

## 3.5: Data Storage

### Variables

Variables are created and used when our script runs in order to create references to (store copies of) pieces of information that are needed for a short span of time, or are expected to change value as the script runs. They contain a reference to a place in the server's memory where the value is stored, not the actual content. This tells the parser where to look to find the information we want. PHP is a loosely typed language, which means we do not have to tell the computer what type of information we are going to keep in a variable. In strictly typed languages, we would have to specify if the variable was going to be an integer, string, float, or other option supported by the language in use. Trying to store a different type of information than declared in a variable would result in an error. PHP will not give us an error or check this by default. This eliminates the need to declare variables, but this can result in some confusing bugs in your code. Those of you with experience in a strictly typed language like C++ will be happy to note that although not required, PHP will allow declarations, and will then give errors upon their misuse.

Variables in PHP must start with a dollar sign (\$), and can be followed by an underscore (\_) or letter (a through z, both upper and lower case). Variables cannot start with a number, but may contain numbers after an underscore or at least one letter; they also cannot contain a space, as spaces are used to determine where commands, variables, and other elements start and end. See the table below for examples:

Table 3.5.1 PHP Variable Naming

GOOD	BAD
<code>\$_first</code>	<code>\$1st</code>
<code>\$LastName</code>	<code>\$(first)name</code>
<code>\$standard_tax_rate</code>	<code>\$first name</code>
<code>\$last4SSN</code>	<code>\$final\$</code>

- [Booleans](#) Can have a value of 0 or 1
- [Integers](#) Whole numbers (1, 3, 20, etc.)
- [Floating point numbers](#) Decimal values (1.33, 34.2325)
- [Strings](#) Contain any number of characters
- [Arrays](#) Structured lists of information
- [Objects](#) Collections of related variables and functions
- [Resources](#) Special variables that hold reference points to things like files
- [NULL](#) An empty (unused or unset) variable
- [Callbacks](#) A mechanism to reference a function declared elsewhere

In the scope of this text we will cover most of these items with exception to callbacks, and with only a cursory examination of objects. If you plan to focus on application development, this would be a good area to continue studying. For most web development, there is little call for robust object oriented programming.

To create a variable in PHP, we first give the name we wish to use, followed by what we want to assign to the variable. In PHP, the equal sign (=) is used to assign what is on the right hand side to what is on the left hand side (we will see how to check login statements later on). To create a variable called `ourString` with the value of Hello World, we would enter the following:

```
1. $ourString = 'Hello World';
```

We can now refer to `$ourString` in one or more places of our code in order to access the string Hello World and use it or modify it.

You might notice the semi-colon (;) at the end of the line. Semi-colons are used to tell the interpreter where a statement ends, and where the next one begins. They are easy to forget! If you see a syntax error, you will want to look at the line before the error to see if a semi-colon is missing.

PHP also maintains some variables of its own, called predefined variables. These start with underscores, and will hold certain types of values for us (avoiding the use of underscores at the start of your variables will help avoid colliding with these reserved

variables). Several of these variables (\$\_GET, \$\_POST and \$\_FILES) will hold items a user has typed or submitted using forms. \$\_COOKIE and \$\_SESSION are used to hold information throughout a user's visit, and \$\_ENV holds information about the server.

### Incrementing Methods

Until now, we have been using the equal sign strictly for assigning strings to variables. We can also use some short hand methods of modifying numerical variables within our code. For example, adding 1 to the variable \$counter can be done with:

1. `$counter = $counter + 1;`

But we can shorten that by using the “plus equal”:

1. `$counter +=1;`

Or if we only need to add 1, the “plus plus”:

1. `$counter++;`

In these example, each one would add 1 to the counter. In the first two examples, we could add more than one, or perform a calculation to be added. We can also perform the opposite calculation, in that we can subtract and assign a given number as well by using `--` or `--`. Where and how you choose to embrace these is up to you, but you should be familiar with each form in order to fully understand any code you are examining.

### Strings

In our first echo examples, we printed a string to the screen. Strings are a type of variable that hold, as seems obvious, strings of words. Full sentences can be stored in one variable name, or can be built by combining other variables. Manipulating strings in PHP can be done through a number of functions, which can complete tasks like finding and replacing words, breaking strings apart, capitalizing one or all words, and a number of other useful tasks.

Strings can be used to create output that the user reads, generate part or all of the HTML code for a page to display, and even commands that can be passed to other languages to direct their operation.

### Single, Double Quotes

Until now, when we have used strings, they have been double quoted. PHP actually lets us use both single and double quotes when defining our strings to support different functions. There is an important difference between the two that we need to keep in mind. Single quoted strings are treated by the interpreter as plain text, meaning the output to the screen will be exactly what is in quotes. Double quoted strings will be examined by the interpreter for anything that can be processed by PHP, which means items like special characters and variables will be replaced with what they represent before the output is sent to the screen. For example, if we set a variable named string to Hello and use it in our output, single quotes will ignore it while double quotes will process it:

1. `$string = "Hello";` // The quotes we use here do not matter for this example
2. `echo "$string there";`
3. `echo '$string there';`
4. Hello there
5. \$string there

### Escaping

Escape characters are symbols with special, secondary meaning when coupled with the language's escaping method. We can indicate that we want a new line in a text file by using `\n`. In this example, n is not “just an n,” it represents that we want a newline because it is preceded by a backslash, PHP's escaping character. In PHP, escape characters are commonly used in double-quoted strings, so we can include special characters (single quoted string will ignore this, just like other PHP commands, variables, and the like).

A helpful way to think about the escape character is that it “reverses” the character or symbol that comes after it. If it precedes a letter, then it is supposed to do something, not display the letter. If it precedes a symbol, the symbol has a special value in PHP, but we actually want to see the character.

Table 3.5.2 Character Escaping

--	--

<code>\"</code>	Print the double quote, not use it as a string opening or closing marker
<code>\'</code>	Print the single quote, not use it as a string opening or closing marker
<code>\n</code>	Print a new line character (for text or output files, not on the screen)
<code>\t</code>	Print a tab character
<code>\r</code>	Print a carriage return (for text or output files, not on the screen)
<code>\\$</code>	Print the next character as a dollar sign, not as part of a variable
<code>\\</code>	Print the next character as a backslash, not an escape character

Writing `$string = "I want to spend $5.00";` would result in a name error. Instead, we can use `$string= "I want to spend \$5.00";` to achieve the output we are looking for. If we wanted to use the backslash, we could write a folder location as `$address = "c:\\www\\ourfolder\\sometext.txt";`. While we could more easily do this with single quotes as `$address = 'c:\\www\\ourfolder\\sometext.txt';` we would need to append any variables we wanted to reference into the string that is single quoted.

## Constants

Sometimes we need to store a piece of information, but we do not want it to change while our script is running. To do this, we can create a constant—essentially, a variable that we are not allowed to change. Creating a constant is done by calling the `define` function, and passing a string of our constant's name and its contents:

```
1. define("OURCONSTANT", "Our constant value");
```

You will notice we do not have a dollar sign in front of our constant name. In fact, to use it, we can just echo the name:

```
1. echo OURCONSTANT;
```

Our example here has the constant name all uppercase. This is a practice many people use to help distinguish between variables and constants. There are also some predefined constants in PHP that can be useful to us, such as `PHP_VERSION` and `PHP_OS`. The former will give us the version number for PHP, and the latter will give us details on the operating system the server is running on. Since constants do not have a leading dollar sign, we cannot embed them in a string, but instead need to concatenate them if we want to use them with other output:

```
1. echo " This server runs on " . PHP_OS;
```

Concatenation is the act of connecting several items into one variable or output. A period is used as we did above to denote where those pieces start and end. If we wanted to add more to our statement, we can keep adding periods like this:

```
1. echo "This server runs on " . PHP_OS . " and use PHP version " . PHP_VERSION;
```

## Arrays

Arrays are a much dreaded topic to many programmers, and almost as frustrating as a missing terminating character. For the uninitiated, an array is a method of storing multiple values under one variable name as a linked list of information. This allows us to keep related values together, and to establish relationships between data. Typically, array information is stored in one of two different formats, either numeric or associative. In numerical format, each element in the list (or, each piece of information) is found by referring to its place in line. Depending on your programming language, counting may start at 1 or 0. In our case, by default, PHP starts with 0. In order to take a look at an array, we should follow best practices and declare our variable as an empty one to begin with. We can do this with:

```
1. $ourFirstArray = array();
```

Now the system will know that we intend to use this variable as an array. If we try to echo or print our array, we would see the following output:

### 1. Array

In order to see the contents of an array, we need to refer to an actual position or view the entire contents. Since we have not added anything yet, we will move along for now.

Here we will create a new array, but one in which we already know what we want the first few values to be. We will set up the cast of Family Guy as an array called theGriffins.

```
1. $theGriffins = array("Peter", "Lois", "Stewie", "Chris", "Brian");
```

Now we can take a look at some output. If we wanted to see what the first element of the array held, we could:

```
1. echo $theGriffins[0];
```

which would give us:

```
1. Peter
```

Or, to take a quick look at the entire array, we can use the built in function `print_r`, which means print recursively, and will output each value for us in a preformatted manner:

```
1. print_r($theGriffins);
2. Array(
3. 0: Peter
4. 1: Lois
5. 2: Stewie
6. 3: Chris
7. 4: Brian
8. )
```

Now, something seems amiss. Is someone missing? Oh yes, Meg. Let's add her to our array. To add a new element to the end of an array, we do not need to worry about knowing how long it is, or what number to assign to the new element. With PHP we can simply append [] to our variable name, adding Meg as a new element at the end of our array:

```
1. $theGriffins[]='Meg';
```

Now if we run `print_r`, we would see:

```
1. Array(
2. 0: Peter
3. 1: Lois
4. 2: Stewie
5. 3: Chris
6. 4: Brian
7. 5: Meg
8. )
```

Perhaps we want to make things a little more formal, and use full first names. In this case, we need to update a few things. First, we should change Stewie to Stewart. Since we have the reference right above this text we can see that Stewie is at position 2 (item 3) in the array. So let us set that position to his full name:

```
1. $theGriffins[2]='Stewart';
```

Your `print $theGriffins[2];` should now give you Stewart instead of Stewie! By placing the item's position number in the brackets of our array variable, we are specifying that we want to see the information that is stored in that spot. Perhaps you have forgotten where in the array you stored a particular value. Most languages supporting arrays will already have built in functions for common tasks such as this. In PHP, we can use the `array_search` function. In this case, we pass the values as a "needle in a haystack" pair, giving the function first what we are looking for, and then the array in which we hope to find it:

```
1. echo array_search("Meg", $theGriffins);
```

will give us:

```
1. 4
```

Note that close matches would be ignored. The interpreter does not know that Pete and Peter, or Meg and Megan represent the same common name. For these types of searches, we would need much more complex algorithms.

In order to update the Meg value to Megan, we will combine our techniques:

1. `$location = array_search("Meg", $theGriffins);`
2. `$theGriffins[$location] = 'Megan';`

We could, for the sake of brevity, take advantage of the inner first nature of the PHP interpreter and combine our statements:

1. `$theGriffins[array_search("Meg", $theGriffins)]= 'Megan';`

Now that we are a bit more comfortable with numbered arrays, we will take a look at associative. In this approach, we provide the reference in which we want a position in the array to be named. For instance, perhaps we want to give short descriptions of each character so someone unfamiliar with the show is better able to recognize them. To distinguish details by character, we will use their names in place of numbers. Our initial array from before with names and descriptions could look as follows:

1. `$theGriffins = array("Peter"=>"The fat guy", "Lois"=>"The red head", "Stewie"=>"The baby", "Chris"=>"The awkward boy", "Brian"=>"The Dog");`

Now that our array is associative, we pass the identifying piece of information we are looking for. This is done as a key and value pair, where the key is the associative word you can reference and the values is still what is stored. You will notice we used `=>` in our declaration this time, which identifies what comes before the `=>` as the key, and what follows as the value. So to find out what we know about Lois:

1. `print $theGriffins['lois'];`

gives us:

1. The red head

Note that we need to put the associative key in quotes (single or double) when using print or echo.

## Reading Get, Post

Earlier we discussed how to set a form to use `Get` or `Post` to transmit data to be processed. To access these pieces of information, PHP has two built in arrays that store what is transmitted under `$_POST` and `$_GET`. These are reserved variables that are always available in your code whether or not any information was sent in that manner (in which case, the variable will simply be an empty array). When we want to see the value of a form element sent using `Get` with a field name of `firstName`, we would use:

1. `print $_GET['firstName'];`

If it was sent using post, all we would change is the variable name:

1. `print $_POST['firstName'];`

To save changes to the variable, we can place the results of the desired change to a local variable we create, or assign them back to the array position in our `GET` or `POST` array. The only way of keeping the changed material for use on another page, though, is to resubmit the data to the page. I recommend using local variables so this is easier to keep in mind.

Note that when you use `GET` or `POST` variables inside of a double quoted string, the single quote characters are not needed in the array element request and will create an error. For example:

1. `print "My first name is $_POST[firstName]";`

We can also easily see everything that was sent by using the `print_r` function or `var_dump` like this:

1. `print_r($_GET);`

Now that we are interacting with data that is not under our control (given to us by the user, or another outside source) we have to keep in mind that what they send us cannot be trusted. If the user gave us a value we did not expect, or if someone is attempting to be malicious, we would be interacting with data that could cause errors or introduce security problems in our site. We address this through validation and sanitization (see [Integration Examples](#)), which are techniques that help us address potential problems with data we did not create.

## Cookies and Sessions

Cookies and sessions are mechanisms we can use to store and use information from any page on our site. These approaches allow us to do this without having to pass information between pages using forms or special links as we have up to this point. Cookies achieve this by storing very small files on the user's computer. They are typically used to hold onto information that identifies the user, whether or not they are logged in, or other information the user needs to achieve their full experience with the site. Cookies can be set to expire after a fixed amount of time, or "forever," by setting an expiration date far after the computer or user is likely to still be around.

Sessions allow the same storing of information, but achieve it by storing the information on the server (instead of your computer) for a fixed amount of time (usually up to 15 minutes unless the user stays active). This means sessions will still work even when the user's security settings block cookies. The use of cookies can be disabled a number of ways such as the use of security software, browser settings, and ad blockers. For this reason it can be useful to use both in your site, allowing as much functional use as possible even when cookies are denied, but still capitalizing on their longer persistence when they are available.

To create a cookie, we need to call the `setcookie()` function and pass it some variables. At a minimum, we need to give our cookie a name and a value for it to store. The name is how we will refer to it, and the value is what we want stored, just like any other variable we would create. Additionally, we can provide `setcookie()` with an expiration time, a path, a domain, and a secure flag.

The time, if passed, must be the number of seconds after creation that the cookie is considered valid for. We can achieve this by passing the `time()` function, which defaults to the current time, and adding seconds to it. For example, passing `time()+60` means the current time plus 60 seconds. If we want to make it 15 minutes, we can pass the math along instead of doing it ourselves by passing `time()+60*15`. 60 seconds, 15 times, is 15 minutes. One whole day (60 seconds, 60 times = 1 hour. 24 hours in a day) would be `time()+60*60*24`.

By default, our cookie will be considered valid on all pages within the folder we are in when we create it. We can specify another folder (and its subfolders) by placing it in the path option. The same holds true for domain, where we can specify that a cookie is only good in certain parts of our site like an admin subdomain.

Finally, we can pass a true or false (false is the default) for secure. When set to true, the cookie can only be used on an https connection to our site.

We can pass the values we want in the following order:

1. `setcookie(name, value, expire, path, domain, secure);`

A simple example setting `user=12345` for a day in our admin section of our site could look like the following:

1. `<?php setcookie("user","12345",time()+60*60*24,,admin.oursite.com); ?>`

From any page in the `admin.oursite.com` portion of our domain, we can now use `$_COOKIE["user"]` to get the value 12345. If we want to store other values, we can pass an array to our cookie, or create other cookies for other values. To change a value in our cookie, we simply use `setcookie` and give the same name with our new value:

1. `<?php setcookie("user","23456"); ?>`

Finally, if we want to get rid of our cookie early (i.e. our user logs out) then we simply set the cookie to a time in the past, and the user's computer will immediately get rid of it as it is expired:

1. `<?php setcookie("users","",time()-60*60); ?>`

In this example we set our cookie to an hour ago.

A session works much the same way, and can be created by calling `session_start()`; at the top of our page. We can then set, update, and delete variables from the session like any other array by using the reserved array `$_SESSION[]`. To add our user again, we would type:

1. `<?php session_start(); $_SESSION["user"]="12345"; ?> <html> rest of page here...`

It is important to remember that `session_start()` must be before the opening of any content, thus above the `<html>` tag. Once on a different page, we would call `session_start()` at the top again to declare that session values are allowed to be used on that page. Once we have done that, we can continue to use `$_SESSION[]` values anywhere on that page. If the user is inactive (does not leave

the page or, click any links, or otherwise trigger an action to the server) for 15 minutes (the default value) the session is automatically destroyed.

We can manually remove items from our session by calling `unset()`, for example:

1. `<?php session_start(); unset($_SESSION['User']); ?>`

Or we can jump right to ending the entire session by using the `session_destroy` function:

1. `<?php session_destroy(); ?>`

This will remove the entire `$_SESSION[]` array from memory. Create or modify another PHP page in your collection (in the same folder and site as the current example). In this second page, you will be able to call the same values out of your cookie or session (as long as you include `session_start()` in this file as well) without passing any information directly between the pages.

Learn more

Keywords, search terms: Variables, strings, arrays, cookies, session, data persistence

Sorting Arrays: <http://php.net/manual/en/array.sorting.php>

All string functions: <http://php.net/manual/en/ref.strings.php>

PHP Sessions: <http://www.sitepoint.com/php-sessions/>

Evolving Toward a Persistence Layer: <http://net.tutsplus.com/tutorials/php/evolving-toward-a-persistence-layer/>

---

3.5: Data Storage is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

- 3.5: Data Storage by [Michael Mendez](#) has no license indicated.