Learning PHP

Prepared by Georgios Lymperopoulos for Matacode

Which service to use?

Cloud9 Vs Mamp Vs Xaamp

Many people would say that local development is faster, easier to work with and it doesn't depend on an internet connection. Maybe they are right but for teaching and learning purposes cloud IDEs are great. To name a few advantages:

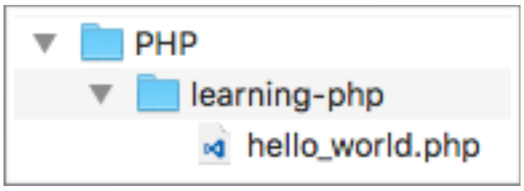
1. You no longer have the pain of getting a local LAMP setup going before you can even start a project. (Or reconfiguring your system when you want to load up an old project)
2. Every window, tab, terminal, editor is saved exactly as you like it for your project.
3. You can show your project to a friend, colleague or client with one link.
4. You can work from anywhere on any device.
5. You get automatic backups and revision history on every file.
6. It is much much much easier for teachers to debug or find any kind of problems if they know that all students work under the same environment with the exact same setup.

What about PHP Built-in web server?!

As of version 5.4.0, PHP provides a built-in web server which was designed to aid application development and it may also be useful for testing purposes or for application demonstrations that are run in controlled environments. Keep in

mind though that this web server is not intended to be a full-featured web server and it should not be used on a public network.
Let's see how it works.

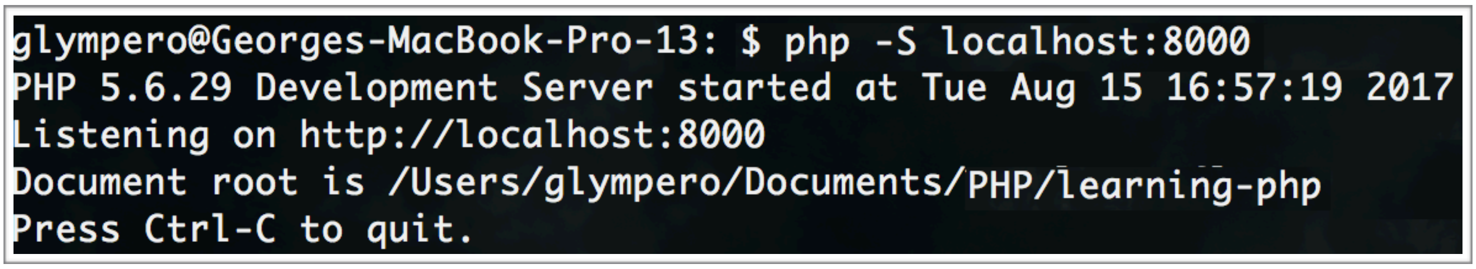
1. Open finder or windows explorer and navigate to your documents folder and create a new folder named: PHP and then a new folder under it named: learning-php. Inside the learning-php folder, create a new file named: hello_world.php.



2. Open your terminal and navigate to the learning-php folder and start the server by typing:

```
php -S localhost:8000
```

The web server starts and is listening on port 8000.



```
glympero@Georges-MacBook-Pro-13: $ php -S localhost:8000
PHP 5.6.29 Development Server started at Tue Aug 15 16:57:19 2017
Listening on http://localhost:8000
Document root is /Users/glympero/Documents/PHP/learning-php
Press Ctrl-C to quit.
```

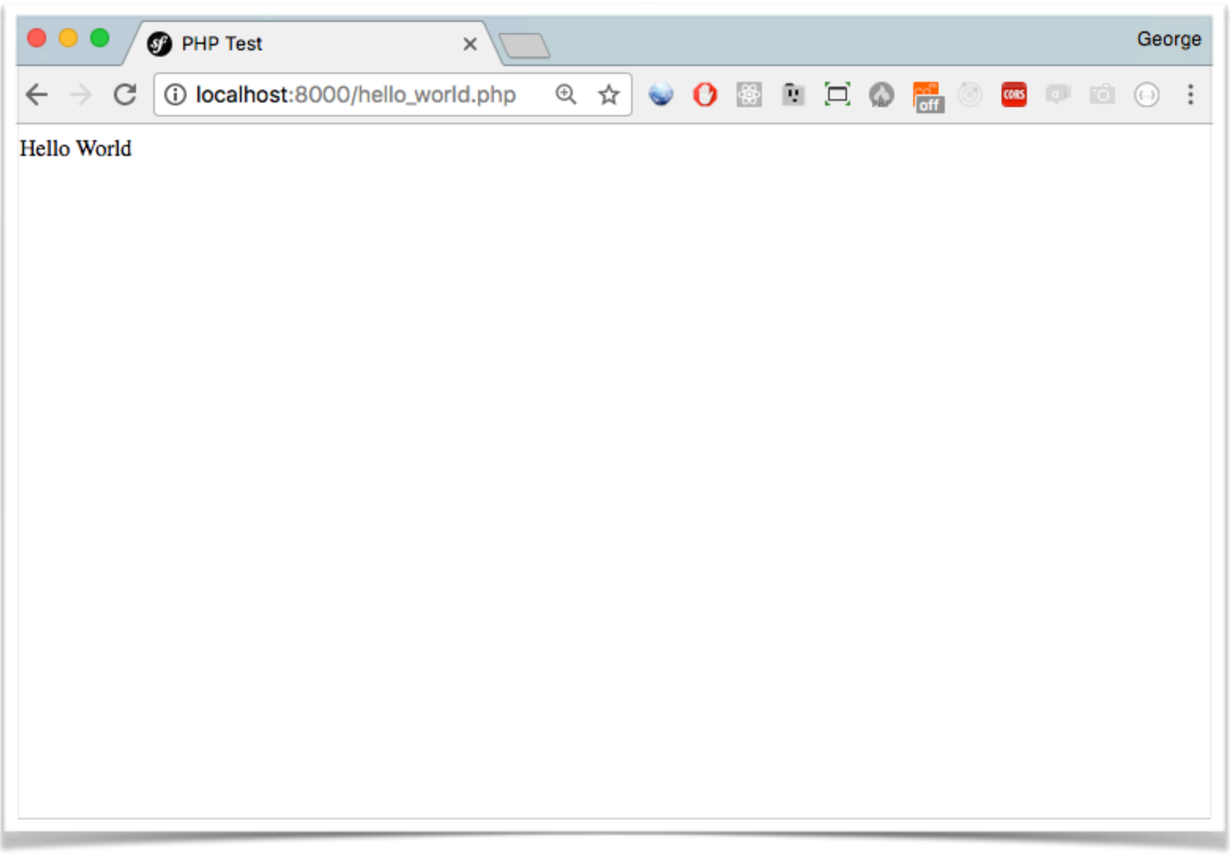
3. Now open your hello-world.php file with your favorite editor (mine would be atom). Let's write a simple, as you have already guessed, hello world page!

```
<html>
 <head>
  <title>PHP using web server</title>
 </head>
 <body>
  <?php echo '<p>Hello World</p>'; ?>
 </body>
</html>
```

4. Now open your favorite browser and navigate to:

```
http://localhost:8000/hello_world.php
```

Voila! Your first php page!



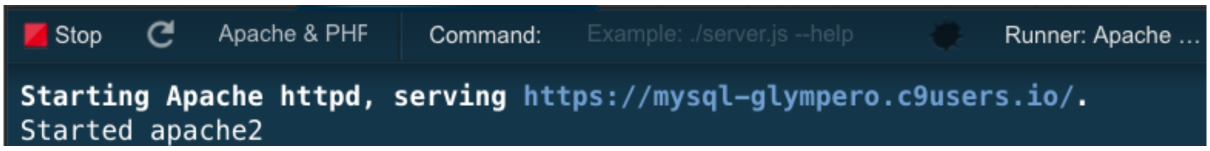
Hello World

"Hello World" using Cloud9

Let's repeat the process using cloud9 this time. Open your workspace, right-click in your mysql folder and create a new file named hello_world.php. Open the file and then write a simple web site in Cloud9 that runs through Apache.

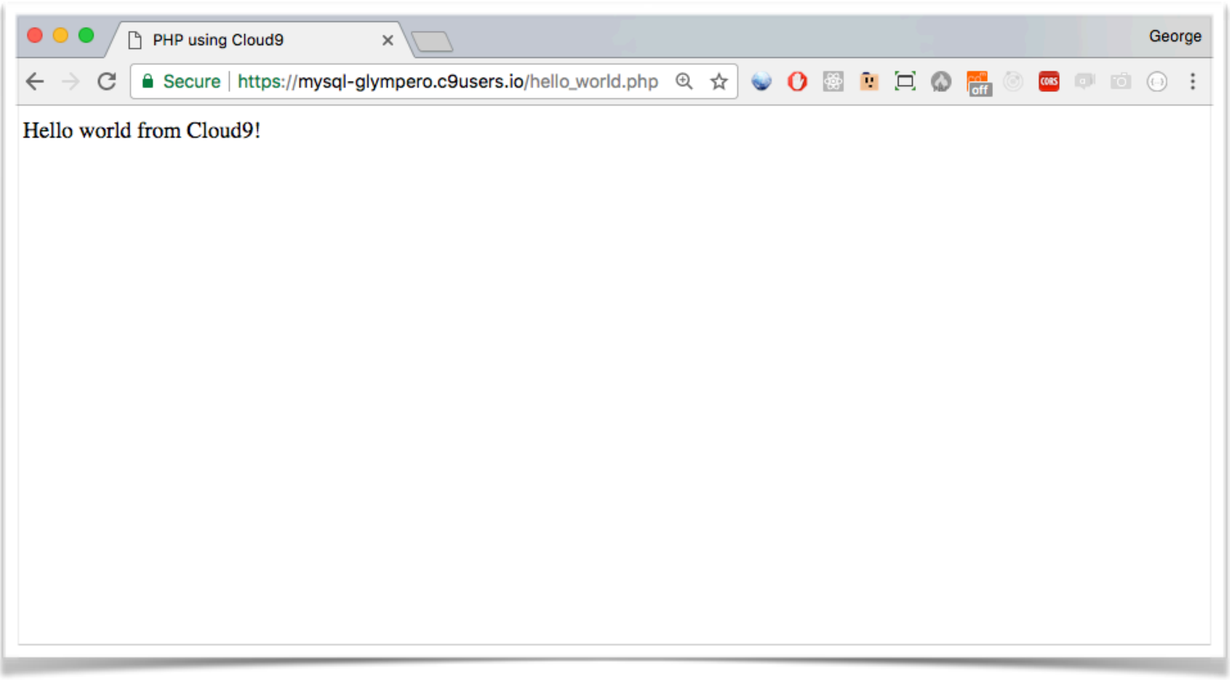
```
<html>
 <head>
  <title>PHP using Cloud9</title>
 </head>
 <body>
  <?php echo '<p>Hello world from Cloud9!</p>'; ?>
 </body>
</html>
```

Press the 'Run Project' button on the top to start the web server, then click the URL that is emitted to the output tab of the console as seen below.



```
Starting Apache httpd, serving https://mysql-glympero.c9users.io/.
Started apache2
```

Your second php page!



Hello world from Cloud9!

PHP Fundamentals

Ok these examples might sound boring but we can already identify some patterns about the structure of PHP programs.

PHP Tags

<?php is the PHP start tag and **?>** is the PHP end tag. All PHP code must be surrounded with these tags (as a start tag you can also use **<?=**) and anything

```
<html>
 <head>
  <title>PHP Tags</title>
 </head>
 <body>
    <p>
     Adding two numbers (1 and 2)
    </p>
  <?php print 1 + 2; ?>
  <p>
   Multiply two numbers (5 and 10)
  </p>
 <?php print 5 * 10; ?>
 </body>
</html>
```

outside of those tags will be completely ignored by PHP engine.
As you can see from the example above, you can have multiple php stand and end tags pairs in your program or web page. Keep in mind that the closing tag of a PHP block at the end of a file is optional, and in some cases omitting it is helpful, so unwanted whitespace will not occur at the end of files.

PHP semicolon

PHP requires instructions to be terminated with a semicolon at the end of each statement. However since **the closing tag of a block of PHP code**

automatically implies a semicolon; you do not need to have a semicolon terminating the last line of a PHP block. The closing tag for the block will include the immediately trailing newline if one is present.

```
<?php
    echo "1";
    echo "2"
          //^ semicolon missing
?>

OUTPUT
12
```

Example

```
<?php
    echo "1";
    echo "2"
          #^ semicolon missing (closing tag missing)

OUTPUT
Parse error: syntax error, unexpected end of file,
expecting ',' or ';' in
```

1. script with missing semicolon at the end, but with closing tag:
2. script with missing semicolon at the end, but without closing tag:

PHP comments

A comment in PHP code is a line that is not read as part of the program. Its only purpose is to be read by someone who is editing the code. There are several ways to add a comment in PHP code. The first (as seen in the examples above) is by using **//** or **#** to comment out a line. This one-line comment style only comments to the end of the line or the current code block, whichever comes first.

If you have a longer, multi-line comment, start the comment with **/*** and end it

```
<?php
 echo "Super Market List";
 /* My super market list which should include:
    - Milk
    - Sugar
    - Tea
 */
 echo "Milk - Sugar - Tea";
 ?>
```

with ***/**. You can contain several lines of commenting inside a block as seen in the example below:

Printing in a new line

If you want to have multiple statements / prints in your tests, you can use this to create a line break between them.

```
<?php
   print "Something here";
   print "\n"; // this prints a line break
   print "In new line now";
?>
```

Data

Echoing pieces of text or numbers is pretty boring i guess. We need a proper way to store and reuse information in the middle of a PHP program and that can be achieved by using **variables.**

- All variables in PHP are denoted with a leading dollar sign:

```
<?php
   $welcome_note = “Welcome to our Website!”;
   $x = 9;
   //Declaring variable y but not initialising it
   $y;
   //Assigning a new value to $x
   $x = 12;
   //Initialising variable $y
   $y = 3.5;
?>
```

- Variables are assigned with the **=** operator, with the variable on the left-hand side and the expression to be evaluated on the right. This means that the variable $welcome will hold the value **Hello John!**, the variable $x will hold the value **12** (The value of a variable is the value of its most recent assignment) and the variable $y will hold the value **3.5**.
- Variables can, but do not need, to be declared before assignment. As seen from the example above, variable $y has been declared and then initialised with a value. When declaring a variable, it does not know in advance whether it will be used to store a number or a string of characters. Keep in mind though that it is a good practice to always initialise your variables.

Variable Naming Conventions

As you can see from the example above, PHP variable names are case-sensitive and can either have a descriptive name (welcome_note, height, total_score) or a short name (like x and y). Rules for PHP variables:

- A variable name must start with a letter or the underscore character
- A variable name cannot start with a number

- A variable name can only contain alpha-numeric characters and underscores

Constants

Constants are similar to variables except that they cannot be changed or undefined after they have been defined. Constants are particularly useful for defining a value that you frequently need to refer to that does not ever change. For example, a program might require to define a value called INCHES_PER_YARD that contains the number of inches in a yard. Since this is a value that typically doesn't change from one day to the next it makes sense to define it as a constant.

By default, a constant is case-sensitive and by convention, constant identifiers are always uppercase and their name starts with a letter or underscore, followed by any number of letters, numbers, or underscores.

To create a constant, use the define() function:

```
<?php
   define(name, value, case-insensitive)
?>
```

The **name** specifies the name of the constant, the **value** specifies the value of the constant and the **case-insensitive** specifies whether the constant name should be case-insensitive. The default value of the latter is false.

```
<?php
  define("WELCOME_MSG", "Welcome to Learning PHP!");
  echo WELCOME_MSG;

  //Outputs "Welcome to Learning PHP!"
?>
```

Keep in mind that no dollar sign (**$**) is necessary before the constant name.

Data Types

PHP has a total of eight data types which we use to construct our variables:

- **Strings** – are sequences of characters, like 'PHP supports string operations.'

- **Integers** – are whole numbers, without a decimal point, like 4195.
- **Doubles** – are floating-point numbers, like 3.14159 or 49.1.
- **Booleans** – have only two possible values either true or false.
- **NULL** – is a special type that only has one value: NULL.
- **Arrays** – are named and indexed collections of other values.
- **Objects** – are instances of classes, which can package up both other kinds of values and functions that are specific to the class.
- **Resources** – are special variables that hold references to resources external to PHP (such as database connections).

Strings

Strings in PHP are sequences of characters, such as:

```
<?php
   $string_double_quotes = "This is a string in double quotes";
   $string_single_quotes = 'I\'have made a singly quoted string';
   $string_39 = "This string has thirty-nine characters";
   $string_0 = ""; // a string with zero characters
   $string_new_line = 'Newline is represented by \n';
?>
```

Because PHP doesn't check for variable interpolation or almost any escape sequences in single-quoted strings, defining strings this way is straightforward and fast.

```
<?php
   print 'We\'ll meet soon!';

   //Prints: We'll meet soon!
?>
```

Double-quoted strings don't recognise escaped single quotes, but they do recognise interpolated variables as well as specially interpreting certain character sequences like **\n**, **\r**, **\t**, \\, **\$**, **\"**.

```php
<?php
   $name = "Georgios";
   $literally = 'My name is $name'; //Will not print!

   print($literally);
   //Prints: My name is $name

   $literally = "My name is $name"; //Will print!
   print($literally);
   //Prints: My name is Georgios
?>
```

```php
<?php
   print "This is a " . 'beautiful day.';

   //Outputs This is a beautiful day.

   $string1 = "The price is:";
   $string2 = '$3.95';

   echo $string1 . " " . $string2;

   //Outputs The price is: $3.95
?>
```

Two strings can be joined together using the dot (**.**) concatenation operator.

PHP has a number of built-in functions that are useful when working with strings which can be found in the online PHP manual. The most popular function can be seen below:

1. substr()

This function helps you to access a substring between given start and end points

```php
string substr(string string, int start[, int length])
```

of a string. It can be useful when you need to get at parts of fixed format strings.

The return value is a substring copied from within string. When you call the function with a positive number for start (only), you will get the string from the start position to the end of the string. Starting positions starts from 0.

```php
<?php
   $name = 'Jim Davis';

   substr($name, 4); //returns 'Davis'
?>
```

When you call the function with a negative start (only), you will get the string from the end of the string minus start characters to the end of the string.

```php
<?php
   $name = 'Jim Davis';

   substr($name, -5); //returns 'Davis'
?>
```

The length parameter can be used to specify either a number of characters to return if it is positive, or the end character of the return sequence if it is negative.

```php
<?php
   $name = 'Jim Davis';
   substr($name, 0, 3); //returns ‘Jim’

   $name = 'My name is Jim Davis.';
   print substr($name, 11, -1); //returns 'Jim Davis'
?>
```

11 signifies the starting character point (**J**) and *-1* determines the ending point (count 1 character backwards starting from the end of the string).

2. strlen() and trim()

The **strlen()** function tells you the length of a string.

```
<?php
   $username = 'jimDavis@gmail.com';
   print strlen($username);
   //18
?>
```

The trim() function strips whitespace from the start and end of a string and returns the resulting string. The characters it strips by default are newlines and carriage returns (n and r), horizontal and vertical tabs (t and x0B), end-of-string characters (), and spaces. You can also pass it a second parameter containing a list of characters to strip instead of this default list. This function is particularly useful when preparing for a database insert or string comparison.
These functions combined, can used for validating input data or making sure a

```
<?php
   $trimit = 'junk Jim Davis junk';
   $trimmed = trim ( $trimit, 'junk' );

   print $trimmed;
   //Jim Davis
?>
```

string variable has a value.

```
<?php
   $username = trim('jimDavis@gmail.com ');
   $username_length = strlen ($username);

   if ($username_length > 0) {
    print 'Thanks for providing a username';
   } else {
    print 'Please enter a username';
   }
   //Outputs 'Thanks for providing a username'
?>
```

3. str_replace()

The str_replace() function replaces some characters with some other characters in a string. You can use find and replace for almost anything your imagination can think of.

```
mixed str_replace(find, replace, string, count)
```

str_replace() replaces all the instances of find with replace and returns the new version of the string optional fourth parameter count contains the number of replacements made.

```
<?php
   $my_string = 'Would you like to have a cup of coffee?';
   $search = 'coffee';
   $replace = 'tea';

   $replaced = str_replace ( $search, $replace, $my_string );

   print $replaced;
  //Prints 'Would you like to have a cup of tea?'
?>
```

A really awesome feature of **str_replace()** is the ability to pass an array to both the search terms and replace terms, as well as an array of strings to apply the rules to, but we haven't talked about arrays yet so we'll talk about that later.

4. strpos()

The **strpos()** function finds the position of the first occurrence of a string inside another string and it is case-sensitive. (**stripos()** is an alternative case-insensitive function).

```
int strpos(string, find, start)
```

The start parameter is optional and specifies where to begin the search. (position 0 is the default). The integer returned is the position of the first occurrence of the find within the string.

```php
<?php
   $my_string = 'First x. This is the second x!';

   $find = 'x';

   print strpos($my_string, $find);
   //Prints 6

   $offset = strpos($my_string, $find) + 1;

   print strpos($my_string, $find, $offset);
   //Prints 28
?>
```

The second time it prints 28 because PHP has started looking for character 'x' at position 7.

5. strtolower()

The **strtolower**() function converts a string to lowercase and it is valuable in cases where we need to compare strings - you want to make sure they are the same case - or correct capitalization.

```php
<?php
   $person = 'Angry people SHOUT!';

   echo strtolower($person);
   //angry people shout!
?>
```

6. strtoupper()

The **strtoupper**() function is also quite popular for many of the reasons listed above, in reverse, meaning take lowercase or a mixed case string and set it to all upper case.

```php
<?php
   $person = 'these people need to get working!';

   echo strtoupper($person);
   //THESE PEOPLE NEED TO GET WORKING!
?>
```

Numbers

In PHP, numbers are expressed using familiar notation, although you can't use commas or any other characters to group thousands and can be anything like the numbers we see in everyday life. Maybe they are like π, the ratio of the circumference to the diameter of a circle , or 1.40, the cost of a bus ride or even 12.00 A.M, the current time.

```
<?php
    print 34;
    print 34.3;
    print 34.30;
    print 0.9876;
    print 12000.21;
    print 0;
    print -343;
    print 100000;
    print 0.00;
?>
```

PHP breaks down numbers into two groups: **integers** and **floating-point** numbers (also called doubles). Integers are whole numbers, such as 1, -1, 0, and 2017. Floating-point numbers are decimal numbers, such as −1.1, 1.0, 3.14159, and 0.9999999999. Conveniently, most of the time PHP doesn't make you worry about the differences between the two because it automatically converts integers to floating-point numbers and floating-point numbers to integers. This conveniently allows you to ignore the underlying details. It also means 3/2 is 1.5, not 1, as it would be in some programming languages. PHP also automatically converts from strings to numbers and back. For instance, 1+"1" is 2.

Operations

PHP supports the basic math operations like addition (+), subtraction (-), division (/) and multiplication (*). In addition it also supports exponentiation (**) and modulus division (%), which returns the remainder of a division operation.

```php
<?php
    print 1 + 1;
    //prints 2
    print 10 - 4.5;
    //prints 5.5
    print 10 / 3;
    //prints 3.3333333333333
    print 6 * 10;
    //prints 60
    print 2 ** 2;
    //prints 4
    print 10 % 3;
    //prints 1
    print 10 % 2;
    //prints 0
?>
```

Booleans

A Boolean value is one that is in either of two states: TRUE or FALSE. Try this script out and see that the **true_value** will print "1", but the **false_value** won't print anything!

```php
<?php
    $true_value = true;
    $false_value = false;

    print ("true_value = " . $true_value);
    // prints true_value = 1
    print (" false_value = " . $false_value);
    // false_value won't print anything!
?>
```

Booleans are often used in conditional testing. You will learn more about conditional testing in a later chapter of this tutorial.

NULL

The special NULL value is used to represent empty variables in PHP. A variable of type NULL is a variable without any data. It identifies variables being empty or not and useful to differentiate between the empty string and null values of databases. There is only one value of type NULL, and that is the case-insensitive keyword NULL.

```
<?php
   $a = NULL;
   var_dump($a);
   //prints NULL
?>
```

Notice that we use the var_dump() function here, which is used to display structured information (type and value) about one or more variables.

Arrays

Arrays are complex variables that allow us to store more than one value or a group of values under a single variable name. Let's suppose you want to store your favourite colours in your PHP script. Storing the colours one by one in a variable could look something like this:

```
<?php
  $my_fav_color1 = "Red";
  $my_fav_color2 = "Green";
  $my_fav_color3 = "Blue";
?>
```

It seems boring and literally a bad idea to store each colour in a separate variable (imagine doing that for storing the city names of a country in separate variables). How about using an array?
There are three types of arrays that you can create. These are:

Numeric array: An array with a numeric key.

Associative array: An array where each key has its own specific value.

Multidimensional array: An array containing one or more arrays within itself.

Indexed Arrays

These arrays can store numbers, strings and any object but their index will be represented by numbers. By default array index starts from zero. The following examples shows two ways of creating an numeric array, the easiest way is:

```php
<?php
   // Define an numeric array
   $my_fav_colors = array("Red", "Green", "Blue");
?>
```

This is equivalent to the following example, in which indexes are assigned manually:

```php
<?php
   $my_fav_colors[0] = "Red";
   $my_fav_colors[1] = "Green";
   $my_fav_colors[2] = "Blue";
?>
```

Associative Arrays

In an associative array, the keys assigned to values can be arbitrary and user defined strings. This means that they are very similar to numeric arrays in term of functionality but they are different in terms of their index. In the following example the array uses keys instead of index numbers:

```php
<?php
   // Define an associative array
   $ages = array("George"=>39, "Maria"=>35, "Danai"=>3);
?>
```

The following example is equivalent to the previous example, but shows a different way of creating associative arrays:

```php
<?php
   $ages["George"] = "39";
   $ages["Maria"] = "35";
   $ages["Danai"] = "3";
?>
```

Multidimensional Arrays

PHP multidimensional array is also known as array of arrays. It allows you to store tabular data in an array. This means that we can have an array in which each element can also be an array and each element in the sub-array can be an array or further contain array within itself and so on. PHP multidimensional array can be represented in the form of matrix which is represented by row * column.

```
<?php
   // Define a multidimensional array
   $contacts = array(
       array(
         "name" => "Jim Carrey",
         "mobile" => 55512341,
       ),
       array(
          "name" => "Maria Kent",
          "mobile" => 55523341,
       ),
       array(
          "name" => "John Johnson",
          "mobile" => 55511241,
       )
   );
   // Access nested value
   echo "John Johnson's mobile number is: " . $contacts[2]
["mobile"];
```

Finding the Size of an Array

The count() function tells you the number of elements in an array.

```
<?php
   // Define array
   $cities = array("London", "Paris", "New York");

   // Count the number of cities
   print count($cities);
?>
```

Viewing Array Structure and Values

You can see the structure and values of any array by using one of two statements **var_dump**() or **print_r**(). The **print_r**() statement shows the key and the value for each element in the array.

```
<?php
   // Define array
   $cities = array("London", "Paris", "New York");
   // Display the cities array
   print_r($cities);

  /* Prints
   Array
   (
      [0] => London
      [1] => Paris
      [2] => New York
   )
   */
?>
```

The **var_dump**() statement, in addition to the key and value, it also shows the data type of each element, such as an array of 3 elements and a string of 6 characters.

```
<?php
   // Define array
   $cities = array("London", "Paris", "New York");

   // Display the cities array
   var_dump($cities);

   /* Prints
   array(3) {
    [0] => string(6) "London"
    [1] => string(5) "Paris"
    [2] => string(8) "New York"
  }
  */
?>
```

Objects

So far, we have been talking about built-in data types. Objects are user defined data types. Programmers can create their data types that fit their domain. More about objects in chapter about object oriented programming, OOP.

Resources

A resource is a special variable, holding a reference to an external resource. Resource variables typically hold special handlers to opened files and database connections and their main benefit is that they're garbage collected when no longer in use.

Type casting

Converting one data type to another one is a common job in programming. **Type-conversion** or **typecasting** refers to changing an entity of one data type into another. There are two types of conversion: **implicit** and **explicit**. Implicit type conversion, , also known as coercion, is an automatic type conversion by the compiler.

```php
<?php
    echo "45" + 12;
    //Prints 57

    echo 12 + 12.4;
    //Prints 24.4
?>
```

In the above example, we have two examples of implicit type casting. In the first statement, the string is converted to integer and added to the second operand. If either operand is a float, then both operands are evaluated as floats, and the result will be a float.

Explicit conversion happens when we use the cast constructs, like **(integer)**. Below we can see explicit casting in action where we assign a float value to variable **$a**. Later we cast it to integer and finally to a string data type.

```
<?php
    $a = 12.43;
    var_dump($a);
    // Prints float(12.43)

    $a = (integer) $a;
    var_dump($a);
    //Prints int(12)

    $a = (string) $a;
    var_dump($a);
    //Prints string(2) "12"
?>
```

PHP Functions

So far we have learned how to set a variable and how to print things but of course there is more. **Functions**. Functions help you create organised and reusable code without having to constantly update identical blocks of code in multiple places and in multiple files.
A function always start with its name, which is NOT case-sensitive, followed by opening and closing parentheses. A function can also have arguments (or parameter), which are passed to the function as a comma-separated list inside its parentheses and they let us control the behaviour of the function.

```
<?php
   function greetGeorge() {
      echo "Hello George!";
   }

   function greet($name) {
      return "Hello ".$name."!";
   }

   function greetDetails($name, $age) {
      printf('Hello %s! You are %u years old.', $name, $age);
   }

   greetGeorge(); //Prints Hello George!

   $greetJohn = greet('John');
   echo $greetJohn; //Prints Hello John!

   greetDetails('George', 40); //Prints Hello George! You are 40 years old.

?>
```

A function will not execute immediately when a page loads. To call a function, simply use the function name, specifying argument values for any parameters to the function. If the function returns a value, you can assign the result of the function to a variable, as shown in the preceding example.

PHP comes with a lot of built-in functions (more than 1000) that you can use immediately. You can read about those functions and learn more about them on php.net.

Capitalising the first character of a string

Let's say that we want to capitalise the first character of a string. We can create a custom function but after looking at php.net, there is a function for that called **ucfirst**. By reading its documentation we can see that this function has only one argument and it's required.

```
<?php
   $foo = 'hello world!';
   $foo = ucfirst($foo);  // Hello world!
?>
```

The point is that PHP has *a lot* of functions, and each has different arguments that mean different things. Some arguments are required, like the first and only argument of **ucfirst** and some are optional.

When you need to do something like generate a random number, the best thing to do is google your question, find the function you need, then research it on php.net. Every page has comments below it and a spot where you can learn about similar functions.

```
<?php
   //the rand function gives us a random number
   $randomNumber = rand();

   //the rand function give us a number that's 50 or larger
   $randomNumberAbove50 = rand(50);

   //the rand function give us a number between 50 and 100
   $randomNumberBetween50and100 = rand(50, 100);
?>
```

For **rand**, the arguments are optional: the function will work without them and has a default value of **0.**

Control Structures

All the examples of PHP code we've seen so far have been either one-statement scripts that output a string of text to the web page / terminal, or a series of statements that were to be executed one after the other in order. If you've ever written programs in other languages (JavaScript, Objective-C, Ruby, or Python), you already know that practical programs are rarely so simple.

PHP, just like any other programming language, provides special statements (decisions and loops) that enable you to affect the **flow of control**. A *decision* lets you run either one section of code or another, based on the results of a specific test. Meanwhile, a *loop* lets you run the same section of code over and over again until a specific condition is met. Those statements, which are called **control structures**, add a lot of power to PHP scripts and can be used in order to deviate from the one-after-another execution order that has dominated our examples so far and make our code truly dynamic.

If - else statements

The easiest decision-making statement to understand is the if statement. With the if() construct, you can have statements in your program that are only run if certain conditions are true. This lets your program take different actions depending on the circumstances.

```
if (condition) {
  ⋮ conditional code to be executed if condition is true
}
```

If the expression inside the parentheses evaluates to true, then the statements inside the curly braces after the if() are executed. If the expression evaluates to false, the code between the braces is skipped.

```
<?php
   $is_sunny = true;
   if ( $is_sunny == true ) {
      echo "The weather is beautiful today";
   }
   //Prints The weather is beautiful today
?>
```

You can run different statements when the if() test expression is false, by adding an **else** statement to an if construction.

```
<?php
    $is_sunny = false;
    if ( $is_sunny == true ) {
        echo "The weather is beautiful today";
    } else {
        echo "It's cloudy today";
    }
    //Prints It's cloudy today
?>
```

What happens with an if...else statement is that the first conditional statement is executed if the condition is TRUE. But if it's FALSE, the second one is executed. One of the two choices *must* be executed and under no circumstance can both (or neither) be executed.
There are also times when you want a number of different possibilities to occur, based upon a sequence of conditions. PHP even gives you a special statement , **elseif**, that you can use to combine an else and an if statement.

```
<?php
    $is_sunny = false;
    $is_raining = true;
    if ( $is_sunny == true ) {
        echo "The weather is beautiful today";
    } elseif ($is_raining == true) {
        echo "It's raining!";
    } else {
        echo "It's cloudy today";
    }
    // Prints It's raining!
?>
```

You may have as many elseif statements as you like. But as the number of elseif statements increase, you would probably be better advised to consider a **switch** statement.

Switch statement

Let's say that you want to extend our weather example to check if it's snowing or if we are expecting a thunderstorm. This can get quite cumbersome, especially if you later want to change the expression used in all of the tests.
PHP provides a more elegant way to run these types of tests: the switch statement. With this statement, you include the expression to test only once, then provide a range of values to test it against, with corresponding code blocks to run if the values match.

```php
<?php
   $weather = "thunderstorm";

   switch ( $weather ) {
      case "cloudy":
         echo "It's cloudy today";
         break;
      case "rain":
         echo "It's raining!";
         break;
      case "snow":
         echo "It's snowing!";
         break;
      case "thunderstorm":
         echo "Expecting a thunderstorm today...";
         break;
      default:
         echo "The weather is beautiful today";
    }

   //Prints Expecting a thunderstorm today...
?>
```

Here's how it works. You use the keyword **switch** followed by the variable you want to switch over and a pair of curly braces. Inside those curly braces, you list as many branches as you like. Then, a series of **case** statements test the expression against various values. If a value matches the expression, the code following the case line is executed. If no values match, the default statement is reached, and the line of code following it is executed.

The ? (ternary) Operator

One way of avoiding the verbosity of if and else statements is to use the more compact ternary operator, **?**, which unlike other PHP operators, which work on either a single expression (**!is_int($number)**) or two expressions (**$x == true**), it uses three expressions:

```
( expression1 ) ? expression2 : expression3;
```

The ? operator is passed an expression that it must evaluate (**expression1**), along with two statements to execute: one for when the expression evaluates to TRUE (**expression2**), the other for when it is FALSE (**expression3**).

```
<?php
   $is_sunny = true;
   $sunny_weather = "The weather is beautiful 
today";
   $cloudy = "It's cloudy today";

   echo $is_sunny ? $sunny_weather : $cloudy;
   //Prints The weather is beautiful today

   echo !$is_sunny ? $sunny_weather : $cloudy;
   //Prints It's cloudy today
?>
```

Programmers familiar with the **?** operator find it more convenient than if statements for such short comparisons but it can be hard to read, especially if you're not used to seeing the operator. However, it's a great way to write compact code if you just need to make a simple if...else type of decision.

Looping

In programming it is often necessary to repeat the same block of code a given number of times, or until a certain condition is met. This can be accomplished using looping statements. As with decisions, that condition must take the form of an expression. If the expression evaluates to true, the loop continues running.

If the expression evaluates to false, the loop exits, and execution continues on the first line following the loop's code block. PHP has two major groups of looping statements: for and while. The For statements are best used when you want to perform a loop a specific number of times. The While statements are best used to perform a loop an undetermined number of times.

The While loop

The while() statement is like a repeating if(). You provide an expression to while(), just like to if() and it executes a block of code if and as long as a specified condition evaluates to true. If the condition becomes false, the statements within the loop stop executing and control passes to the statement following the loop. As you have probably guessed, your code block should do something that changes the outcome of the test expression so that the loop doesn't go on forever.

```
while (condition) {
  ⁝ conditional code to be executed if condition is true
}
// Continue here when condition is false
```

Here's a simple, practical example of a while loop:

```
<?php
   $i=10;
   while ($i > 0) { // Output values from 10 to 1
      echo "The number is ".$i."<br />";
      $i--;
   }
   echo "Countdown complete!";
?>
```

The example above defines a loop that starts with variable **$i** set to 10 and continues to run as long as the **$i** is greater than 0. Once **$i** reaches 0, the expression inside the parentheses (**$i** > 0) becomes false, and the loop exits. Control is then passed to the echo() statement outside the loop, and the message **"Countdown complete!"** is displayed.

The Do...While loop

If you check our previous example, you'll notice that the expression is tested at the start of the loop, before any of the code inside the braces has had a chance to run. This means that there is always a chance (if the expression evaluates to false) that the code inside the loop would never be executed.
The Do...While statements are similar to While statements, except that the condition is tested at the end of each iteration, rather than at the beginning. This means that the Do...While loop is guaranteed to run at least once.

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

The previous example with a do…while loop where the value of **$i** will decrement at least once, and it will continue decrementing as long as it has a value of greater than 0:

```
<?php
   $i = 10;

   do {
      echo "The number is ".$i."<br />";
      // Output values from 10 to 1
      $i--;
   }

   while( $i > 0 );
   echo "Countdown complete!";
?>
```

The For loop

The for statement is used when you know how many times you want to execute a statement or a block of statements. It is a bit more complex than do and do...while, but it's a neat and compact way to write certain types of loops.

```
for (initialization; condition; increment) {
  ⋮ code to be executed;
}
```

The For statement takes three expressions inside its parentheses, separated by semi-colons. When the For loop executes, the following occurs:

- The initializing expression is executed. This expression usually initializes one or more loop counters, but the syntax allows an expression of any degree of complexity.
- The condition expression is evaluated. If the value of condition is true, the loop statements execute. If the value of condition is false, the For loop terminates.
- The update expression increment executes.
- The statements execute, and control returns to step 2.

Let's write the previous example by using a for loop:

```
<?php
   for ($i=0; $i > 0; $i--) {
      echo "The number is ".$i."<br />";
      // Output values from 10 to 1
   }
   echo "Countdown complete!";
?>
```

The Foreach loop

The Foreach loop is a variation of the For loop and allows you to iterate over elements in an array. For each pass the value of the current array element is assigned to $value and the array pointer is moved by one and in the next pass next element will be processed.

```
foreach (array as value) {
  ⋮ code to be executed;
}
```

The following example lists out the values of an array:

```
<?php
   $array = array( 10, 9, 8, 7, 6, 5, 4, 3, 2, 1);

   foreach( $array as $value ) {
      echo "The number is ".$value."<br />";
   }
?>
```

Break and continue

Normally, a while, do…while, for or foreach loop will continue looping as long as its test expression evaluates to true. Sometimes you may want to let the loops start without any condition, and allow the statements inside the brackets to decide when to exit the loop. There are two special statements that can be used inside loops: **Break** and **Continue**.

The Break statement terminates the current While or For loop and continues executing the code that follows after the loop (if any). In the following example condition test becomes true when the counter value reaches 5 and loop terminates.

```
<?php
   $i=10;
   while ($i > 0) { // Output values from 10 to 1
      $i--;

      if( $i == 5 ) {
         break;
      }
   }
   echo ("Loop stopped at i = $i" );
?>
```

The PHP **continue** keyword is used to halt the current iteration of a loop but it does not terminate the loop. This can be useful if you want to skip the current item of data you're working with; maybe you don't want to change or use that

particular data item, or maybe the data item can't be used for some reason (for example, using it would cause an error).
In the following example loop prints the value of the array but when the counter value reaches 5, it just skips the code and next value is printed.

```
<?php
    $array = array( 10, 9, 8, 7, 6, 5, 4, 3, 2, 1);

    foreach( $array as $value ) {
       if( $value == 5 )continue;
       echo "Value is $value <br />";
    }
?>
```

Object Oriented Programming

Coming soon…