

Module 4

This is a single, concatenated file, suitable for printing or saving as a PDF for offline viewing. Please note that some animations or images may not work.

Module 4 Study Guide and Deliverables

Readings:	Murach's PHP and MySQL: Chapters 1–4 & 16–18
Assignments:	Assignment 4 due Tuesday, February 14 at 6:00 AM ET
Term Project:	Term Project Milestone: Project Selections due Tuesday, February 14 at 6:00 AM ET
Live Classroom:	<ul style="list-style-type: none">• Tuesday, February 7, 6:00–8:00 PM ET• Facilitator live office: TBD

■ Introduction to PHP and MySQL

Learning Objectives

By reading the lectures and textbook, participating in the discussions, and completing the assignments, you will be able to do the following:

- Differentiate the use cases of PHP.
- Write PHP scripts using variables, functions, conditionals, and loops.
- Identify various PHP data types and structures.
- Create MySQL databases and tables.
- Perform CRUD (Create, Read, Update, and Delete) tasks with PHP and MySQL.
- Upload files to a web server.

Introduction

PHP is a server-side scripting language that is used to create dynamic websites. In the vast majority of cases, PHP is used with a database. In this lecture, you will learn how to create PHP scripts that interact with MySQL databases. We won't cover every feature and attribute of the PHP programming language in this course, but we will cover the most common ones and even some selected areas of advanced functionality.

NOTE: *The lectures found in this module and in subsequent ones will assume that you have some level of programming knowledge. You don't need to be an expert, but you should understand the basic programming concepts such as variables, data types, functions, conditionals, loops, etc.*

PHP has syntax very similar to C, JavaScript, and other languages derived from C.

There is a lot of information to learn, but it's going to be fun!



Why Use PHP?

As outlined in the previous lecture, PHP is a server-side scripting language. It is an interpreted language and therefore does not need to be compiled before execution. PHP is open-source and free software. It is cross platform, meaning you can develop, deploy, and use it on Windows, Mac, and Linux machines.

PHP is powerful, scalable, and very robust. Many large websites such as Facebook, Yahoo, and Wikipedia use PHP for their server-side development language. It was developed with web development in mind.

PHP is fully object oriented since version five. It has great [documentation](#) and has a large and very active developer community. You will certainly need to refer to the PHP documentation numerous times as you work through examples and assignments in this course. It is suggested that you bookmark the site and even keep a browser tab open with that page so it is readily accessible to you as you are working on your PHP projects.

To learn more about the capabilities of PHP, be sure to view this article [What Can PHP Do?](#)

History of PHP

PHP came about as a programming/scripting language when Rasmus Lerdorf needed a set of tools to manage his personal home page in 1994. The set of tools he created at the time were extremely tiny compared to the features that

PHP offers today. He coined the scripts he created as “Personal Home Page Tools” or PHP.

In 1995, version 2 was created and released to the public. Over the years much work has been put into PHP, which really started to accelerate in 1998 with the release of version 3. Andi Gutmans and Zeev Suraski really took over much of the development of PHP and re coined it as PHP: Hypertext Preprocessor (a recursive acronym, geek humor).

Version 4 was released in 2000 in which Zend rewrote the core of PHP that resulted in a more modular Zend Engine 1.

Version 5 was released in 2004. This version introduced Zend Engine 2, which offered better performance, and improved OOP support. This is the latest major and stable release and the version we will be using in this course.

Version 6 was a failed initiative and development has stopped.

Version 7 is currently being developed now.

PHP Requirements

There are a few components you will need in order to start writing PHP code and to then run/parse it.

Plain Text Editor or Integrated Development Environment (IDE)

This will serve as your development environment for writing, testing, and debugging your PHP code.

Some examples include:

Mac

- TextEdit (included with OS)
- TextMate
- Sublime Text
- Eclipse or Aptana
- Netbeans
- Komodo
- Coda
- BBEdit

Windows

- Notepad (included with OS)
- Notepad++
- Sublime Text
- E Text Editor
- Eclipse or Aptana

- Netbeans
- Komodo

There are other viable choices as well. Just be sure to select a *plain* text editor or IDE and stay away from word processing applications such as Microsoft Word. It is highly recommended that you choose a solution that includes the following features:

- Line numbers
- Code coloring/syntax highlighting
- Ability to navigate between multiple files within the editor
- Strong search/replace features
- Automatic auto-pairing (brackets, parenthesis, quotes)

Web Browser(s)

You'll need the latest versions of one or more (preferably 2+) web browsers such as:

- Google Chrome
- Mozilla Firefox
- Apple Safari
- Microsoft Internet Explorer

Web Server running PHP/MySQL

A web server will be needed to host your files and to work with the PHP engine in order to parse and return your PHP code as HTML. While there are various types of web servers available, we recommend that you use the *Apache* web server for this course. You will want to ensure that a later version of PHP is installed such as PHP 5.4 or greater. The latest version at the time of this writing is version 5.6.11 and it is recommended that you use PHP 5.6.x or greater.

Important: Even though PHP 5.4+ has a built in web server, Apache should still be used.

In addition to the web server and PHP, you'll want to ensure that the server also has access to a database server. For the first part of this course, we will be using MySQL.

There are a few different ways to setup or obtain access to a web server meeting the requirements above.

1. Install it all on your own on your personal computer or a virtual machine (not recommended).
 - [Read more information about setting up a web server.](#)
2. Use an existing installer that sets up and configures Apache, PHP, and MySQL to all work together. (Recommended).
 - [LAMP](#)
 - [MAMP](#)

– XAMPP

– **XAMPP works on all operating systems**, LAMP is for Linux, WAMP is for Windows, and MAMP is for Mac.

3. Sign up for a hosting account online.

- Be sure that the service offers the requirements listed above. Your instructor will share recommendations with you and it is *highly recommended that you don't signup for any hosting service until you receive instructions from your instructor.*

Other Software

Other software might be useful while writing PHP code. If you are hosting your files on a remote web server, you'll probably want to use a FTP client such as Filezilla to move the files from your local machine to the server. There are other options as well which will be shared by your instructor.

Timely Setup is Critical!

It is very important that you **quickly** get your programming environment setup in order to write and execute your PHP applications. This course moves very quickly and a huge amount of information is covered in six short weeks. You'll need the environment setup within the first day or two of this course starting.

Your First PHP Page

Now that you have learned what tools you need in order to write and execute PHP scripts, let's write our first bit of PHP.

Start off by creating a basic HTML page, ensuring that it uses valid HTML syntax (<https://validator.w3.org/>). We'll be using HTML 5 code examples in the course.

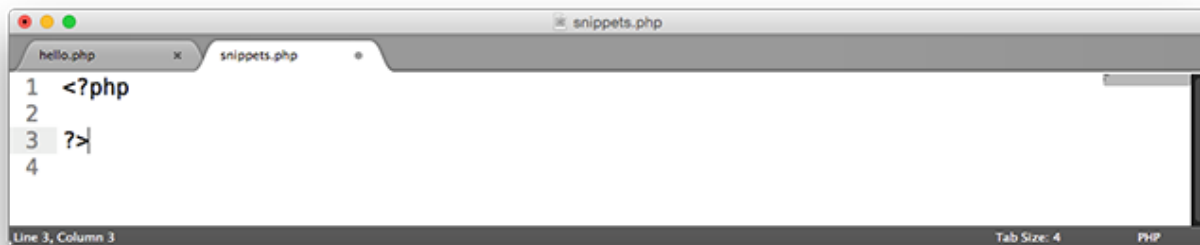


```
1 <!DOCTYPE HTML>
2 <html>
3 <head>
4     <meta charset="utf-8" />
5     <title>Title</title>
6 </head>
7 <body>
8     <h1>Heading</h1>
9 </body>
10 </html>
```

The screenshot shows a code editor window titled 'hello.php'. The code is a basic HTML document structure. The status bar at the bottom indicates 'Line 9, Column 8', 'Tab Size: 4', and 'PHP'.

Important: *Instead of saving this file with a .html or .htm extension, be sure to save it with a **.php** extension. In this example, I am naming the file: **hello.php***

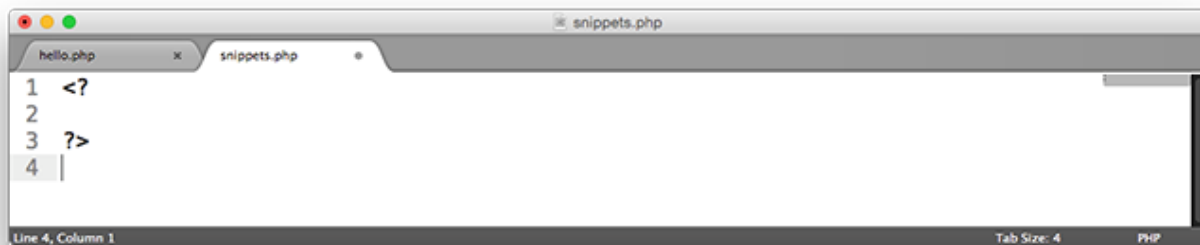
We are now going to start adding PHP code to our web page. To do this, we need to add starting and ending tags so the web server knows where the PHP is on the page so it can parse it and return HTML to the web browser. There are multiple ways to indicate the start and end of PHP code. The method we will use in this course looks like this:



```
1 <?php
2
3 ?>
4
```

The screenshot shows a code editor window titled 'snippets.php'. The code contains the short PHP opening and closing tags. The status bar at the bottom indicates 'Line 3, Column 3', 'Tab Size: 4', and 'PHP'.

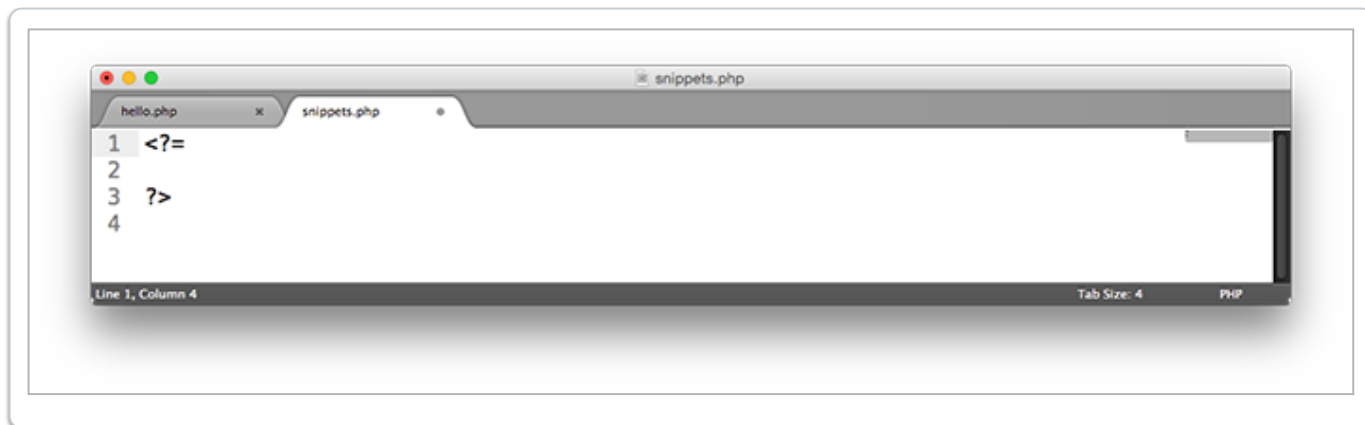
You may see other starting and ending tags for PHP that are known as *short-open* tags. Here are a few examples:



```
1 <?
2
3 ?>
4 |
```

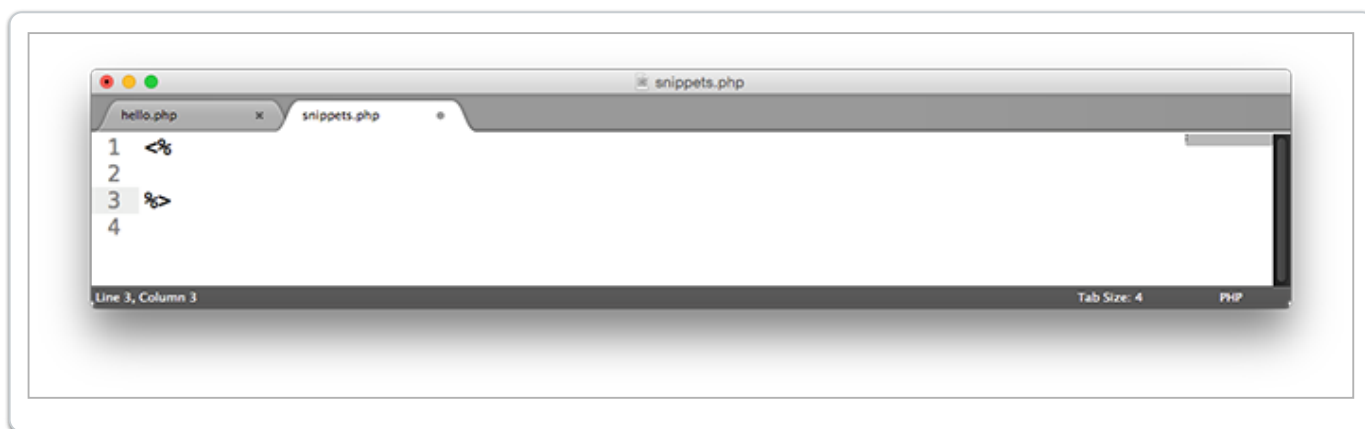
The screenshot shows a code editor window titled 'snippets.php'. The code contains the short PHP opening and closing tags. The status bar at the bottom indicates 'Line 4, Column 1', 'Tab Size: 4', and 'PHP'.

And even ...

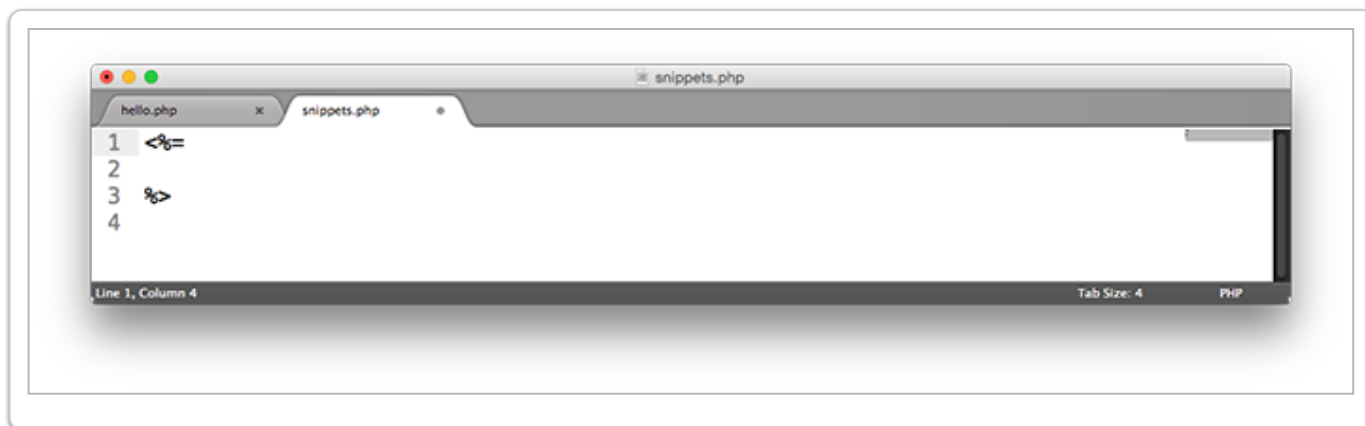


We won't be using *short-open* tags in this course as they are considered to be *bad form*.

You may also run into some PHP code that was written using ASP style-tags:



Or even ...



One major issue with using ASP style tags is that it requires a special setting to be enabled in the server's *php.ini* file. You can't count on other systems having this enabled so this greatly reduces the portability of your code. It is recommended that you don't use this style of opening and closing tags in your PHP projects.

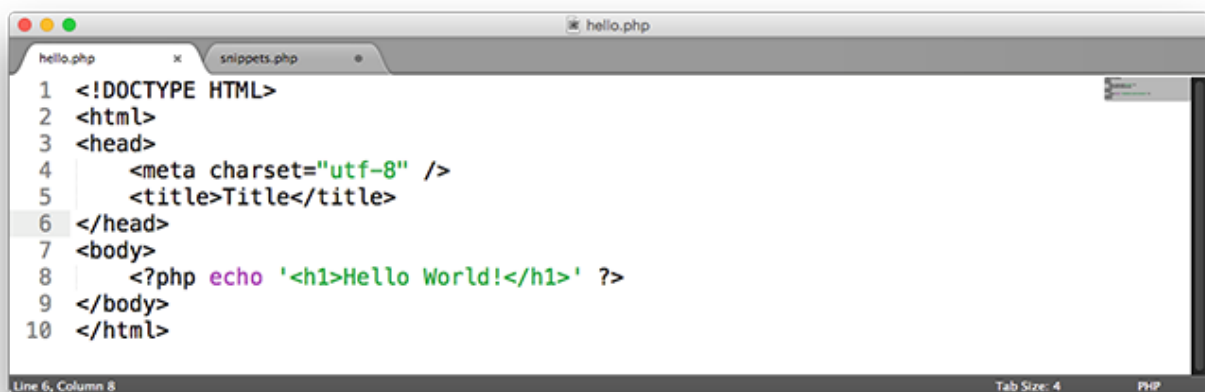
Let's look at the code we will use for our first PHP page, please continue to follow along in your text editor or IDE. Be sure to save the file to your web server's document root directory using the file manager feature in the hosting control panel or with a FTP client.



```
1 <!DOCTYPE HTML>
2 <html>
3 <head>
4     <meta charset="utf-8" />
5     <title>Title</title>
6 </head>
7 <body>
8     <h1><?php echo "Hello World!" ?></h1>
9 </body>
10 </html>
```

You can see that we used a classic introductory programming example for our first PHP page. The [echo command](#) is similar to *console.log* in JavaScript or the *print()* function in other languages. It simply echoes or prints the statement to standard output, which in this case is the web browser and we've wrapped it in a HTML `<h1>` tag.

TIP: Since the result of parsing PHP is HTML code, you can also insert HTML tags within your PHP code. In the previous example we could have put the opening and closing `<h1>` tags within the echo statement like this:



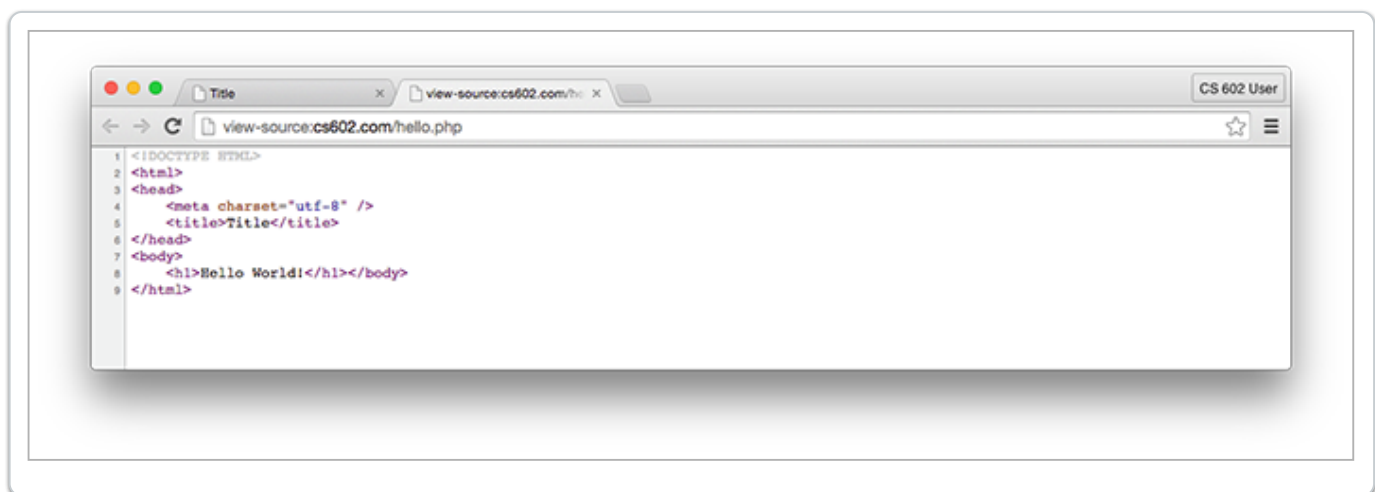
```
1 <!DOCTYPE HTML>
2 <html>
3 <head>
4     <meta charset="utf-8" />
5     <title>Title</title>
6 </head>
7 <body>
8     <?php echo '<h1>Hello World!</h1>' ?>
9 </body>
10 </html>
```

You may have also noticed that I switched from double quotes (`"`) to single quotes (`'`) in that last code sample. We'll discuss the differences in an upcoming section on *Strings*. For this example, the result is the same.

Here is how the code should render in your browser:



And this is what it should look like when you view the source for this page via the browser. You'll notice that you don't see any PHP code; instead, it just shows HTML:



Congratulations! If you have achieved the same result, you have successfully written your first PHP page!

Here are some important things to keep in mind:

- Whitespace doesn't matter. PHP ignores it.
- In most cases, every statement needs to end with a semi-colon.
- Web files with PHP code embedded within them must have a .php file extension.

PHP Comments

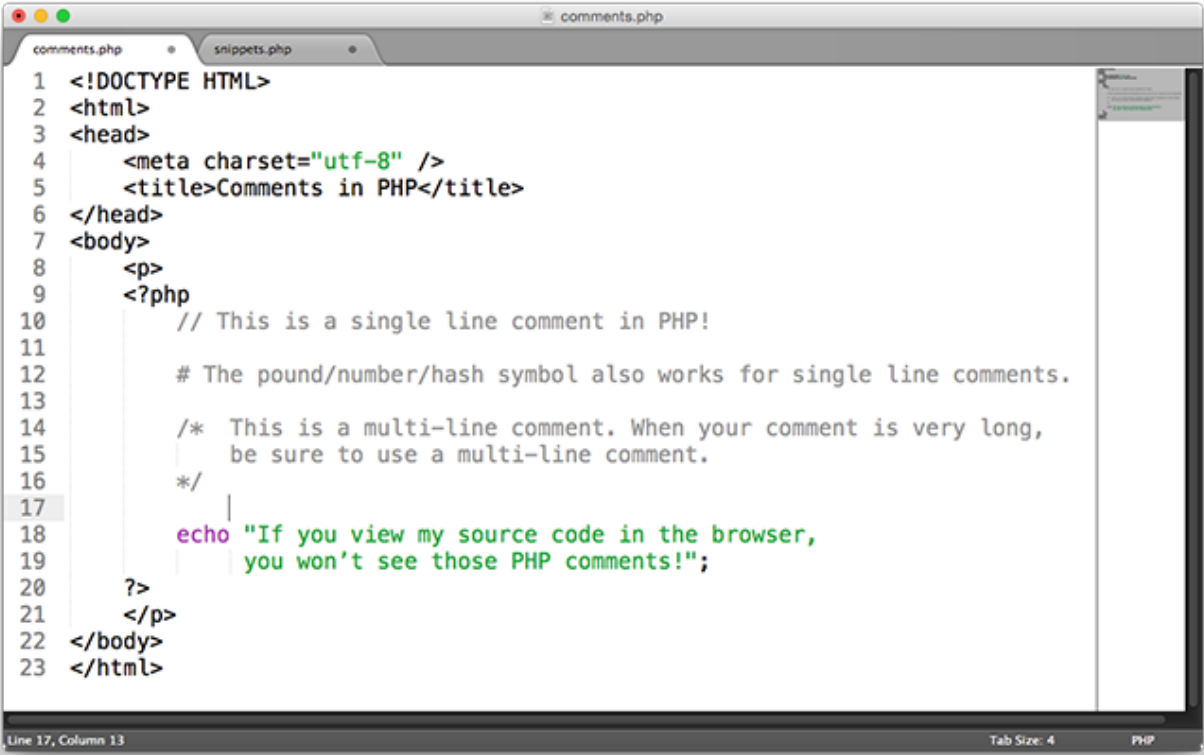
As with other programming languages, adding comments to your source code has many benefits. It lets you provide documentation for others and yourself that can be referred to in the future. This helps the next person looking at the source code understand what certain lines of code are intended to do and can also refresh your memory on your approach to solving a problem or implementing a solution.

IMPORTANT!

You will be required to provide comments within your programming assignments for any code that is not self-

documenting or explicitly clear.

Here are a few examples of using comments in PHP:



```
1 <!DOCTYPE HTML>
2 <html>
3 <head>
4     <meta charset="utf-8" />
5     <title>Comments in PHP</title>
6 </head>
7 <body>
8     <p>
9         <?php
10             // This is a single line comment in PHP!
11
12             # The pound/number/hash symbol also works for single line comments.
13
14             /* This is a multi-line comment. When your comment is very long,
15                be sure to use a multi-line comment.
16             */
17
18             echo "If you view my source code in the browser,
19                you won't see those PHP comments!";
20         ?>
21     </p>
22 </body>
23 </html>
```

Line 17, Column 13

Tab Size: 4 PHP

As you will see in the screenshots below, the comments do not render in the browser window or when you view the source via the web browser. This is a bit different from HTML and JavaScript comments where the comments don't render on the page but do show up when viewing the source code. This is an important distinction to remember.





The screenshot shows a web browser window with two tabs. The active tab is titled 'view-source:cs602.com/comments.php'. The address bar shows 'view-source:cs602.com/comments.php'. The page content is the source code of an HTML document. The code is as follows:

```
1 <!DOCTYPE HTML>
2 <html>
3 <head>
4   <meta charset="utf-8" />
5   <title>Comments in PHP</title>
6 </head>
7 <body>
8   <p>
9     If you view my source code in the browser,
10    you won't see those PHP comments!
11  </p>
12 </body>
13 </html>
```

Variables

Variables are a symbolic representation for a value that can be changed overtime. PHP is a loosely typed language, meaning you don't need to declare what type of data the variable will contain. This allows you to initialize a variable that stores a string and then change it to hold a number type such an integer or float later on. That's probably not a good idea, but it can be done.

Here are a couple of examples of declaring and initializing variables in PHP:



The screenshot shows a code editor window with a tab titled 'variables.php'. The code is as follows:

```
1 <!DOCTYPE HTML>
2 <html>
3 <head>
4   <meta charset="utf-8" />
5   <title>Variables</title>
6 </head>
7 <body>
8   <p>
9     <?php
10      $firstName = "Rasmus"; // String data type
11      $lastName = "Lerdorf"; // So is this
12      $numberOfCookies = 12; // Integer data type
13      $numberOfCookies = 11.5; // Now it is a float.
14      $someVar;
15      /* In the last example, we have declared a variable but did not
16         initialize it with a value. While you can do this, it is not
17         a good idea.
18      */
19    ?>
20   </p>
21 </body>
22 </html>
```

You can see that during initialization, we assigned values to the variables using the equals sign (=). This is known as the *assignment* operator in PHP. Whatever is on the right side of the equals sign gets assigned to the variable on the left side of the equals sign.

We will talk more about the data types available in PHP in an upcoming section.

Variable Names

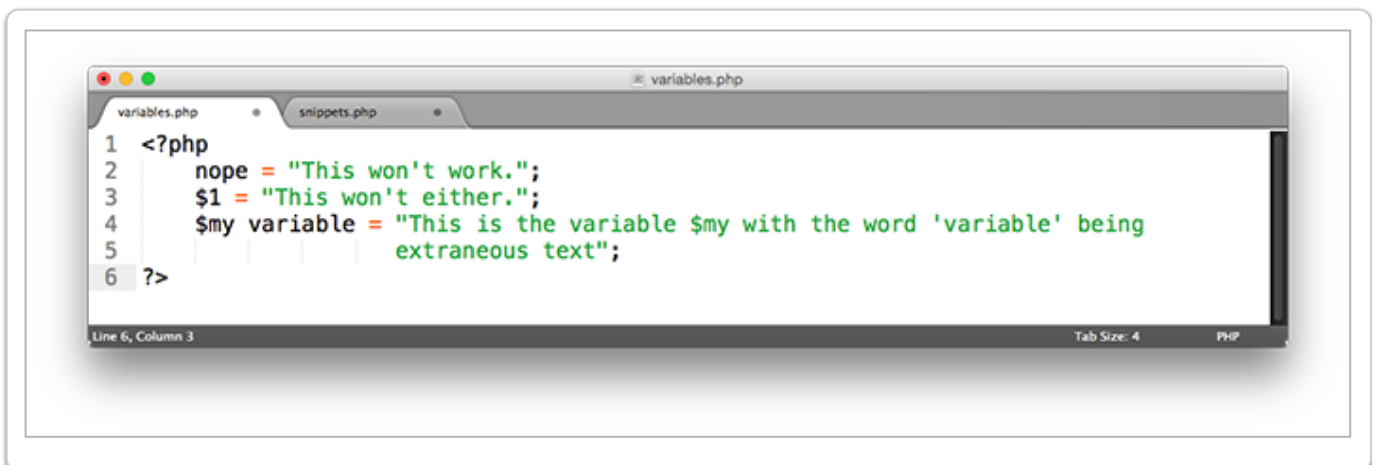
PHP has some pretty specific rules on naming variables:

- They must start with: \$
- The \$ can be followed with a letter or underscore
- They can contain letters, numbers, underscores, and dashes
- They can **not** contain spaces
 - For variable names with more than one word, it is recommended that you use camelCase notation or underscores to signify the start of a new word.
- They are case-sensitive

There are some names that are reserved by PHP and you can't use them for your own variable names. These are known as *reserved words*. You can view [all of the reserved words in PHP](#).

So here are some examples of *acceptable* variable names:

And here are a couple of examples of *unacceptable* variable names. If you use an advanced text editor or IDE that supports PHP syntax, you'll get an error/warning message and/or the syntax coloring will be not as expected.



Constants

As the name implies, the value of a constant can't change while the script is being executed. While constants are case-sensitive, the standard naming convention is to use ALL CAPS and to **not** prefix the name with a \$.

The syntax for initializing a constant is also different from initializing a variable. Let's take a look at an example:



Data Types

PHP supports the following data types:

- **String**
 - A sequence of characters such as: "I like PHP." They are typically contained in single quotes and double quotes, but there are other ways to represent them as well.
 - Single quoted strings will display most things literally, meaning as they are written. Variables are **not** interpreted within single quoted strings.
 - Double quoted strings do in fact interpret the variables contained within them.
- **Integer**
 - A whole number without a decimal point, like 10. They can also be represented in hex, octal, and binary notation. They can hold positive or negative values.
- **Floating point number (float or double)**
 - A number with precision or with a decimal point: 3.14. They can be represented in scientific notation as well. As always, floating-point numbers have limited precision. You may get some unexpected results depending on how they are used. These too can be positive or negative in value.
- **Boolean**
 - These represent TRUE or FALSE. They can be represented as a literal and are not case-sensitive.
- **Array**
 - We'll discuss this in a section dedicated to Arrays
- **Object**
 - We'll discuss objects in the next module.
- **NULL**
 - This is a special variable that represents a variable with no value. It is not case sensitive.
- **Resource**
 - This is another special variable that holds a reference to an external source. They are used by special functions. The use of this type of variable will likely be limited in this course.

You can read more about [the eight primitive data types available in PHP](#).

Arrays

An array is a *data structure* that can store multiple values in a single variable. These multiple values are referred to as elements and each element is identified by an index or key. You'll often hear the term "key-value" pair. This refers to the matching sets of the key position and the value contained at that spot in the array. You can think of an array as a type of list.

Note: A data structure is a particular format for storing and organizing data.

There are three different types of arrays in PHP:

- **Indexed arrays**
 - Sometimes referred to as a numeric array, but this doesn't mean that it only stores numbers. It means that the indices of the array are a number and values are stored and accessed in a linear manner. They can store anything like numbers, strings, or objects. The default array index starts at zero.
- **Associative arrays**
 - An associative array uses a string as the index. In this case, the index is known as a named key.
- **Multidimensional arrays**
 - This is an array that contains other arrays

We are going to share a code example for each type of array here and we will use arrays a bit more when we talk about *foreach* loops later on in this lecture.

Indexed Array Examples

.

Here is what is displayed in the browser as a result:

.

Associative Array Examples

.

Here is what is displayed in the browser as a result:

```
.
```

Multidimensional Array Examples

```
.
```

Here is what is displayed in the browser as a result:

```
.
```

You can see that in the multi-dimensional array example, it was a bit tricky to keep track of all of the pieces, so the formatting used in the example above makes things a bit more readable. Finally, it was a bit tedious to print out all of the information for all three courses. We'll learn a more efficient way to do this when we learn about loops later on in this lecture.

Type Juggling

As we discussed in a previous section, PHP is a loosely typed language. We don't have to specify our data type when declaring our variables, but this doesn't mean that the variable does not have a data type. The data type of a variable is determined by the contents stored within the variable. One example of this is to create a variable and assign a String value to it and then change the value to an integer with a new assignment statement. PHP automatically changes the data type of the variable.

In some cases, we might end up converting the type of a variable without using the assignment operator to give it a new value. This can happen through type casting and type juggling. We'll talk about type casting in the next section. In this section, we will focus in on type juggling.

Type juggling is PHP's automatic way of doing type conversion. There are a few cases in which this will happen. Here is one example:

```
.
```

This is what we will see when the code is executed and viewed in the browser:

```
.
```

You can see that we didn't get an error when we tried to add a String and an integer value together. Instead, PHP converted the String to an integer and returned the result, which was an integer.

Let's take a look at another example.

```
.
```

This is what we will see when the code is executed and viewed in the browser:

```
.
```

In the previous example, we introduced a special String operator known as the *concatenation operator*. This is represented as a period (.) in PHP. This returns the concatenation of its right and left operator. In other words, it joins or adds them together.

So we concatenated a String with a float and then concatenated all of that with another String. PHP converted the float to a String, so the entire thing resulted in a String. We used a PHP function named *gettype()* to return the type of the variable. We'll learn more about functions later on in this lecture.

A simple way to summarize type juggling is that when PHP does automatic type conversion, it has been juggled. You shouldn't rely on PHP to do a lot of type juggling as it is considered to be a sloppy practice.

In the next section we will discuss how we can explicitly convert variable types on our own volition.

Type Casting

In the previous section, we demonstrated how PHP can automatically convert variables to a different type when needed. In this section, we will explore how we can do this ourselves with type casting.

There are two ways to do type casting in PHP. Here is an example that shows both methods:

```
.
```

Here is the result:

```
.
```

Let's discuss what happened here. On line 12 we put (float) in front of \$price during the operation. When the statement is evaluated and returns the value, we also echo out the data type, which is a *double*. A double is the same thing as a float in PHP. The way in which we did this lets us convert it while it is being used, but without changing or mutating the variable type. You see that after we output \$price back to the screen and get it's type.

You might be asking yourself: *But what if we wanted to change the type permanently?* PHP will allow us to do that with the *settype()* function. Here is how it works:

```
.
```


And our result is:

In this example, we used the `settype()` function to actually convert `$price` from a String to a double. This is confirmed when we call `gettype()` on `$price` after evaluating the statement that subtracts the two values.

There are [other aspects of type juggling and type casting](#) to consider.

Arithmetic, Assignment, and Incrementing Operators

PHP has a number of different operator groups. An operator is used to perform operations on operands in the form of variables and values.

Arithmetic Operators

- **+**
 - Adds two operands
 - Ex. $2 + 2 = 4$
- **-**
 - Subtracts the operand on the right from the operand on the left
 - Ex. $4 - 2 = 2$
- *****
 - Multiplies two operands
 - Ex. $5 * 5 = 25$
- **/**
 - Divides operand on the left by the operand on the right
 - Ex. $9 / 3 = 3$
- **%**
 - Returns the remainder after division (modulus operator)
 - Ex. $5 \% 2 = 1$
- ******
 - Raises the power of the operand on the left to the power of the operand on the right
 - Ex. $3 ** 3 = 27$

Assignment Operators

- **=**
 - Assignment: The variable on the left gets the value of the expression on the right

- **+=**
 - Adds the value on the left to the value on the right and assigns the result back to the value on the left
 - Ex. `x += y` is the same as `x = x + y`
- **-=**
 - Subtracts the value on the right from the value on the left and assigns the result back to the value on the left
 - Ex. `x -= y` is the same as `x = x - y`
- ***=**
 - Multiplies the value on the left with the value on the right and assigns the result back to the value on the left
 - Ex. `x *= y` is the same as `x = x * y`
- **/=**
 - Divides the value on the right into the value on the left and assigns the result back to the value on the left
 - Ex. `x /= y` is the same as `x = x / y`
- **%=**
 - Value on the left % value on the right and assigns the result back to the value on the left
 - Ex. `x %= y` is the same as `x = x % y`

Incrementing and Decrementing Operators

- **++**
 - Pre-increment: used in front of the value or variable (`++$var`)
 - Increments value by one, then returns the result
 - Post-increment: used behind the value or variable (`$var++`)
 - Returns the result, then increments the value by one
- **--**
 - Pre-decrement: used in front of the value or variable (`--$var`)
 - Decrements value by one, then returns the result
 - Post-decrement: used behind the value or variable (`$var--`)
 - Returns the result, then decrements the value by one

Comparison and Logical Operators

Before we move on to the next section, it will be helpful to list the comparison and logical operators that are available in PHP. We will learn how to use these in the next couple of sections that discuss program flow control.

Comparison Operators

- **== : Equal**

- This is **not** assignment, it is equality. Think of this as asking the question: *Is equal to?*
- **=== : Identical**
 - Like equality but it checks the value *and* the data type. Equality only checks the value.
- **> : Greater than**
 - Returns true if the value on the left is greater than the value on the right.
- **< : Less than**
 - Returns true if the value on the right is greater than the value on the left.
- **>= : Greater than or equal**
 - Returns true if the value on the left is greater than or equal to the value on the right.
- **<= : Less than or equal**
 - Returns true if the value on the right is greater than or equal to the value on the left.
- **<> : Not equal**
 - Returns true if the values on the left and right are **not** equal to each other.
- **!= : Not equal**
 - Same as <>, but != is more commonly used.
- **!== : Not identical**
 - Returns true if the values on the left and right are **not** identical.

Logical Operators

- **&& : And**
 - You can also use the word **and** in addition to &&. Returns TRUE if the value on the left *and* the value on the right are TRUE. There is *operator precedence* between && and **and**. ([Operator Precedence](#))
- **|| : Or**
 - You can also use the word **or** in addition to ||. Those two vertical lines are the pipe character on your keyboard. They must be typed without a space in between the two. Returns TRUE if the value on the left *or* the value on the right are TRUE. There is operator precedence between || and **or**. ([Operator Precedence](#))
- **xor : Xor**
 - Returns TRUE if either the value on the left or the right is TRUE, but **not** both.
- **! : Not**
 - Returns TRUE if a value is **not** TRUE. It makes it the opposite of it's current value.

Conditionals

Conditionals are a group of control structures in PHP. They help determine the flow of the program based on asking a question and getting an answer. Different code blocks are executed depending on the answer to that question. The types of questions we will be using are in the form of a Boolean expression, which evaluates to TRUE or FALSE.

if Statements

The basic format of an *if* statement looks like this:

```
.
```

In the example listed above, *if* is our keyword (a language construct), everything inside of our opening and closing parenthesis is the Boolean expression that **must** evaluate to TRUE or FALSE. We accomplish this through a variety of comparison and/or logical operators.

We then start our code block with an opening curly brace. The opening curly brace starts a block of code. We then write code for whatever statements we would like to use and then close the block of code with a closing curly brace. Let's take a look at a real example:

```
.
```

If we were to view this in our browser, we would get a blank page. The reason for this is that we compared the value of \$age to the integer 18. The comparison we made was essentially asking the question, is \$age greater than or equal to 18? We know that \$age has a value of 17, so we can change this question to: Is 17 greater or equal to 18? The answer is no, or in this case *FALSE*. Because the answer is FALSE, nothing inside of the code block is evaluated, therefore, nothing prints out to the screen. If the evaluation of the Boolean expression resulted in an answer of TRUE, the code block would have been executed and we would see this on the screen:

```
.
```

else Statements

As you saw above in the example using an *if* statement, we asked one question and based on the result of that question, we only executed a statement if it was TRUE. You will often need to execute one statement if a certain condition is TRUE and a different statement if the condition is FALSE. We can do this with an *else* statement. Let's modify the example above to demonstrate this concept:

```
.
```

When we view this in the browser, this is what we will see:

```
.
```

As you can see, what it really boils down to is that else is used to execute an alternative or final statement if the conditional expression returns FALSE.

elseif Statements

An *elseif* statement is used to string together multiple conditional statements and the resulting actions that should be taken based on a TRUE value. As you can see by the name, it is a combination of *if* and *else*. You can have several *elseif* statements within an if statement. Only the first *elseif* statement that evaluates to TRUE will be executed and only if the if statement evaluates to false.

Let's look at an example:

```
.

```

Here we are checking to see if someone is at least 18 years old and if they are a citizen. We used a Boolean value for citizen and check that value in our *if* and *elseif* statements. We use the *else* statement as a final expression that gets evaluated if both the *if* and *elseif* conditions both prove to be FALSE.

Here is what we will see in the browser:

```
.

```

switch Statements

Similar to an *if* statement, a *switch* statement is used to perform different sets of actions based on the result of a condition. A *switch* statement is useful when you want compare a variable or expression with many different values and then execute a particular piece of code that is associated with a specific value.

It is important to note that with a *switch* statement condition, we are testing for equality. We are not making a comparison or checking for a Boolean value.

The basic format of *switch* statement looks like this:

```
.

```

It will probably be better understood with an example:

```
.

```

So how does this actually work? The *switch* command evaluates the expression within the set of parenthesis, which is the variable \$school. The value of that variable (or an expression) is compared against the values for each case listed in the code block that follows. If there is a match, the code within that case is executed. A *break* statement is included to prevent the code from continuing on to the next case automatically. If you do not include the *break* statement,

everything after the matching case will be executed. This is probably not what you want to have happen, but there might be some cases where that is the desired outcome. Just remember that the *break* statement is optional.

Finally, we have a *default* statement (also optional) that is executed if no match is found with the *case* statements listed above.

Here is what is displayed in the browser as a result:

.

Loops

Loops are another type of control structure in PHP. They are similar to conditional statements in the way that they execute a block of code when the specified condition is TRUE. However, where they truly come to benefit us is in the fact that they allow us to run the same block of code over and over again until a condition changes to FALSE. There are a few different types of looping mechanisms in PHP and we'll discuss them in further detail below.

IMPORTANT! - *It is very important that the condition being evaluated in a loop eventually change to FALSE in order to prevent an infinite loop. An infinite loop is one that does not stop looping.*

while Loop

The while loop is the simplest of all loop types available in PHP. As long as the condition within the while loop is true, the statements within the code block will execute repeatedly. The value of the condition gets checked each time at the start of the loop. We need to increment the value of the condition every time that the contents of the loop are executed.

Here is the basic structure of a while loop:

.

Here is an example:

.

Here is what is displayed in the browser as a result:

.

do-while Loop

A do-while loop is essentially the same as a while loop with the difference being that the condition is checked at the end of the loop iteration, not at the beginning as in the case with the while loop. This ensures that a do-while loop is going to run at least one time. That's not the case for the while loop.

Here is the basic structure of a do-while loop:

```
do {  
    // code to be executed  
} while (condition);
```

Here is an example:

```
do {  
    console.log(5);  
} while (false);
```

Here is what is displayed in the browser as a result:

```
5
```

As you can see in the example above, the loop printed out the number 5 even though the condition wasn't true, because the condition does not get checked until the loop executes at least one time.

for Loop

For loops are very common in many programming languages. They are also similar to while loops, but we have a lot more control over how they behave.

Here is the basic structure of a for loop:

```
for (initialize; condition; increment) {  
    // code to be executed  
}
```

Here is an example:

```
for (let i = 0; i < 5; i++) {  
    console.log(i);  
}
```

Here is what is displayed in the browser as a result:

```
0  
1  
2  
3  
4
```

As you can see, we have a lot more flexibility with a for loop and its code is a lot cleaner. However, it might look a bit confusing at first, so let's explain the parts.

Looking back at the basic structure of the for loop, you'll see the keyword `for` and then three separate expressions within the set of parenthesis. Here is what each expression does:

- initialize `expr`;

- This sets the starting value of a counter variable. We did this outside of the loop with while and do-while loops. *Take note of the semi-colon at the end of the expression.*
- condition expr;
 - This is the condition we are checking for to see if it is true or false. *Take note of the semi-colon at the end of the expression.*
- each expr
 - This is what happens at the end of each loop iteration. This is a way to increment the loop to ensure the value of what we are checking against does in fact get changed. Notice that there is **no** semi-colon at the end of the expression. Instead of incrementing the counter variable by one as we did in previous examples with the ++ increment operator, we multiplied the value by 2 and assigned the result back to the value itself using the shorthand *= operator.

foreach Loop

The foreach loop is very different from the other loops we have covered up to this point. It is specifically used to loop through the contents of an array. It is not checking for a condition to be TRUE or FALSE and there is no test to see when it should quit. It knows when it is finished processing the contents of the array and stops when it reaches the end.

Here is the basic structure of a foreach loop:

```

foreach ($array) {
    // do something
}
  
```

In the code structure above, \$array is the name of the array that we want to iterate through. Each item in the array is assigned to \$value (you can call it anything, \$value references the spot of the array it is in). as is a PHP keyword.

Let's take a look at an example of using a foreach loop on an indexed array:

```

<pre>
foreach ($array) {
    // do something
}
  
```

Here is what is displayed in the browser as a result:

```

<pre>
foreach ($array) {
    // do something
}
  
```

Using a foreach loop with an associative array is a bit different because we have to account for the key and the value. Here is what the basic structure looks like:

```

foreach ($array as $key => $value) {
    // do something
}
  
```

Let's jump right into an example:

```

<pre>
foreach ($array) {
    // do something
}
  
```


From the standpoint of iterating through the associative array, it isn't much different than an indexed array. We just need to account for the key and value for each element. In the example above, `$attribute` represents the key and `$value` represents the value of each array element.

Inside of the code block for the loop, we introduce something new, *functions*. Don't be too concerned with how the two functions work at this point. In the browser screenshot below, you will see that we cleaned up the key (`$attribute`) names by uppercasing the words and replacing the underscores with spaces. These two functions, `ucwords` and `str_replace` are functions that PHP provides us with. We'll learn more about PHP provided functions in an upcoming section of this lecture.

Here is what is displayed in the browser as a result:



Continue and Break

There are two special commands that we can include in our loops:

- *continue*
 - Used inside of a loop to skip the rest of the current iteration and to go immediately to the conditional value that starts the next iteration. It essentially tells the loop to loop again earlier than it normally would.
- *break*
 - Allows us to break completely out of the entire loop. It stops the entire process.

You can learn more about break and continue and see examples of how they are use:

- [continue](#)
- [break](#)

PHP Functions

In many cases, we will write code that we want to use multiple times or in different places. Functions are simply a name we give to a block of code. These are not new to PHP; they are available in many programming languages.

There are two broad categories of functions.

- Built-in or language provided
 - These functions are provided by PHP. PHP has over 1000 predefined functions that we can use in our programs. We used two of these built-in functions in the previous section, do you remember? They were `ucwords` and `str_replace`.
- User-defined
 - We, the programmers, create these functions. We can wrap a code block of any type in a function and then use the function as we need to. It saves a great deal of time and can help reduce troubleshooting

time.

You will end up using a lot of functions in your programs, both built-in and user-defined.

Find [a listing of *all* PHP functions and methods](#).

We haven't talked about methods yet, but we will cover them in the next module. A method is actually a function, but it is just used in the context of classes/objects.

There are built-in functions to handle arrays, strings, date/time, error handling, file system interactions, network functions, and so much more.

The listing of all PHP functions provided above can be overwhelming, but you can search the PHP website for specific functions or functionality. You can also browse through various categories if you are not quite sure what you are looking for, a categorized list is available at [Function Reference](#).

We are going to cover user-defined functions in the next section.

User-defined Functions

In this section, we are going to cover creating (declaring) our own functions and calling them when we need to use them.

There are a few things to know when creating our own functions in PHP. We start by giving the function a name. With a name, we will be able to use (call) the function. With that in mind, be sure to name the function in a way that is easy to remember and type. The syntax used to create a function is fairly simple. You tell PHP that you want to create a function by using a keyword: function and then provide the name that you would like to assign to the function. The naming rules for functions are pretty much the same as PHP variables, but function names don't need to be preceded with a dollar sign. A function can also take parameters, also called arguments, and we'll cover these soon.

Let's take a look at the basic structure of a function declaration:

```
.
```

Notice the use of the function keyword. We then have our function name and a set of opening and closing parenthesis. We'll take more about the parenthesis when we cover parameters a bit later. Finally we have an area for our code that will be contained within the function within a set of opening and closing curly braces.

Now lets jump into an example and create our first function:

```
.
```

We just declared our first function. It's that simple. One important point: This program doesn't actually do anything just yet because we have only declared the function. We haven't called it yet. It is like declaring a variable and not using it. If we were to view the program now in a browser, we would just see a blank page.

Let's tack on some more code to this example:

```
.
```

We've now called the function in addition to declaring it, so let's take a look at what happens when we load the page in a browser:

```
.
```

No surprises there. The function is pretty static, meaning it just prints out a line of code contained within it and we can't change its behavior to be more dynamic when we call the function. We can change that by allowing the function to accept one or more parameters or arguments. Information can be passed into a function and the function can process this information. A parameter is just like a variable.

Let's jump right into an example where we improve upon our `welcomeUser()` function and allow it to take arguments.

```
.
```

Let's take a look and see what this does:

```
.
```

You can see that we just end up using the values we pass into functions as variables within the body of the function itself. Functions can accept a number of different data types and data structures as arguments. Functions can even accept other functions as arguments. You can create very simple or very complex functions using what you have learned in this section.

Returning Values

In the examples we shared above, the functions didn't return a value. It is often helpful for a function to return a value. One purpose of this is to use the value that is returned by the function and assign it to a variable that can be used later on. Here is an example of returning a value from a function:

```
.
```

And here is the result:

```
.
```

You have now learned a great deal about the basics of PHP. In the next section, we will learn about MySQL in order to be prepared to use PHP with a database.

