeroes	0 through 23
h	Time	Two-digit 12-hour format of hour (with leading zeroes)	01 through 12
H	Time	Two-digit 24-hour format of hour (with leading zeroes)	00 through 23
i	Time	Two-digit minutes (with leading zeroes)	00 through 59
s	Time	Two-digit seconds (with leading zeroes)	00 through 59

Listing 53 shows some examples of typical ways of formatting dates and times. In each case, it’s formatting the date and time the command was run.

Listing 53. PHP provides many ways to format dates and times.

```
<?php
  echo date('Y-M-j').nl2br("\n");
  echo date('n/j/Y g:i:s A').nl2br("\n");
  echo date('l, F j, Y').nl2br("\n");
  echo date('H:i:s').nl2br("\n");
?>
```

```
2019-Jun-27
6/27/2019 3:51:06 PM
Thursday, June 27, 2019
15:51:06
```

Figure 26. Dates and times can be formatted in many ways in PHP.

What you can't tell from the previous example is that the information is returned in UTC format, not based on the current time zone. Use the `DATE_DEFAULT_TIMEZONE_SET()` function before working with datetimes to tell PHP what time zone you're in (or want to use, anyway). In **Listing 54** (`DateFunction.php` in the materials for this session), the default time zone is set to my home zone of "America/New_York"; **Figure 27** shows the results in that case.

Listing 54. To ensure that dates and times reflect a local time zone, you can set the desired time zone before calling `DATE()`.

```
<?php
  date_default_timezone_set('America/New_York');
  echo date('Y-M-j').nl2br("\n");
  echo date('n/j/Y g:i:s A').nl2br("\n");
  echo date('l, F j, Y').nl2br("\n");
  echo date('H:i:s').nl2br("\n");
?>
```

```
2019-Jun-27
6/27/2019 11:54:52 AM
Thursday, June 27, 2019
11:54:52
```

Figure 27. These results reflect my home time zone.

However, that setting lasts only for the code you're running. After running the previous example, I commented out the call to `DATE_DEFAULT_TIMEZONE_SET()` and ran the code again and I was back to UTC time. If you want all your code to run in a time zone other than UTC, you can set the default in the `PHP.INI` file that should be part of your configuration.

Beyond that, some of the datetime functions let you specify the desired time zone or the offset from UTC. (In fact, though it's not shown in Table 6, you can specify time zone or offset from UTC in a date format.)

DATE_CREATE() accepts a date and time as a string and returns a datetime value. If you omit the parameter, it returns the current date and time (in UTC time). The function accepts a wide range of formats for the parameter; they're documented at <https://www.php.net/manual/en/datetime.formats.php>. **Listing 55** (DateCreate.php in the materials for this session) demonstrates a couple of possibilities; note the inclusion of the time zone for the \$keynote variable. The results are shown in **Figure 28**.

Listing 55. The DATE_CREATE() functions turns a string into a datetime.

```
<?php
    $today = date_create();
    $birthday = date_create('1958-09-28');
    $keynote = date_create('24-Oct-2019 07:00:00 PM -08:00');

    print_r($today);
    echo "<br>";
    print_r($birthday);
    echo "<br>";
    print_r($keynote);
?>
```

```
DateTime Object ( [date] => 2019-06-28 19:08:17.759917 [timezone_type] => 3 [timezone] => UTC )
DateTime Object ( [date] => 1958-09-28 00:00:00.000000 [timezone_type] => 3 [timezone] => UTC )
DateTime Object ( [date] => 2019-10-24 19:00:00.000000 [timezone_type] => 1 [timezone] => -08:00 )
```

Figure 28. You can create datetime values using DATE_CREATE().

While PHP is pretty smart about inferring the format from the string you supply, if you're concerned about ambiguity, you can instead use the DATE_CREATE_FROM_FORMAT() function, which accepts both the datetime string and a format string in order to return a datetime value. Be aware that the format string is the first parameter.

DATE_FORMAT() accepts a datetime and a format string and returns a string showing the specified date in the specified format. It's especially useful for taking data that comes from a database and formatting it for display. **Listing 56** (DateFormat.php in the materials for this session) demonstrates; the output is shown in **Figure 29**.

Listing 56. The DATE_FORMAT() function converts a datetime to a string, using a specified format.

```
<?php
    date_default_timezone_set('America/Phoenix');
    $today = date_create();

    echo date_format($today, 'Y-M-j').nl2br("\n");
    echo date_format($today, 'n/j/Y g:i:s A').nl2br("\n");
    echo date_format($today, 'l, F j, Y').nl2br("\n");
    echo date_format($today, 'H:i:s').nl2br("\n");
?>
```

```
2019-Jun-28  
6/28/2019 12:17:07 PM  
Friday, June 28, 2019  
12:17:07
```

Figure 29. DATE_FORMAT() lets you convert from datetime to string, in the format you specify.

Date Math functions

PHP also has functions for date math. DATE_ADD() and DATE_SUB() let you add or subtract a time period to or from a datetime, while DATE_DIFF() lets you find the difference between a pair of datetimes.

All three functions deal with date intervals. A date interval is an object with properties for each component of a datetime (and more).

There is an ISO standard for specifying date intervals using strings; the components for that format are shown in **Table 7**. To specify an interval, start with “P” and then follow it with as many of the others as you need. Precede each with the quantity of those units. So, for example, P6D is six days, P2Y3M is two years and three months, PT3H is three hours, and P5DT12H is five days and 12 hours.

Table 7. Combine the components here you need to specify a date interval.

Component	Meaning
P	Period, required
Y	Years
M	Months
D	Days
T	Time, required if including any time components
H	Hours
M	Minutes
S	Seconds

To turn one of these strings into a date interval object, use the NEW command, as in **Listing 57**.

Listing 57. You can create a date interval using an ISO standard format.

```
$twomonths = new DateInterval("P2M");
```

PHP also lets you create date intervals from strings with the function DATE_INTERVAL_CREATE_FROM_DATE_STRING(). There’s lots of flexibility in what you can pass to the function. It accepts the kinds of strings you might naturally type, though it does not accept the ISO format. **Listing 58** (DateIntervals.php in the materials for this session) shows several examples; the results are shown in **Figure 30**.

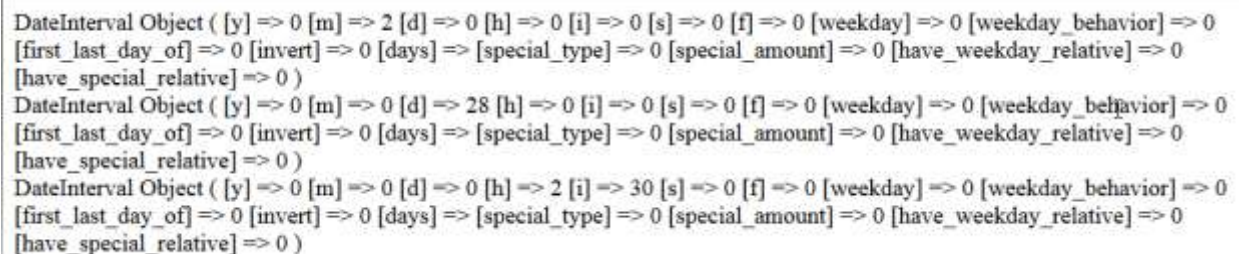
Listing 58. You can also create date intervals with a function call and a simple string.

```
<?php
    $today = date_create();

    $twomonths = new DateInterval("P2M");
    $fourweeks = date_interval_create_from_date_string('4 weeks');
    $twoandahalfhours = date_interval_create_from_date_string('2 hours 30 minutes');

    print_r($twomonths);
    echo "<br>";
    print_r($fourweeks);
    echo "<br>";
    print_r($twoandahalfhours);

?>
```



```
DateInterval Object ( [y] => 0 [m] => 2 [d] => 0 [h] => 0 [i] => 0 [s] => 0 [f] => 0 [weekday] => 0 [weekday_behavior] => 0
[first_last_day_of] => 0 [invert] => 0 [days] => [special_type] => 0 [special_amount] => 0 [have_weekday_relative] => 0
[have_special_relative] => 0 )
DateInterval Object ( [y] => 0 [m] => 0 [d] => 28 [h] => 0 [i] => 0 [s] => 0 [f] => 0 [weekday] => 0 [weekday_behavior] => 0
[first_last_day_of] => 0 [invert] => 0 [days] => [special_type] => 0 [special_amount] => 0 [have_weekday_relative] => 0
[have_special_relative] => 0 )
DateInterval Object ( [y] => 0 [m] => 0 [d] => 0 [h] => 2 [i] => 30 [s] => 0 [f] => 0 [weekday] => 0 [weekday_behavior] => 0
[first_last_day_of] => 0 [invert] => 0 [days] => [special_type] => 0 [special_amount] => 0 [have_weekday_relative] => 0
[have_special_relative] => 0 )
```

Figure 30. Date intervals have many properties, including each component of a datetime (except time zone).

`DATE_ADD()` accepts two parameters, a datetime and a date interval, and adds the date interval to the original datetime. While the function returns the new value, note that it also changes the first parameter.

`DATE_SUB` accepts the same two parameters, but subtracts the date interval from the specified datetime and assigns the new value to the first parameter.

Listing 59 (DateAddSub.php in the materials for this session) demonstrates, using the date intervals created in the previous example. Note the need to keep recreating the `$today` variable in order to start from the current date and time. The results are shown in **Figure 31**.

Listing 59. The `DATE_ADD()` and `DATE_SUB()` functions add or subtract the specified interval to the specified datetime, changing its value.

```
<?php
    $today = date_create();

    $twomonths = new DateInterval("P2M");
    $fourweeks = date_interval_create_from_date_string('4 weeks');
    $twoandahalfhours = date_interval_create_from_date_string('2 hours 30 minutes');

    echo "Today: ";
    print_r($today);
```

```
echo "<br>Two months from now: ";
print_r(date_add($today, $twomonths));
echo "<br>Two months ago: ";
$today = date_create();
print_r(date_sub($today, $twomonths));
echo "<br>Four weeks from now: ";
$today = date_create();
print_r(date_add($today, $fourweeks));
echo "<br>Four weeks ago: ";
$today = date_create();
print_r(date_sub($today, $fourweeks));
echo "<br>Two-and-a-half hours from now: ";
$today = date_create();
print_r(date_add($today, $twoandahalfhours));
echo "<br>Two-and-a-half hours ago: ";
$today = date_create();
print_r(date_sub($today, $twoandahalfhours));
?>
```

```
Today: DateTime Object ( [date] => 2019-07-01 14:10:39.675334 [timezone_type] => 3 [timezone] => UTC )
Two months from now: DateTime Object ( [date] => 2019-09-01 14:10:39.675334 [timezone_type] => 3 [timezone] => UTC )
Two months ago: DateTime Object ( [date] => 2019-05-01 14:10:39.675334 [timezone_type] => 3 [timezone] => UTC )
Four weeks from now: DateTime Object ( [date] => 2019-07-29 14:10:39.675334 [timezone_type] => 3 [timezone] => UTC )
Four weeks ago: DateTime Object ( [date] => 2019-06-03 14:10:39.675334 [timezone_type] => 3 [timezone] => UTC )
Two-and-a-half hours from now: DateTime Object ( [date] => 2019-07-01 16:40:39.675334 [timezone_type] => 3 [timezone] => UTC )
Two-and-a-half hours ago: DateTime Object ( [date] => 2019-07-01 11:40:39.675334 [timezone_type] => 3 [timezone] => UTC )
```

Figure 31. DATE_ADD() and DATE_SUB let you add and subtract date intervals to and from datetimes.

DATE_DIFF() computes the difference between two datetimes and returns a date interval.

Listing 60 (DateDiff.php in the materials for this session) shows an example. It also uses the DATE_INTERVAL_FORMAT() function to convert the dateinterval into days only. Analogous to DATE_FORMAT(), DATE_INTERVAL_FORMAT accepts a date interval and a format string and returns a formatted string. The format string for this function requires a % symbol in front of each character; while it accepts many of the same format characters as DATE_FORMAT(), some of them have different meanings here. The list is available at <https://www.php.net/manual/en/dateinterval.format.php>. For this example, note that 'a' indicates the total number of days.

The output from the example is shown in **Figure 32**.

Listing 60. DATE_DIFF() computes the difference between two datetimes as a date interval. DATE_INTERVAL_FORMAT() lets us display that difference in a desired format.

```
<?php
    $today = date_create();
    $swfox = date_create('2019-10-24');

    $untilswfox = date_diff($swfox, $today);

    echo "There are ".date_interval_format($untilswfox, '%a days')." until Southwest
Fox";
?>
```

```
There are 114 days until Southwest Fox
```

Figure 32. Use `DATE_DIFF()` to figure out how long between two datetimes.

Numeric functions

There are only a few numeric functions you're likely to use often. As its name suggest, `ROUND()` rounds numbers to a specified number of decimal places. `RAND()` generates random numbers.

`ROUND()` accepts three parameters; only the first, the number to be rounded is required. When only one parameter is passed, the number is rounded to an integer. The second parameter is the rounding precision. Pass a positive number to specify decimal places; pass a negative to specify rounding before the decimal point. **Listing 61** (`Round.php` in the materials for this session) shows each of these three cases, and demonstrates that you can use `ROUND()` on integer values as well as floats. The results are shown in **Figure 33**.

Listing 61. `ROUND()` rounds numbers.

```
<?php
$val = 1234.56789;
$intval = 3767;

echo round($val).nl2br("\n");
echo round($val, 2).nl2br("\n");
echo round($val, -2).nl2br("\n");
echo round($intval, -1).nl2br("\n");
?>
```

```
1235
1234.57
1200
3770
```

Figure 33. `ROUND()` lets you round both floats and integers.

The optional third parameter lets you determine what happens when the leftmost digit to be rounded is 5. There are four options, documented at <https://www.php.net/manual/en/function.round.php>.

`RAND()` returns a random integer. If you pass no parameters, the value is in the range from 0 to a system-specified maximum (which you can look up with the `GETRANDMAX()` function). Alternatively, you can pass a minimum and maximum to return values in a different range. **Listing 62** (`Rand.php` in the materials for this session) demonstrates, with the results shown in **Figure 34**.

Listing 62. The `RAND()` function returns random integers.

```
<?php
  echo rand().nl2br("\n");
  echo rand(1,27).nl2br("\n");
  echo getrandmax().nl2br("\n");
?>
```

```
649885793
2
2147483647
```

Figure 34. Use the `RAND()` function to return random integers in a specified or default range.

Array functions

PHP has a huge collection of functions for working with arrays. This section looks at a few that you're likely to use often.

First, it's worth noting that the `ARRAY()` notation for populating an array that was introduced in "Arrays," earlier in this document is, in fact, a function that accepts a set of values and puts it into an array.

The `ARRAY_PUSH()` function lets you add elements to the end of an array. It isn't strictly necessary, since you can actually add elements directly. `ARRAY_PUSH()` accepts an array and one or more items to add to it; items are added at the end. **Listing 63** (`ArrayPush.php` in the materials for this session) creates an empty array and then adds three elements, creating the same array as the first executable line in **Listing 25**.

Listing 63. You can populate or add to an array with `ARRAY_PUSH()`.

```
<?php
  $Organizers = array();

  array_push($Organizers, 'Rick');
  array_push($Organizers, 'Doug');
  array_push($Organizers, 'Tamar');

  print_r($Organizers);
?>
```

You can also add elements to an array simply by assigning them to the array. The code in **Listing 64** (`ArrayDirectAdd.php` in the materials for this session) has the same result as that in **Listing 63**.

Listing 64. You can add elements to an array directly.

```
<?php
  $Organizers = array();

  $Organizers[] = 'Rick';
```

```
$Organizers[] = 'Doug';
$Organizers[] = 'Tamar';

print_r($Organizers);
?>
```

While these examples use a one-dimensional array, the function and direct assignment work for multi-dimensional arrays, as well. However, if you're using an associative array, the direct notation is better, as it lets you specify a key for the new value, as in **Listing 65** (AssociativeArrayAdd.php in the materials for this session).

Listing 65. To add elements in an associative array, the direct notation is better.

```
<?php
$Organizers = array();

$Organizers['Rick'] = array(
    "First"    => "Rick",
    "Last"     => "Schummer",
    "Company"  => "White Light"
);
$Organizers['Doug'] = array(
    "First"    => "Doug",
    "Last"     => "Hennig",
    "Company"  => "Stonefield"
);
$Organizers['Tamar'] = array(
    "First"    => "Tamar",
    "Last"     => "Granor",
    "Company"  => "Tomorrow's Solutions",
    "Age"      => 60
);

?>
```

The SORT() functions sorts an array. Pass an array and the values are sorted in place. By default, the sort order is determined by the data, so numbers are sorted numerically and strings are sorted as strings. PHP is smart enough to sort numeric strings correctly (avoiding the common problem of winding up with 1, 11, 12, 2, 3, 4, ...). **Listing 66** (ArraySort.php in the materials for this session) shows sorting of an array of numbers, an array of numeric strings, and an array of names; the results are shown in **Figure 35**.

Listing 66. The SORT() function is smart enough to sort numeric strings correctly.

```
<?php
$numns = array(37, 19, 7, 552, 11, 39, 98);
$strs = array('37', '19', '7', '552', '11', '39', '98');
$names = array('Tamar', 'Rick', 'Doug', 'Therese', 'Marshal');

sort($numns);
echo "Sorting numbers: ";
print_r($numns);
echo "<br>";
```



```
sort($strs);
echo "Sorting numeric strings: ";
print_r($strs);
echo "<br>";

sort($names);
echo "Sorting strings: ";
print_r($names);
echo "<br>";

?>
```

```
Sorting numbers: Array ( [0] => 7 [1] => 11 [2] => 19 [3] => 37 [4] => 39 [5] => 98 [6] => 552 )
Sorting numeric strings: Array ( [0] => 7 [1] => 11 [2] => 19 [3] => 37 [4] => 39 [5] => 98 [6] => 552 )
Sorting strings: Array ( [0] => Doug [1] => Marshal [2] => Rick [3] => Tamar [4] => Therese )
```

Figure 35. PHP's SORT() function is pretty smart.

If you pass a multidimensional array, SORT() sorts based on the first element of each row. So the code in **Listing 67** (SortMultiDim.php in the materials for this session) produces the results in **Figure 36**, with the data sorted by the “First” element.

Listing 67. Sorting a multidimensional array uses the first element at the lowest level.

```
<?php
$Organizers = array(
    "Hennig" => array(
        "First"    => "Doug",
        "Last"     => "Hennig",
        "Company"  => "Stonefield"
    ),
    "Granor" => array(
        "First"    => "Tamar",
        "Last"     => "Granor",
        "Company"  => "Tomorrow's Solutions",
        "Age"      => 60
    ),
    "Schummer" => array(
        "First"    => "Rick",
        "Last"     => "Schummer",
        "Company"  => "White Light"
    )
);

sort($Organizers);
var_dump($Organizers);

?>
```

```
array (size=3)
  0 =>
    array (size=3)
      'First' => string 'Doug' (length=4)
      'Last' => string 'Hennig' (length=6)
      'Company' => string 'Stonefield' (length=10)
  1 =>
    array (size=3)
      'First' => string 'Rick' (length=4)
      'Last' => string 'Schummer' (length=8)
      'Company' => string 'White Light' (length=11)
  2 =>
    array (size=4)
      'First' => string 'Tamar' (length=5)
      'Last' => string 'Granor' (length=6)
      'Company' => string 'Tomorrow's Solutions' (length=20)
      'Age' => int 60
```

Figure 36. When you sort a multidimensional array with `SORT()`, PHP bases the sort on the first element of each item.

`SORT()` accepts an optional second parameter to specify how to perform the sorting; the choices are documented at <https://www.php.net/manual/en/function.sort.php>.

Creating custom functions

As in most programming languages, you can define your own functions in PHP. The syntax is similar to that of other languages, with its own PHP twist. **Listing 68** shows PHP's function definition syntax.

Listing 68. The syntax for defining a function in PHP isn't very different than in other languages.

```
function name($arg1, $arg2, ... $argn)
{
    //commands
    return $retval;
}
```

You can put any PHP code inside a function. Functions are called as in other languages, by specifying the function name and enclosing any parameters in parentheses. **Listing 69** (SimpleFunction.php in the materials for this session) shows a simple function that accepts a datetime and returns a string with the datetime formatted into a specific format; you might use such a function in an application where this particular format is the standard. **Figure 37** shows the results.

Listing 69. This simple function transforms a datetime into a particular format.

```
<?php
function MyDateFormat($date) {
    return date_format($date, 'l, F j, Y');
}

echo MyDateFormat(date_create()).nl2br("\n");
```

```
echo MyDateFormat(date_create('1958-9-28')).nl2br("\n");
?>
```

```
Monday, July 1, 2019
Sunday, September 28, 1958
```

Figure 37. You might use a function to format datetimes into a particular format.

PHP does not require you to define a function before using it, and doesn't object to a function definition in the middle of a block of code. So the code in **Listing 70** is functionally equivalent to that in **Listing 69**. More broadly, this means you can decide whether to define functions at the top of a page or the bottom. (While you can put them in the middle, I don't recommend it; it's not very readable.)

Listing 70. PHP doesn't care whether functions are defined before you use them.

```
<?php
echo MyDateFormat(date_create()).nl2br("\n");

function MyDateFormat($date) {
    return date_format($date, 'l, F j, Y');
}

echo MyDateFormat(date_create('1958-9-28')).nl2br("\n");
?>
```

By default, parameters are passed to functions by value. To specify that a parameter is passed by reference, precede that parameter with an ampersand in the function definition.

Listing 71 (PassByReference.php in the materials for this session) shows an example; the result is shown in **Figure 38**.

Listing 71. You can specify that a particular function parameter is passed by reference by preceding it with an ampersand in the function definition.

```
<?php
function add_name(&$string) {
    $string .= "\r\nmodified by add_name";
}

$str = 'This is a test';
add_name($str);
echo nl2br($str);
?>
```

```
This is a test
modified by add_name
```

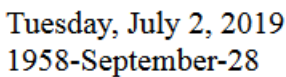
Figure 38. PHP functions can accept some parameters passed by reference.

PHP also lets you specify a default value for a parameter. When you do so and omit that parameter from the function call, the default value is used. To demonstrate, I've extended the MyDateFormat function from **Listing 69**. The new version, shown in **Listing 72** (DefaultParm.php in the materials for this session), accepts an optional second parameter of format. If you omit that parameter, it uses the format that was built into the earlier version of the function. The results are shown in **Figure 39**; note the difference in the second call here from the previous example.

Listing 72. This version of MyDateFormat lets you pass a format, but if you don't, it has one built-in.

```
<?php
function MyDateFormat($date, $format = 'l, F j, Y') {
    return date_format($date, $format);
}

echo MyDateFormat(date_create()).n12br("\n");
echo MyDateFormat(date_create('1958-9-28'),'Y-F-j').n12br("\n");
?>
```



```
Tuesday, July 2, 2019
1958-September-28
```

Figure 39. Specifying a default parameter for a function lets you set up the behavior you usually want, but still easily change it.

Using INCLUDE to create libraries of functions

You may want the same functions to be available on multiple pages. Save your collection of functions in a file with a PHP extension. To make those functions available in another page, put the INCLUDE command at the top of the page as in **Listing 73**. INCLUDEs can be nested, so if you have a number of function files, you can create a single file that includes them all, and then include that one file on your pages.

Listing 73. The INCLUDE command lets you make all the functions from another page available in the current page.

```
<?php
include 'myfunctions.php'

//code that uses the functions in myfunctions.php

?>
```

The next major section of this document includes examples of such function libraries, in this case, used for accessing databases.

Working with data

Many of the examples to this point may have seemed forced because variables were populated and then used to demonstrate. But PHP also provides a way to retrieve data

from databases, including MySQL and SQL Server. Many of the features discussed earlier in this paper really shine when processing data.

Database access is provided through extensions to PHP. There are both generic database extensions and database-specific extensions. On the whole, if you're going to be dealing with a particular database, the database-specific extensions are easier to work with. In this section, we'll look at specific extensions for MySQL and SQL Server.

In order to use extensions, they have to be enabled in your PHP.INI file. When you install PHP, the default INI file includes the necessary lines for many extensions, but they may be commented out. Of course, the way you enable support on a webserver will depend on that host's tools.

For the examples in this section, we'll work with the example Chinook database that's available for quite a few database servers. You can find Chinook at <https://github.com/lerocha/chinook-database>.

Talking to MySQL with MySQLi

There are several extensions for working with MySQL data, but the most current is MySQLi, which supports both an object-oriented approach and a procedural approach. Supporting both means there are a set of MySQLi classes, but that the core functionality is also exposed via a set of functions. Here, I'll look at the functions. In fact, you can do most of what you need with just a few functions.

To use MySQLi, you need to uncomment the line `extension=mysqli` in the PHP.INI file.

Connecting to a database

As with any code that uses an external database, step 1 is connecting to the database. The relevant function is `mysqli_connect()`; the syntax is shown in **Listing 74**. All the parameters are optional; their default values are lookups in the PHP.INI file. **Listing 75** shows an example connecting to the Chinook database as `DemoUser`.

Listing 74. Use `mysqli_connect()` to connect to a MySQL database.

```
$conn = mysqli_connect( $host, $username, $password, $database, $port, $socket)
```

Listing 75. To make the connection, pass the appropriate values.

```
<?php
    $hostname = "localhost";
    $username = "DemoUser";
    $password = "demo";
    $database = "chinook";
    $conn = mysqli_connect($hostname, $username, $password, $database);
?>
```

Of course, it's best to check whether the connection worked and handle the case where it doesn't. The function in **Listing 76** (`ConnectMySQL.php` in the materials for this session)

attempts the connection, and handles the error if it fails. The `DIE()` function used in the error case halts the current code while returning the enclosed string. In this case, we call the MySQLi function `mysqli_connect_error()` to return the error message that explains why the connection failed.

Listing 76. This function attempts to connect to the Chinook database. It returns the connection object if successful, and fails if not.

```
function connect2db() {
    $hostname = "localhost";
    $username = "DemoUser";
    $password = "demo";
    $database = "chinook";
    $conn = mysqli_connect($hostname, $username, $password, $database);

    if( $conn == false ) {
        echo "Connection could not be established.<br />";
        die( print_r( mysqli_connect_error(), true));
    }

    return $conn;
}
```

You might wonder why the username and password are hard-coded in this function. The websites I've created with PHP are public sites where users don't log in and data is available to all. So all data is accessed using the same MySQL user. (Because PHP code isn't included in what's shown when a user views the source of a web page, having the username and password in the PHP code doesn't expose it to the world.)

Sending and receiving data

To send commands to MySQL, use the `mysqli_query()` function. It sends a single command to MySQL and returns the results. Although the name says "query," it accepts all kinds of MySQL commands. If something goes wrong, the return value is `False`.

The code in **Listing 77** (`GetMySQLData.php` in the materials for this session) includes a function `runquery` that's a wrapper for `mysqli_query()`, and code that calls `runquery`. The function uses the `connect2db()` function defined above. If the command fails, the function calls `mysqli_error()` to find out what went wrong.

Listing 77. The code here sends a query to MySQL and reports whether it was successful.

```
<?php

$empquery = 'SELECT FirstName, LastName, BirthDate FROM employee;';
$emps = runquery($empquery);

if ($emps)
    echo 'query successful';

function connect2db() {
    $hostname = "localhost";
```

```
$username = "DemoUser";
$password = "demo";
$databse = "chinook";
$conn = mysqli_connect($hostname, $username, $password, $database);

if( $conn == false ) {
    echo "Connection could not be established.<br />";
    die( print_r( mysqli_connect_error(), true));
}

return $conn;
}

function runquery($query) {
    $conn = connect2db();
    $result = mysqli_query($conn, $query);

    if ($result == false) {
        $errmessage = mysqli_error($conn);
        echo $errmessage;
        die( print_r( mysqli_error(), true));
    }

    return $result;
}

?>
```

Working with MySQL data

Of course, you'll want to do more with the data you retrieve than just acknowledge that you got it. The value returned from `mysqli_query()` is an object containing the results. MySQLi provides a couple of ways to traverse those results.

The `mysqli_fetch_object()` function returns the next row of the results as an object. You can use a loop to grab each row in turn, as in **Listing 78** (`EmployeesMySQLObject.php` in the materials for this session). For this and subsequent examples in this section, the `connect2db()` and `runquery()` functions have been moved to a separate PHP file, `mysqldbfns.php`, which is included in the page where we want to access data (and in the materials for this session).

This example also demonstrates how a single page can mix PHP and HTML. You can simply insert PHP code into HTML by bracketing it with `<?php ... ?>`. The example also uses the `<?= ...?>` shorthand notation for expressions.

Figure 40 shows the result. It's not particularly attractive, but making it prettier is just a matter of adding HTML or CSS.

Listing 78. You can mix PHP and HTML in order to use data in a page.

```
<?php
    include 'mysqldbfns.php';
```

```
?>

<!DOCTYPE html>
<html lang="en">
  <head>
  </head>

  <body>
  <h1>Chinook employees</h1>

  <?php
    $empquery = 'SELECT FirstName, LastName, BirthDate FROM employee;';
    $emps = runquery($empquery);
  ?>

  <table>
    <tr>
      <td><h2>First name</h2></td>
      <td><h2>Last name</h2></td>
      <td><h2>Birthdate</h2></td>
    </tr>

    <?php
      while ($emp = mysqli_fetch_object($emps)) {
    ?>
    <tr>
      <td><?=$emp->FirstName?></td>
      <td><?=$emp->LastName?></td>
      <td><?=date_format(date_create($emp->BirthDate),'F j, Y')?></td>

    <?php
      }
    ?>

  </table>

  </body>

</html>
```


Chinook employees

First name Last name Birthdate

Andrew	Adams	February 18, 1962
Nancy	Edwards	December 8, 1958
Jane	Peacock	August 29, 1973
Margaret	Park	September 19, 1947
Steve	Johnson	March 3, 1965
Michael	Mitchell	July 1, 1973
Robert	King	May 29, 1970
Laura	Callahan	January 9, 1968

Figure 40. It doesn't take much code to show data on a page.

One final note about this example: even though the BirthDate field is a DateTime in MySQL, it comes to PHP as a string, and so has to be handled that way.

If you prefer to work with arrays rather than objects, you can use the `mysqli_fetch_assoc()` function to turn a record of the query result into an associative array. **Listing 79** (`EmployeesMySQLAssoc.php` in the materials for this session) shows the portion of the previous example that changes when you do it that. In this case, to refer to a field of the table, you use its name (capitalized as in the database) as the key for an element of the array. The output, of course, is the same.

Listing 79. If you prefer, you can turn the rows of the query result into associative arrays rather than objects.

```
<?php
while ($emp = mysqli_fetch_assoc($emps)) {
?>
<tr>
  <td><?=$emp['FirstName']?></td>
  <td><?=$emp['LastName']?></td>
  <td><?=date_format(date_create($emp['BirthDate']),'F j, Y')?></td>

<?php
}
?>
```

The key insight about both `mysqli_fetch_object()` and `mysqli_fetch_assoc()` is that they keep track of where you are in the result and always return the next row.

Talking to SQL Server with SQLSRV

The SQLSRV extension lets you talk to SQL Server data. It's procedural with functions to perform all the necessary tasks. Unlike MySQLi, SQLSRV is not automatically installed when you install PHP. To use it locally, you'll need to download it and then add the necessary line to PHP.INI. You'll find instructions at <https://docs.microsoft.com/en-us/sql/connect/php/loading-the-php-sql-driver?view=sql-server-2017>.

Connecting to a database

The `sqlsrv_connect()` function makes a database connection. It expects two parameters, the host and a connection string. The connection string is passed as an associative array.

Listing 80 shows the very simple syntax. The function returns either a connection resource or `False`, if a connection cannot be made. **Listing 81** (`ConnectSQLServer.php` in the materials for this session) provides a function for making the connection, including handling failure.

Listing 80. The `sqlsrv_connect()` function lets you connect to SQL Server.

```
$conn = sqlsrv_connect($host, $connstr)
```

Listing 81. When you use `sqlsrv_connect()`, you should check the return value to see whether a connection was made.

```
<?php
$conn = connect2db();
if ($conn)
    echo 'Connection made';

function connect2db() {
    $hostname = "localhost";
    $username = 'DemoUser';
    $password = 'mYdemoUser';
    $database = 'chinook';
    $connectionInfo = array("Database"=>$database, "UID"=>$username, "PWD"=>$password,
"CharacterSet"=>"UTF-8");
    $conn = sqlsrv_connect($hostname, $connectionInfo);

if( $conn == false ) {
    echo "Connection could not be established.<br />";
    die( print_r( sqlsrv_errors(), true));
}
return $conn;
}

?>
```

As in the MySQL examples, the username and password are hard-coded here because my experience is that most websites use a single account to connect to a database.

SQLSRV has a single function for retrieving error messages, `sqlsrv_errors()`. Called after an error, it returns information about the error that occurred.

Sending and receiving data

The `sqlsrv_query()` function lets you send a command to SQL Server. As with its MySQL analogue, any valid command can be sent. The function returns `False` if it fails.

Listing 82 (`GetSQLServerData.php` in the materials for this session) shows a `runquery` function that wraps a call to `sqlsrv_query()` with some error handling. The `runquery` function uses the `connect2db()` function to connect to the database and then, if a connection was made, calls `sqlsrv_query()` to send the command it receives as a parameter.

Listing 82. The code to retrieve SQL Server data in PHP isn't very different from the MySQL version.

```
<?php

$empquery = 'SELECT FirstName, LastName, BirthDate FROM employee;';
$emps = runquery($empquery);

if ($emps)
    echo 'query successful';

function connect2db() {
    $hostname = "localhost";
    $username = 'DemoUser';
    $password = 'mYdemoUser';
    $database = 'chinook';
    $connectionInfo = array("Database"=>$database, "UID"=>$username,
"PWD"=>$password, "CharacterSet"=>"UTF-8");
    $conn = sqlsrv_connect($hostname, $connectionInfo);

    if( $conn == false ) {
        echo "Connection could not be established.<br />";
        die( print_r( sqlsrv_errors(), true));
    }

    return $conn;}

function runquery($query) {
    $conn = connect2db();
    $result = sqlsrv_query($conn, $query);

    if ($result == false) {
        $errmessage = sqlsrv_errors();
        echo $errmessage;
        die( print_r( sqlsrv_errors(), true));
    }

    return $result;
}

?>
```

Working with SQL Server data

As with MySQL data, of course, you'll want to do more than confirm that you were able to send or receive data. The `sqlsrv_fetch_array()` and `sqlsrv_fetch_object()` functions put one record of the result into an array or object, respectively. You can then use that data in HTML code.

However, in order to use those functions, you must do so while the connection that created them still exists. So it's not good enough to simply return the resource returned by `sqlsrv_query()` from the custom `runquery()` function and process it. Instead, you need to loop through the results inside `runquery()` and build an array of results. **Listing 83** shows an updated version of `runquery()` that builds such an array and returns that array. The second parameter to `sqlsrv_fetch_array()` indicates that the record should be retrieved as an associative array. You can instead retrieve them as an indexed array by passing `SQLSRV_FETCH_NUMERIC` or in both forms by passing `SQLSRV_FETCH_BOTH`.

Listing 83. With SQLSRV, it's best to process the results in the same function that retrieves them.

```
function runquery($query) {
    $conn = connect2db();
    $result = sqlsrv_query($conn, $query);

    if ($result == false) {
        $errmessage = sqlsrv_errors();
        echo $errmessage;
        die( print_r( sqlsrv_errors(), true));
    }

    $rows = array();
    while ($row = sqlsrv_fetch_array($result, SQLSRV_FETCH_ASSOC)) {
        $rows[] = $row;
    }
    return $rows;
}
```

Using this function (as well as `connect2db()`, both contained in a new file `sqlserverdbfns.php`, which in the materials for this session), the code in **Listing 84** (`EmployeesSQLServerAssoc.php` in the materials for this session) produces the output in **Figure 40**. One difference between the data returned by MySQL using MySQLi and that returned by SQL Server using SQLSRV is the handling of datetimes. SQLSRV creates actual datetimes rather than returning them as strings. Thus, the code doesn't need to use `DATE_CREATE()` to turn the string into a datetime for formatting.

Listing 84. Once you've put SQL Server data into an array, displaying it is straightforward.

```
<?php
    include 'sqlserverdbfns.php';
?>

<!DOCTYPE html>
<html lang="en">
```

```
<head>
</head>

<body>
<h1>Chinook employees</h1>

<?php
    $empquery = 'SELECT FirstName, LastName, BirthDate FROM employee;';
    $emps = runquery($empquery);
?>

<table>
    <tr>
        <td><h2>First name</h2></td>
        <td><h2>Last name</h2></td>
        <td><h2>Birthdate</h2></td>
    </tr>

    <?php
        foreach ($emps as $emp) {
            ?>
            <tr>
                <td><?=$emp['FirstName']?></td>
                <td><?=$emp['LastName']?></td>
                <td><?=date_format($emp['BirthDate'],'F j, Y')?></td>

            <?php
            }
            ?>
        </table>

</body>

</html>
```

You can do the same thing by building an array of objects. **Listing 85** shows an alternate version of `runquery()` that uses `sqlsrv_fetch_object()` to build an array of objects; it's included in `sqlserverobjdbfns.php`.

Listing 85. This version of `runquery` returns the query results as an array of objects.

```
function runquery($query) {
    $conn = connect2db();
    $result = sqlsrv_query($conn, $query);

    if ($result == false) {
        $errmessage = sqlsrv_errors();
        echo $errmessage;
        die( print_r( sqlsrv_errors(), true));
    }

    $rows = array();
    while ($obj = sqlsrv_fetch_object($result)) {
        $rows[] = $obj;
    }
}
```

```
    }  
    return $rows;  
}
```

Listing 86 (EmployeesSQLServerObject.php in the materials for this session) shows code that uses this version of `runquery()` to produce the output shown in **Figure 40**.

Listing 86. This code produces the list of employees from an array of objects.

```
<?php  
    include 'sqlserverobjdbfns.php';  
?>  
  
<!DOCTYPE html>  
<html lang="en">  
    <head>  
    </head>  
  
    <body>  
    <h1>Chinook employees</h1>  
    <?php  
        $empquery = 'SELECT FirstName, LastName, BirthDate FROM employee;';  
        $emps = runquery($empquery);  
    ?>  
  
    <table>  
        <tr>  
            <td><h2>First name</h2></td>  
            <td><h2>Last name</h2></td>  
            <td><h2>Birthdate</h2></td>  
        </tr>  
  
        <?php  
            foreach ($emps as $emp) {  
    ?>  
        <tr>  
            <td><?=$emp->FirstName?></td>  
            <td><?=$emp->LastName?></td>  
            <td><?=date_format($emp->BirthDate,'F j, Y')?></td>  
  
        <?php  
            }  
    ?>  
        </table>  
  
    </body>  
  
</html>
```

Resources

This paper is a starting point for working with PHP, but there's plenty of material available to take you farther. The PHP documentation is online at <https://www.php.net/>. Searching

for PHP and the name of a function almost always yields the official documentation for that function. The documentation includes comments and examples from users that often provide additional insight.

Much of PHP is also documented at <https://www.w3schools.com/>. Again, searches typically include w3school pages in the results. These pages often include simple runnable examples.

Finally, for pretty much any web development subjects, [StackOverflow](#) is the premier question-and-answer site.