PHP

- PHP is a server-side and general-purpose scripting language especially suited for web development
- PHP originally stood for Personal Home Page. However, now, it stands for Hypertext Preprocessor.
- PHP was created by Rasmus Lerdorf in 1994

PHP is a server-side language

When you open a website on your web browser, for example, https://www.phptutorial.net

The web browser sends an HTTP request to a web server where phptutorial.net is located. The web server receives the request and responds with an HTML document.

In this example, the web browser is a client, while the web server is the server. The client requests for a page, and the server serves the request.

PHP runs on the web server, processes the request, and returns the HTML document.

PHP is a general-purpose language

- Programming languages according to purpose can be: domain-specific and general-purpose languages.
- Domain-specific languages are used within specific application domains. For example, SQL is a domain-specific language. It's used mainly for querying data from relational databases and cannot be used for other purposes.
- On the other hand, PHP is a general-purpose language because PHP can develop various applications.

PHP is a cross-platform language

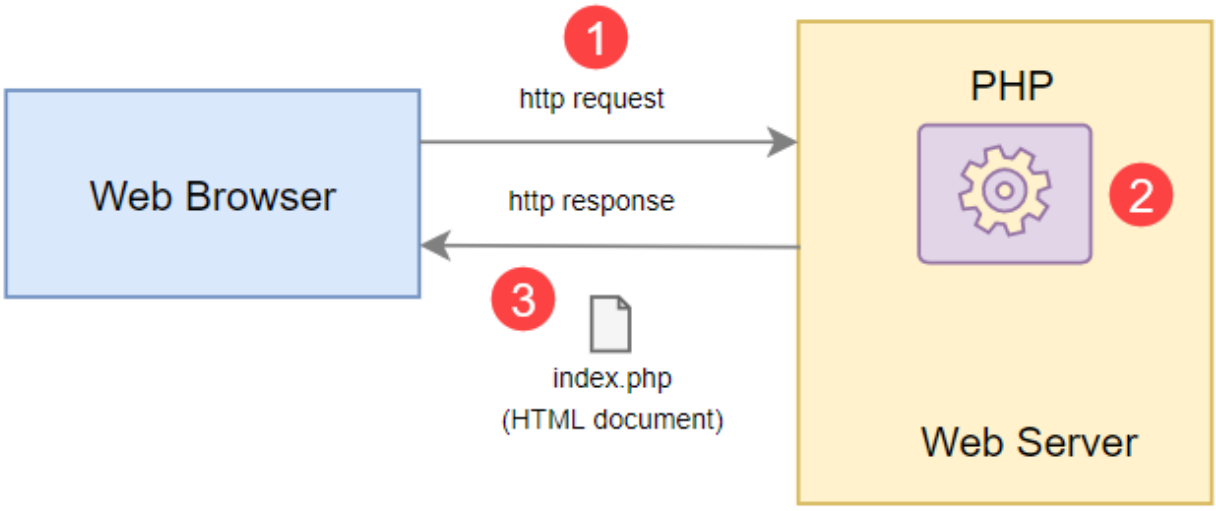
- PHP can run on all major operating systems, including Linux, Windows, and macOS.
- PHP can be used with all leading web servers, such as Nginx, OpenBSD, and Apache.
- Some cloud environments, such as Microsoft Azure and Amazon AWS, also support PHP.
- PHP is not just limited to processing HTML. It supports generating PDF, GIF, JPEG, and PNG images.
- One notable feature of PHP is that it supports many databases, including MySQL, PostgreSQL, MS SQL, db2, Oracle Database, and MongoDB.

What can PHP do

PHP has two main applications:

- Server-side scripting – PHP is well-suited for developing dynamic websites and web applications.
- Command-line scripting: Like Python and Perl, you can run PHP scripts from the command line to perform administrative tasks like sending emails and generating PDF files.

How PHP works



How PHP works:

- First, the web browser sends an HTTP request to the web server, e.g., index.php.
- Second, the PHP preprocessor located on the web server processes PHP code to generate the HTML document.
- Third, the web server returns the HTML document to the web browser.

Advantages of PHP

- Simple
- Fast
- Stable
- Open-source and Free
- Community Support

Installing PHP

- Installing PHP on your computer allows you to safely develop and test a web application without affecting the live (deployed) system.
- To work with PHP locally, you need to have the following software:
  - PHP
  - A web server that supports PHP. We'll use the Apache webserver
  - A database server. We'll use the MySQL database server
- Basically, installing all this software separately is tricky and not intended for beginners. Rather, we use an all-in-one software package that includes PHP, a web server, and a database server. One of the most popular is XAMPP which is an easy-to-install Apache distribution that contains PHP, MariaDB, and an Apache webserver. It supports Windows, Linux, and MacOS
- Note that MariaDB is a fork of the most popular relational database management system, MySQL. That is MariaDB is very similar to MySQL.

    - A fork is when developers take the source code of an existing project and start developing it independently as a separate project.

Download XAMPP

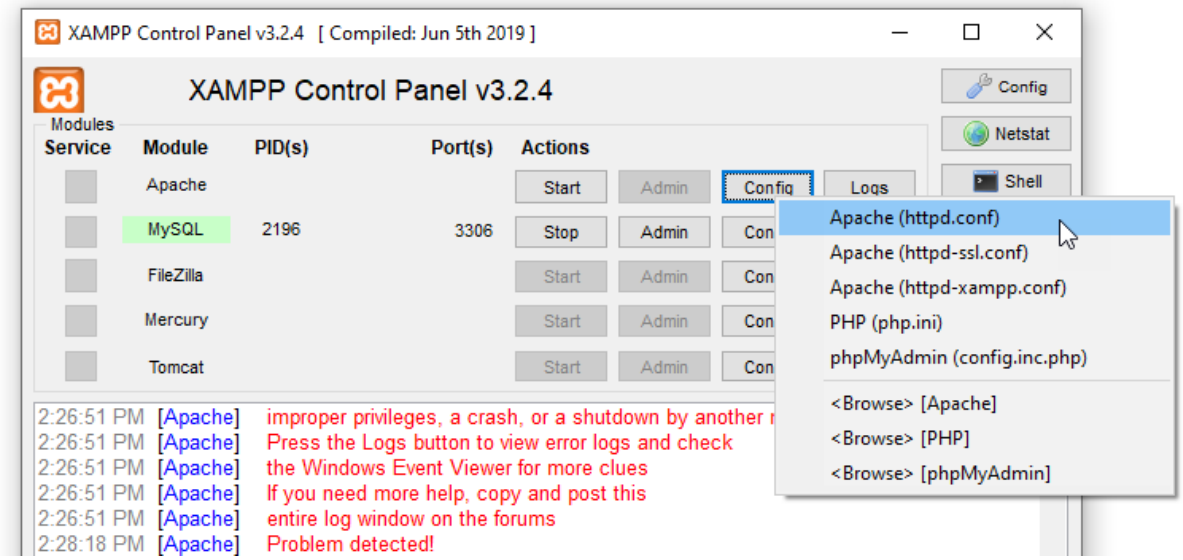
- To install XAMPP on Windows, go to the XAMPP official website and download the appropriate version for your platform.
- While installing, in the selection of components that you want to install, select Apache, MySQL, PHP, and phpMyAdmin, deselect other components, and click the Next button to go to the next step.

Troubleshooting

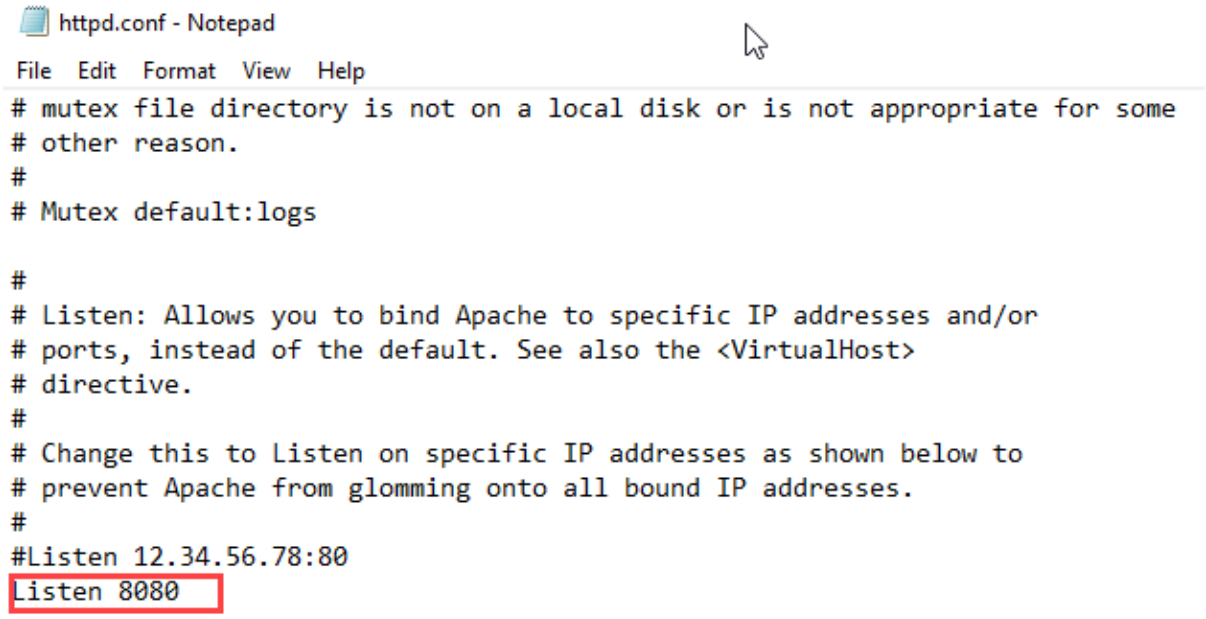
- By default, Apache uses port 80. However, if port 80 is used by another service, you'll get an error like this

```
Problem detected! Port 80 in use by "Unable to open process" with PID 4!
Apache WILL NOT start without the configured ports free! You need to
uninstall/disable/reconfigure the blocking application or reconfigure Apache
and the Control Panel to listen on a different port
```

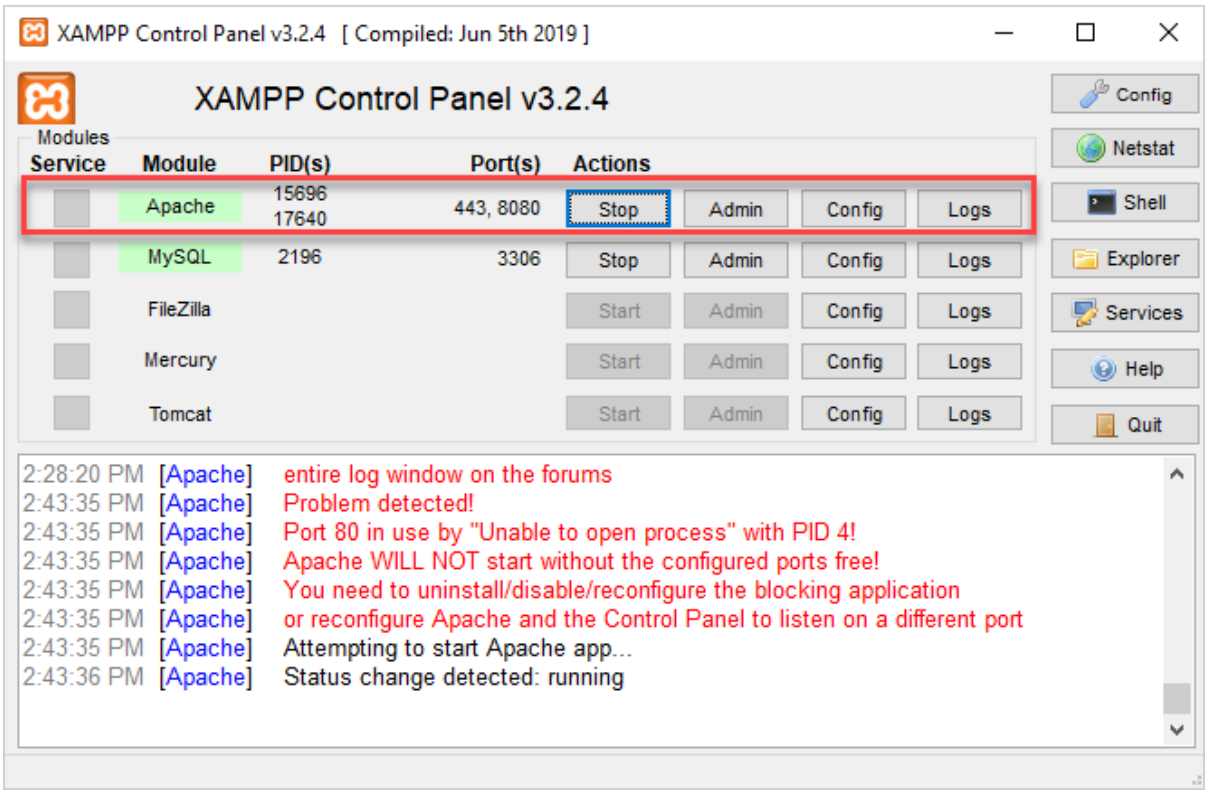
- In this case, you must change the port from 80 to a free one, e.g., 8080. To do that, you follow these steps:
    - First, click the Config button that aligns with the Apache module:



- Second, find the line that has the text Listen 80 and change the port from 80 to 8080 like this:



- Third, click the Start button to start the Apache service. If the port is free, Apache should start properly, as shown in the following picture:



How to write a PHP Program on the web browser

- First, open the folder `htdocs` under the xampp folder. Typically, it is located at `C:\xampp\htdocs`.
- In that folder, create your project directory
- Create your PHP file(s) with the `.php` extension
- Basically, your code will be in the form

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

```
  <head>
    <title>PHP Project</title>
  </head>
  <body>
    <h1><?php echo 'Hello World'; ?></h1>
  </body>
</html>
```

- Notice that the code in our PHP file looks like a regular HTML document except for the part `<?php` and `?>`.
- The code between the opening tag `<?php` and closing tag `?>` is PHP:
- This PHP code prints out the Hello, World message inside the `h1` tag using the `echo` statement (similar to `print()` in Python or `console.log()` in JS:
- Now, to see the output of your PHP code (here, output is Hello World) on the web through the browser, type `localhost/ProjectDirName`
- If you view the source code of the page, you'll see the following HTML code:

```
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>PHP - Hello, World!</title>
</head>
<body>
        <h1>Hello, World!</h1>
</body>
</html>
```

PHP Hello World on the command line

- First, open Xampp Control Panel
- Second, navigate to the Xampp Shell through that, navigate to `c:\xampp\htdocs\ProjectDirName\`.
- Third, type the following command to execute the index.php file:

```
c:\xampp\htdocs\ProjectDirName>php index.php
```

- You'll see the HTML output. Since the terminal doesn't know how to render HTML to the web, it just shows the pure HTML code.
- To fix this, write only the PHP code in the PHP file and delete all HTML
- Now, when you embed PHP code with HTML, you need to have the opening tag . However, if the file contains only PHP code, you don't need to the closing tag ?> like the index.php above.

PHP Syntax

- If you decide to mix your PHP code with your HTML file, you need to have the opening tag (`<?php`) and the enclosing tag (`?>`)
- For example

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <title>PHP Syntax</title>
  </head>
  <body>
    <h1><?php echo 'PHP Syntax'; ?></h1>
  </body>
</html>
```

- However, if a file is expected to contain only a PHP code, the enclosing tag is optional

```
<?php
  echo Hello World
```

Case Sensitivity

- PHP is partially case-sensitive. For example, if you have a function such as count, you can use it as COUNT. It would work properly
- The following are case-insensitive in PHP:
    - PHP constructs such as if, if-else, if-elseif, switch, while, do-while, etc
    - Keywords such as true and false
    - User-defined function & class names
- On the other hand, variables are case-sensitive. e.g., `$message` and `$MESSAGE` are different variables.

Statements

- A statement is a piece of code that is executed but doesn't yield a value. Statements includes such as assigning a value to a variable or calling a function.
- A statement usually end with a semi-colon `;`
- Below is a statement that assigns a literal string to the `$message` variable:
-

```
$message = "Hello";
```

This is a Simple statement

- A Compound statement consists of one or more simple statements. It uses Curly Braces to mark a block of code. Example

```
if( $is_new_user ) {
    send_welcome_email();
}
```

You don't need to place the semicolon after the curly brace (`}`)

- Also, the closing tag of a PHP block (`?>`) automatically implies a semicolon (`;`). Therefore, you don't need to place a semicolon in the last statement in a PHP block.

Whitespace & line breaks

- Whitespace and line breaks don't have special meaning in PHP. For Example,

```
login( $username, $password );
```

and

```
login(
  $username,
  $password
);
```

are equivalent

PHP Variables

Defining a variable

- A variable stores a value of any type, e.g., a string, a number, an array, or an object.
- To define a variable, you use the following syntax:

```
$variable_name = value;
```

- When defining a variable, you need to follow these rules:
  - The variable name must start with the dollar sign (`$`).
  - The first character after the dollar sign (`$`) must be a letter (a-z) or the underscore (`_`)
  - The remaining characters can be underscores, letters, or numbers.
- To display the values of variables on a webpage, you'll use the `echo` construct. For example:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>PHP Variables</title>
  </head>
  <body>
    <?php $title = 'PHP is awesome!'; ?>
    <h1><?php echo $title; ?></h1>
  </body>
</html>
```

- Variables are used for runtime data (user input, calculations, loop counters)
- Mixing PHP code with HTML will make the code unmaintainable, especially when the application grows. To avoid this, you can separate the code into separate files. For example
  - `index.php` – store the logic for defining and assigning value to variables.
  - index.view.php – store the code that displays the variables
  - Use the `require` construct to include the code from the `index.view.php` in the `index.php` file.
  - The `require` construct is used to include and evaluate a PHP file. Essentially, it takes the content of the specified file and inserts it at the point where require is called. For example suppose you created a different PHP file for storing your variables, say `index.php` and another PHP file that contains the HTML with PHP Embeded code, say `index.view.php`. We can include the `index.php` file by simply writing `require 'index.view.php'` in the `index.php` file The code `<?= $name ?>` is a shorthand for `<?php echo $name ?>`

PHP Comments

- Comments are essentially useful information for anyone reading the code. Comments are ignored by the PHP engine.
- PHP supports three types of comments:
  - Single-line comments using `//` or `#`
  - Multi-line comments using `/* */`
- Comments are written at the end of a line or a code block or on a separate line. For example

```
// This is a single-line comment
# This is also a single-line comment
/* This is a
   multi-line comment */
```

- Comments are useful for:
  - Explaining the purpose of the code
  - Making notes for future reference
  - Temporarily disabling code during debugging

- Comments should be clear and concise to effectively communicate the intended message to anyone reading the code.
- Comments should explain why a code works not what the code does. For example, the following comment is not useful

```
$total = $price * $quantity; // Multiply price by quantity to get total
```

- A better comment would be

```
// Calculate the total cost of items in the shopping cart
$total = $price * $quantity;
```

PHP Constants

- PHP constants are basically a name that holds a value. PHP constants are similar to PHP variables with the exception that
  - Constants by convention don't use the dollar sign (`$`) prefix compare to variables. Also, Constants are defined in uppercase letters. Notice that `SITE_NAME` and `site_name` are different
  - Once the value of a constant is defined, it cannot be changed or undefined
- PHP constants are defined using
  - The `define()` function - This function takes two parameters: first, the name of the constant and and second its value.
  - It's syntax is as follows:

    ```
    define( string $constant_name, mixed $value, bool $case_insensitive =
    false ): bool
    ```

    where,
    - `$constant_name` - The name of the constant. It must be a string or a string expression (such as concatenation of strings). By convention, constant names are written in uppercase letters
    - `$value` - The value of the constant. It can be of any data type, including scalar types (string, integer, float, boolean), arrays, and objects
    - `$case_insensitive` - An optional parameter that specifies whether the constant name should be case-insensitive. By default, it is set to false (case-sensitive). This parameter is deprecated as of PHP 7.3.0 and removed in PHP 8.0.0
    - Example

```
define( 'SITE_NAME', 'My PHP Website' );
```

The `define()` function cannot be used to define constants within a class.

- The `const` keyword - This keyword is used to define constants within a class or outside a class. Example

```
class Config {
    const VERSION = '1.0.0';
}
```

is a better syntax to define constants within a class. Also

```
class Config {
  define( 'Config::VERSION', '1.0.0' );
}
```

is an invalid syntax.

`define()` vs `const`

- The `define()` function can be used to define constants at runtime, while the `const` keyword is used to define constants at compile time.
- The `const` keyword can be used to define constants within a class, while the `define()` function cannot.
- The `const` keyword supports visibility modifiers (public, protected, private) when defining constants within a class, while the `define()` function does not.
- The `define()` function can be used to defined constants within a control structure while the `const` keyword cannot. Example

```
  if(!defined('PAYMENT_STATUS')) {
      const PAYMENT_STATUS = 'Completed';
  }
```

The above snippet runs an error

```
  if(!defined('PAYMENT_STATUS')) {
  define('PAYMENT_STATUS', 'Completed');
  }
```

While this is the correct way to define a constant within a control structure

- The define() is a function while the const is a language construct. That is `define()` defines a function at run-time (execution time) while `const` defines a constant at compile-time (Compile time is when your code is checked, translated, and prepared before execution). Thus `const` is faster than `define()`.
- The `define` function allows to name a constant using an expression, while the `const` keyword requires a constant name to be a valid identifier. Example

```
define( 'SITE_' . 'NAME', 'My PHP Website' ); // Valid
const SITE_' . 'NAME' = 'My PHP Website'; // Invalid
```

- Constants are often used for configuration values (site name, version, database limits)
- To check if a certain constant is defined, you can use the `defined()` function. It takes the name of the constant as a parameter and returns true if the constant is defined, otherwise false.
- It's syntax is as follows:

```
defined(string $constant_name): bool
```

- Example

```
const PAGE_TITLE = 'Animal Category';

if(defined('PAGE_TITLE')) {
    echo 'Constant defined';
} else {
    echo 'Constant not defined';
}
// Output: Constant defined
```

- It's use-cases includes
  - To avoid redefining a constant that is already defined
  - To check if a constant is defined before using it in the code
- An example of some PHP inbuilt constants includes
  - `PHP_VERSION` - This constant contains the current version of PHP that is running on the server
  - `PHP_OS` - This constant contains the operating system on which PHP is running
  - `DIRECTORY_SEPARATOR` - This constant contains the directory separator character used by the operating system (e.g., `/` for Unix-based systems and `\` for Windows)

PHP Magic Constants

- PHP Magic Constants are predefined constants that change depending on where they are used. They are called "magic" because they provide useful information about the current script or environment without requiring any additional code.
- Some examples are:
  - `__LINE__` - This constant contains the current line number in the script
  - `__FILE__` - This constant contains the full path and filename of the file or current PHP script
  - `__DIR__` - This constant contains the directory of the file. It is equivalent to `dirname(__FILE__)`
  - `__FUNCTION__` - This constant contains the name of the current function
  - `__CLASS__` - This constant contains the name of the current class
  - `__METHOD__` - This constant contains the name of the current method
  - `__NAMESPACE__` - This constant contains the name of the current namespace

PHP `var_dump()` function

- The `var_dump()` function is a built-in PHP function that dumps the value of a variable. It accepts a variable and display its type, value and size. For example, if the data-type of a variable is an string, it will display the type as "string", the value as the string itself, and the size as the number of bytes used to store the string (number of characters). Syntax

```
var_dump( $variable );
```

- To dump information about multiple variables, you can pass them as a comma-separated list to the `var_dump()` function. Example

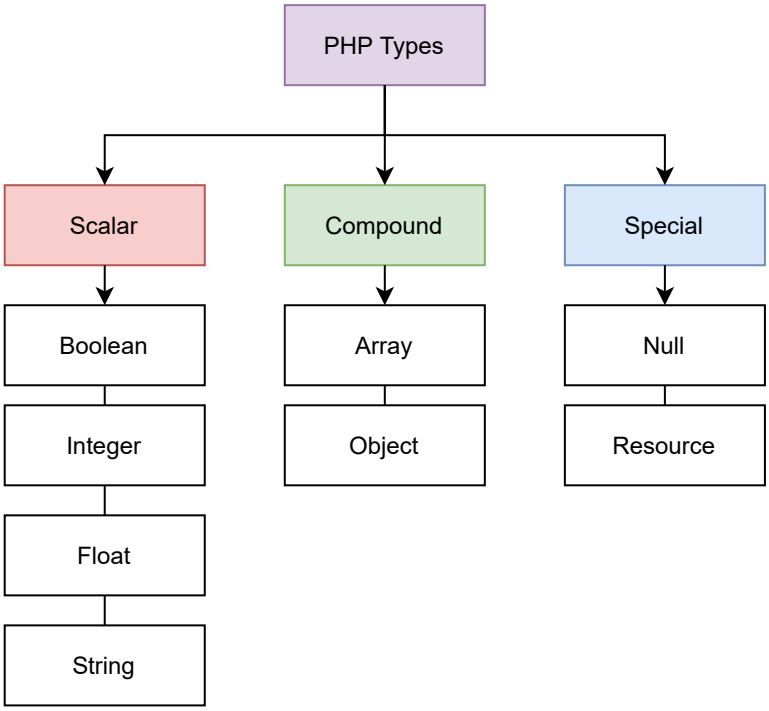
```
$age = 25;
$name = "John Doe";
var_dump( $age, $name );
```

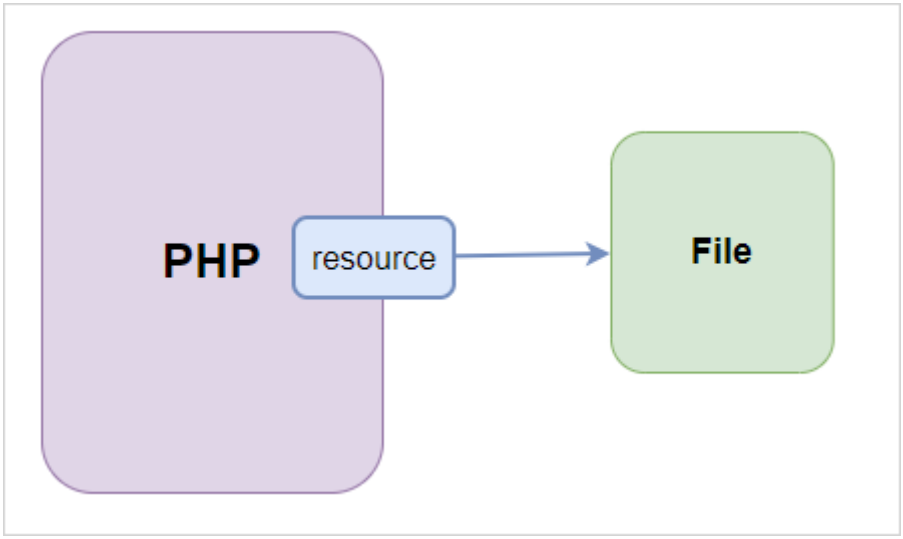
- To have a more intutive output, you can wrap the `var_dump()` function inside HTML `<pre>` tags. Example

```
$age = 25;
$name = "John Doe";
echo '<pre>';
var_dump( $age);
echo '</pre>';
echo '<pre>';
var_dump($name);
echo '</pre>';
```

The dump helper function

- It's always tedious to write the `<pre>` tags whenever you want to use the `var_dump()` function. To avoid this, you can create a helper function that takes in variables as parameters and wraps the `var_dump()` function inside the `<pre>` tags. Example

```
function dump( $variable ) {
    echo '<pre>';
    var_dump( $variable );
    echo '</pre>';
}```
+ Now, you can use the `dump()` function instead of the `var_dump()` function.
Example
```php
```
```

```
dump($age)
dump($name)
```

Dump and Die using the `var_dump()` function and `die()` function

- Sometimes, while debugging your code, you want to dump the value of a variable and stop the execution of the script immediately. To do that, you can use the `var_dump()` function along with the `die()` function. The `die()` function is used to terminate the current script. Example

```
$age = 25;
var_dump( $age );
die();
```

Notice that any code after the `die()` function will not be executed

- The `die()` function is similar to be `break` statement in other programming languages. It stops the execution of the script immediately.
- To make it more intutive, you can create a helper function that combines the `var_dump()` function and the `die()` function. Example

```
function dump_die($variable) {
  echo '<pre>';
  var_dump( $variable );
  echo '</pre>';
  die();
}
```

- Now, you can use the `dump_die()` function to dump the value of a variable and stop the execution of the script

PHP Data Types

- A type specifies the amount of memory allocated to store a value and the operations that can be performed on that value.
- PHP has 10 primitive data types - four scalars, four compound types, and two special types. They include:
    - String
    - Integer
    - Float (also called double)
    - Boolean
    - Array

    - Object
    - Callable
    - Iterable
    - NULL
    - Resource



Scalar Types

- A variable is scalar when it holds a single value of the type integer, boolean, flot(double), or string.
    - Integers are whole numbers defined from the set of positive and negative numbers including zero (Z). They are without decimal points. The size of an integer depends on the system in which PHP is running. For example, On a 32-bit system, the size of an integer is 4 bytes, while on a 64-bit system, it is 8 bytes. The constant `PHP_INT_SIZE` specifies the size of the integer on a specific platform
    - Floats (or doubles) are numbers that have decimal points or are in exponential form. They are real numbers. PHP uses the IEEE 754 double-precision format to store float values. The size of a float is platform-dependent, but it is typically 8 bytes (64 bits) on most systems. Floating-point numbers includes:
        - Decimal notation: e.g., 3.14, -0.001, 42.0
        - Exponential notation: e.g., 1.5e3 (which is equivalent to 1.5 * 10^3 or 1500)
    - Boolean represents a truth value that can be either true or false. In PHP, the boolean values are represented by the keywords `true` and `false`. Boolean values are often used in conditional statements and logical operations. Since booleans are case-insensitive, you can also write them as `TRUE` and `FALSE`. When you use other data types in a boolean context, PHP automatically converts them to boolean values using the following rules:
        - The following values are considered false:
            - The boolean value `false`
            - The integer value `0` (zero)
            - The float value `0.0` (zero)
            - An empty string `""` or a string containing only whitespace characters

        - An array with no elements
        - The special type `NULL`
        - The `SimpleXML` objects created from attributeless empty elements
      - All other values are considered true.
  - Strings are sequences of characters surrounded by either single (' ') or double (" ") quotes. Strings can contain letters, numbers, symbols, and whitespace characters.

Compound Types

- Compound types include values that can hold multiple values or collections of values. The compound types in PHP are arrays and objects. They include:
  - Arrays - An array is an ordered map that associates keys to values. There are three types of arrays in PHP:
    - Indexed arrays - These are arrays where the keys are numeric indices starting from 0. Example

```
$fruits = array("Apple", "Banana", "Orange"); // $fruits = ["Apple", "Banana", "Orange"];
```

    - Associative arrays - These are arrays where the keys are strings. Example

```
$person = array("name" => "John", "age" => 30, "city" => "New York");
// $person = ["name" => "John", "age" => 30, "city" => "New York"];
```

    - Multidimensional arrays - These are arrays that contain other arrays as elements. Example

```
$matrix = array(
    array(1, 2, 3),
    array(4, 5, 6),
    array(7, 8, 9)
); // $matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]];
```

  - Objects - An object is an instance of a class. An object consist of properties (attributes) and methods (functions). Example

```

```

Special Types

- They include
  - NULL - NULL simply represents a variable with no value. It is represented by the `null` keyword

- Resource - is a specisl variable that references to another source outside of PHP. For example, a database connection is a resource. Resources are created and used by special functions in PHP. When you are done using a resource, you should free it using the appropriate function to avoid memory leaks. Other examples of resources include file handles, image canvases, and network connections.



PHP Boolean

- A boolean is simply a truth value that can be either true or false. PHP use the `bool` type to represent boolean values.
- The `is_bool()` function can be used to check whether a variable is of boolean type. It does not allow type juggling. Example

```
$is_adult = false;
echo is_bool( $is_adult ); // Output: 1 (true)

$is_adult = 0;
echo is_bool( $is_adult ); // Output:  (false)
```

Now, when you `echo` a boolean value, PHP converts the result to a string by representing `true` as `1` and `false` as an empty string.

PHP int

- To get the size of an integer on a specific platform, you can use the constant `PHP_INT_SIZE`. Similarly, to get the maximum and minimum values of an integer, you can use the constants `PHP_INT_MAX` and `PHP_INT_MIN`, respectively.
- PHP represnts integers in decimal (base 10), hexadecimal (base 16), octal (base 8), and binary (base 2) notation.
    - Decimal numbers - PHP uses sequences of digits (0-9) to represent decimal numbers. These sequence may begin with a plus (+) or minus (-) sign to indicate positive or negative numbers. If no sign is specified, the number is assumed to be positive. Example

```
42
-7
+15
-2000
```

From PHP 7.4 onwards, you can use underscores (_) as visual separators in numeric literals to improve readability. Example

```
$largeNumber = 1_000_000;
```

- Octal numbers - PHP uses a leading zero (0) followed by digits (0-7) to represent octal numbers. Similar to decimals, Octal numbers also takes sign - they can be either positive or negative. Example

```
echo 0755;  // Equivalent to decimal 493
echo -0123; // Equivalent to decimal -83
```

- Hecadecimal numbers - Hexadecimals consist of leading zeroes (0x) or (0X) followed by digits (0-9) or letters (A-F) to represent values from 10 to 15. Hexadecimal numbers can also be either positive or negative. Example

```
echo 0x1A3;  // Equivalent to decimal 419
echo -0X4F;  // Equivalent to decimal -79
```

Particularly, the leading `0x` or `0X` is what indicates a hexadecimal number in PHP.

- Binary numbers - begins with `0b` or `0B` followed by a sequence of 0s and 1s. Binary numbers can also be either positive or negative. Example

```
echo 0b1101;  // Equivalent to decimal 13
echo -0B1010; // Equivalent to decimal -10
```

- When an integer exceeds the maximum size allowed by the platform (this is gotten by the `PHP_INT_MAX` constant), it is automatically converted to a float.
- Sum of `PHP_INT_MAX` and `PHP_INT_MIN` outputs `-1` irrespective of the order in which they appear
- The `is_int()` function can be used to check whether a variable is of integer type. It takes a variable as a parameter and returns true if the variable is an integer, otherwise false.
- The `number_format()` function is used to formmat an integer into redable form. Example,

```
print number_format(1000000);
// Output: 1000,000
```

PHP Float

- Floating-point numbers represents numerical values with decimal points or in exponential form.
- You can use the `PHP_FLOAT_MAX` and `PHP_FLOAT_MIN` constants to get the maximum and minimum representable floating-point numbers on a specific platform, respectively. You can also use `PHP_FLOAT_DIG` to get the number of decimal digits that can be represented without losing precision.
- To represent any floating point number in exponential form, you simply do this

```
echo 1.5e3; // Output: 1500
```

- Similar to integers, the range/size of float is dependent on the platform running PHP. Also, you can use underscore (_) as visual separators in numeric literals to improve readability. Example

```
$largeFloat = 3.1415_9265;
+ Floating-point number Accuracy
```

- The computer can not completely represent some floating-point numbers due to their binary representation. Therefore, when performing arithmetic operations with floating-point numbers, you may encounter precision issues. Example, the result of 0.1 + 0.1 + 0.1 is 0.299999999..., not 0.3.
- The Loose comparision operator (`==`) is used to (loosely) compare two values for equality. It allows type juggling. However, when comparing floating-point numbers, it may not yield the expected result due to precision issues. Therefore, it is recommended to use a tolerance value (epsilon) when comparing floating-point numbers. Example

```
$total = 0.1 + 0.1 + 0.1;
$expected = 0.3;
echo $total == $expected; // Output: (false)
```

Thus, PHP can only represent floating-point numbers approximately not precisely.

- To compare two floating-point numbers for equality, you can use the following approach:

```
$epsilon = 0.00001; // Define a small tolerance value
if ( abs( $total - $expected ) < $epsilon ) {
    echo "The numbers are equal.";
} else {
    echo "The numbers are not equal.";
}
```

- The `is_float()` function returns true if the variable is of float type, otherwise false

Unexpected Errors with Floats

- Consider the following code snippet

```
echo floor((0.1 + 0.7) * 10);
//Output: 7
```

How it works: - First, the expression `0.1 + 0.7` is evaluated. Due to the way floating-point numbers are represented in binary, this addition does not yield exactly `0.8`, but rather a value very close to it (e.g., `0.7999999999999999`). - Next, this approximate result is multiplied by `10`, resulting in a value close to `7.999999999999999`. - Finally, the `floor()` function is applied to this value. The `floor()` function rounds down to the nearest integer, which in this case is `7`, not `8` as one might intuitively expect.

- Consider another example:

```
echo ceil((0.1 + 0.2) * 10);
// Output: 4
```

How it works: - First, the expression `0.1 + 0.2` is evaluated. Similar to the previous example, this addition does not yield exactly `0.3`, but rather a value very close to it (e.g., `0.30000000000000004`). - Next, this approximate result is multiplied by `10`, resulting in a value close to `3.0000000000000004`. - Finally, the `ceil()` function is applied to this value. The `ceil()` function rounds up to the nearest integer, which in this case is `4`, not `3` as one might intuitively expect.

- The `NAN` constant represents a special floating-point value that stands for "Not a Number." It is used to indicate that a value is undefined or unrepresentable, especially in cases of invalid mathematical operations.
- Example of generating `NAN`:

```
$result = 0 / 0; // Division by zero
echo $result; // Output: NAN
```

- The `is_nan()` function checks whether a value is `NAN`. It returns true if the value is `NAN`, otherwise false.

- The `INF` constant represents positive infinity in PHP. It is used to indicate that a value exceeds the maximum representable floating-point number.
- The `-INF` constant represents negative infinity in PHP. It is used to indicate that a value is less than the minimum representable floating-point number.
    - `is_infinite()` function checks whether a value is positive or negative infinity. It returns true if the value is infinite, otherwise false.
    - `is_finite()` function checks whether a value is a finite number (not infinite or `NAN`). It returns true if the value is finite, otherwise false.

PHP String

- A string is a sequence of characters enclosed in either single quotes (' ') or double quotes (" "). Notice that you cannot start a string with a single quote and end it with a double quote and vice versa. The quotes must be consistent else, you'll get a syntax error. Example `'Hello, World!'` and `"Hello, World!"` are valid strings, while `'Hello, World!"` is invalid.
- To concatente string in PHP, we use the dot / concatenate (`.`) operator. Example

```
$firstName = "John";
$lastName = "Doe";
$fullName = $firstName . " " . $lastName; // Concatenate first
// and last names with a space in between
echo $fullName; // Output: John Doe
```

This also follows for single quotes

- Single vs Double Quotes
    - Single Quotes (' ') - When you enclose a string in single quotes, PHP treats the content literally. That is, it does not interpret any special characters or variables within the string. Example

```
$name = 'John';
echo 'Hello, $name\n'; // Output: Hello, $name\n
```

    - Double Quotes (" ") - When you enclose a string in double quotes, PHP interprets special characters and variables within the string. Example

```
$name = 'John';
echo "Hello, $name\n"; // Output: Hello, John (followed by a new line)
```

    Also, with double quotes, variables are parsed and their values are inserted into the string. This feature is known as *Variable Interpolation*. Example

```
$age = 25;
echo "I am $age years old."; // Output: I am 25 years old

An alternative syntax is to wrap the variable in curly braces like this:
```php
$age = 25;
echo "I am {$age} years old."; // Output: I am 25 years old.
```

Accessing String Characters
```

- A string is zero-indexed, meaning the first character is at index 0, the second character is at index 1, and so on. You can access individual characters in a string using square brackets (`[]`) with the index of the character. We use the following syntax:

```
$string[index] #index ranges from 0 to length-1
#length is the total number of characters in the string
```

- This characters can also be modified by assigning a new value to the specific index. Example

```
$name = 'isaac';
$name[0] = 'I'; // Change the first character to uppercase 'I'
echo $name; // Output: Isaac
```

If you try to reassign a character at an index that is out of bounds (greater than or equal to the length of the string), PHP will automatically extend the string and fill the gap with spaces. Example

```
$name = 'IsaaC';
$name[6] = 's';
echo $name;
// Output: IsaaC s
```

- Characters can also be accessed using negative indices. Negative indices count from the end of the string, with -1 being the last character, -2 being the second-to-last character, and so on. Example

```
$message = "Hello, World!";
echo $message[-3]; // Output: l
```

The built-in `strlen()`

- The `strlen()`function returns the length of a string. Example

  ```
  $message = "Hello, World!";
  $length = strlen( $message ); // $length will be 13
  ```

- The in built `is_string()` function checks whether a variable is of string type. It returns true if the variable is a string, otherwise false.

PHP Heredoc

- Heredoc is a syntax for defining multi-line strings in PHP. It allows you to create strings that span multiple lines without the need for concatenation or escape characters.

- Suppose you have a string that contains the double quotes, you escape them using backslashes (`\`). Example

```
$he = 'Bob'
$she = "Alice"
$message = "He said, \"Hello, $he!\"\nShe replied, \"Hi, $she!\"";
echo $message;
// Output:
// He said, "Hello, Bob!"
// She replied, "Hi, Alice!"
```

Thus for any string that contains both single and double quotes say `"any_str"`, we can escape as follows

```
$any_str = "\"any_str\""; // Using double quotes
$any_str = '\'any_str\''; // Using single quotes
```

Forward slash => Single quote => Character to be escaped => Single quote => Forward slash

- However, using escape characters can make the string hard to read and maintain. To avoid this, you can use Heredoc syntax as follows

```
$str = <<<IDENTIFIER
place a string here
it can span multiple lines
and include single quote ' and double quotes "
IDENTIFIER;
```

- How it works
  - The `<<<IDENTIFIER` indicates the start of a Heredoc string. You can use any identifier you like, but it must be a valid label (e.g., it cannot start with a number or contain spaces).
  - The string content follows the opening identifier and can span multiple lines and contain double and single quoted text.
  - The closing identifier (`IDENTIFIER;`) must be on a new line and should not have any leading or trailing whitespace.
  - Variables within the Heredoc string are parsed and their values are inserted into the string, similar to double-quoted strings.
- Asides from the readability advantage, Heredoc syntax is particularly useful to generate large blocks of text, such as HTML templates or SQL queries, where maintaining the original formatting is important. Example - Using Heredoc to generate HTML

```
$name = John Doe;
$html = <<<HTML
<div>
    <h4>Welcome</h4>
```

```
    <h1>$name</h1>
    <p>This is a sample HTML block generated using Heredoc syntax.</p>
</div>
HTML;
```

- Note that the identifier used in this case `HTML` is according to preference. You can use any valid identifier of your choice. But for context, you can use `HTML` when generating HTML content, `SQL` when generating SQL queries, and so on.
- The `nl2br()` function can be used to convert new line characters (`\n`) in a string to HTML line breaks (`<br>`). This is particularly useful when displaying multi-line strings in a web page. Example

```
$message = <<<EOT
This is line one.
This is line two.
This is line four.
This is line five.
EOT;
echo nl2br( $message );
```

- It is practical to use the `nl2br()` function when displaying Heredoc strings that contain multiple lines in an HTML context, as it preserves the line breaks in the output. If it is used with an Heredoc string generating HTML content, it may lead to unintended formatting issues.

PHP Nowdoc Syntax

- It is simlar to Heredoc syntax with the exception that variables within a Nowdoc string are not parsed. That is, they are treated literally. Syntax

```
$str = <<<'IDENTIFIER'
place a string here
Another line here
and include single quote ' and double quotes "
variables like $name will not be parsed
IDENTIFIER;
```

Notice that the identifier after the `<<<` operator is enclosed in single quotes (`' '`). Also, the embedded variables like `$name` ar treated as literal text not parsed.

- Heredoc strings are like double-quoted strings without escaping.
- Nowdoc strings are like single-quoted strings without escaping.

Introduction to NULL in PHP

- The `null` type is a single-value type with one value: `NULL`. A variable of type `null` has no value assigned to it. It is classified as a special type in PHP

- A variable is considered to be of type `null` if:
  - It has been assigned the constant `NULL`
  - It has not been assigned any value yet
  - It has been unset using the `unset()` function - the `unset()` function is used to destroy a variable. Once a variable is unset, it no longer exists in the current scope and is considered to be of type `null`. Example

```
$var = "Hello, World!";
var_dump( $var ); // Output: string(13) "Hello, World!"
unset( $var );
var_dump( $var ); // Output: NULL
```

- Since PHP keywords are case-insensitive, you can also use `null`, `Null`, or `NuLL` to represent the `NULL` constant.

- The `is_null()` function checks whether a variable is of type `null`. It returns true if the variable is `null`, otherwise false.
- You can also use the Strict comparison operator (`===`) to check if a variable is `null`. Example

```
$var = null;
if ( $var === null ) {
    echo "The variable is null.";
} else {
    echo "The variable is not null.";
}
```

- Pratically, if you try to echo/print the value of a null variable, you'll get an empty string. Example

```
$x = 95;
unset($x);

echo $x;
Output:
```

Also, type casting null to - string results to an empty string `""` - integer results to `0` - boolean results to `false` - array result to `array{}`

PHP gettype() function

- The `gettype()` function is a built-in PHP function that returns the data type of a variable as a string. It takes a single parameter, which is the variable whose type you want to determine.
- Syntax

```
gettype( mixed $variable ): string
```

- Basically, the data-type of each variable is determined at runtime based on the value assigned to it. PHP sees anything surrounded with quotes as a string, whole numbers as integers, numbers with decimal points as floats, and so on.

PHP Type Casting

- Type casting is the process of converting a value from one data-type to another. To cast a value to a specific type in PHP, you can use the following syntax:

```
$variable = (type) $value;
```

Thus for example, to cast a variable to an integer, you can use `(int)`, `(integer)`, to cast to a float, you can use `(float)`, `(double)`, or `(real)`, to cast to a string, you can use `(string)`, to cast to a boolean, you can use `(bool)` or `(boolean)`, and to cast to an array, you can use `(array)`.

- Cast to an integer
    - Suppose you want to cast a string to an integer. If the string is leading numeric, PHP will convert the numeric part to an integer and ignore the rest. Example

```
$price = "199 USD";
$price_float = (int) $price; // $price_float will be 199
```

    - If the string is not leading numeric or contains no numeric characters or null, PHP will convert it to 0. Example

```
$price = "USD 199";
echo (int) $price; // Output: 0
```

    - If the string is decimal, it truncates the decimal part and converts only the integer part. Example

```
echo (int) "99.99"; // Output: 99
```

    Same applies if the numeric string is separated by _ for readability

```
echo (int) "1_000.99"; // Output: 1
```

    This also applied if the numeric string is separated by _ only for readability

```
echo (int) "1_000"; // Output: 1
```

- Cast to a float
    - Similar to casting to an integer, when casting a string to a float, if the string is leading numeric, PHP will convert the numeric part to a float and ignore the rest. Example

```
$price = "199.99 USD";
$price_float = (float) $price; // $price_float will be 199.99
```

    - If the string is not leading numeric, PHP will convert it to 0.0. Example

```
$price = "USD 199.99";
echo (float) $price; // Output: 0
```

    - Also notice that in Python programming language, when you cast an integer to a float, it adds a decimal point followed by a zero. However, in PHP, it simply converts the integer to a float without adding the decimal point. Example

```
$num = 42;
$num_float = (float) $num; // $num_float will be 42 not 42.0
```

- Cast to a string
    - When casting a value to a string, PHP converts the value to its string representation. Example

```
$age = 25;
$age_str = (string) $age; // $age_str will be "25"
```

    - For boolean values, `true` is converted to "1" and `false` is converted to an empty string. Example

```
$is_adult = true;
$is_adult_str = (string) $is_adult; // $is_adult_str will be "1"
```

    - The `(string)` operator converts an array to the string "Array". Example

```
$fruits = array("Apple", "Banana", "Orange");
$fruits_str = (string) $fruits; // $fruits_str will be "Array"
```

- Notice that when concatenating an array with a string, PHP automatically converts the array to the string "Array". Example

```
$fruits = array("Apple", "Banana", "Orange");
$message = "Fruits: " . $fruits; // $message will be "Fruits: Array"
```

- Also when concatenating a string with an integer or float, PHP automatically converts the integer or float to its string representation. Example

```
$age = 25;
$message = "Age: " . $age; // $message will be "Age: 25"
```

PHP Type Juggling

- PHP is a losely/dynamically typed programming language. That is, you don't need to explicitly declare the data-type of a variable when defining it. PHP automatically determines the data-type of a variable based on the value assigned to it. This feature is known as type juggling. For example, if you assign an integer value to a variable, PHP treats it as an integer. If you later assign a string value to the same variable, PHP treats it as a string. Example

```
$var = 42; // $var is treated as an integer
echo gettype( $var ); // Output: integer
$var = "Hello, World!"; // $var is now treated as a string
echo gettype( $var ); // Output: string
```

- Type juggling simply mean when converting a data-type to another data-type, PHP automatically convert them to the common most comparable data-type. For example
  - when you use the addition operator (+) to add an integer and a float, PHP automatically converts the integer to a float before performing the addition. Example

```
$int_num = 10; // Integer
$float_num = 5.5; // Float
$sum = $int_num + $float_num; // $int_num is converted to float
echo $sum; // Output: 15.5
```

- Also, when you use the concatenation operator (.) to concatenate a string and an integer, PHP automatically converts the integer to its string representation before performing the concatenation. Example

```
$str = "The answer is ";
$int_num = 42;
```

```
$message = $str . $int_num; // $int_num is converted to string
echo $message; // Output: The answer is 42
```

- When you compare two values of different data-types using the equality operator (==), PHP automatically converts them to a common comparable data-type before performing the comparison. Example

```
$int_num = 10; // Integer
$str_num = "10"; // String
if ( $int_num == $str_num ) { // $str_num is converted to integer
    echo "The values are equal."; // This will be executed
} else {
      echo "The values are not equal.";
  }
```

- Comparing string and integer using the equality operator (==) results in type juggling, where the string is converted to an integer for comparison.
- Loosely comparing `NULL` with the boolean false using the equality operator (==) results in type juggling, where `NULL` and `false` are treated as false for comparison.
- PHP has conversion priorities:
    - Numeric comparison if possible
    - Boolean comparison only when necessary
    - Special loose rules for NULL, "", "0"

PHP Operators

- PHP provides various operators to perform operations such as addition, subtraction, multiplication, division, modulus and exponenetial on variables and values.
- In PHP, numerical operators require a numerical value (integer or float) on both sides. If one of the operands is not a number, PHP will attempt to convert it to a number using type juggling rules before performing the operation. If the conversion is not possible, PHP will treat the non-numeric operand as 0.
- The following are the arithmetic operators in PHP:
    - Addition (+) -
        - The sum of an integer and a float or a float and a float is a float.
        - Computers cannot represent floats precisely. In some cases, the result will not be what you expect. For example:

```
$result = 0.1 + 0.2;
echo $result; // Output: 0.30000000000000004
```

An issue will occur if you add floats and then compare the result using the equality operator (==) to another float. To avoid this, you can use a tolerance value (epsilon) when comparing floating-point numbers. Example

```
$total = 0.1 + 0.2;
$expected = 0.3;
echo $total == $expected; // Output: (false)
// Fix
$epsilon = 0.00001; // Define a small tolerance value
if ( abs( $total - $expected ) < $epsilon ) {
    echo "The numbers are equal.";
} else {
    echo "The numbers are not equal.";
}
```

- Add a number to a string reults in PHP converting the string to a number using type juggling rules before performing the addition to yield a numerical result. If the numerical string contains a non-numeric character, PHP issues a warning and ignores the non-numeric part. Example

```
$num_str = "100 apples";
$num = 50;
$sum = $num_str + $num; // $num_str is converted to
echo $sum; // Output: 150
```

Warning: A non-numeric value encountered in /path/to/file.php on line X

- If PHP fails to convert a string to a number, it will result into a fatal error. Example

```
$num_str = "apples 100";
$num = 50;
$sum = $num_str + $num; // Fatal error
```

- Notice that the string wasn't converted to a truthy value which is 1. Instead, it resulted into a fatal error.
- Also, the + operator can be used for type casting as follow

```
var_dump(+'10')
// Output: int(10)
```

- Subtraction (-) - subtracts one number from another
  - Subtracting a float from an integer or a float from a float results in a float.

        - Similar to addition, subtracting a float from another float is inaccurate due to the imprecision of float representation in computers. Example

```
$x = 0.3 - 0.1;
echo $x; // Output: 0.19999999999999998
// expected output is 0.2
```

        - Also, the - operator can be used for type casting as follow

```
var_dump(-'10')
// Output: int(-10)
```

    - Multiplication (*) - multiplies two numbers together
        - Multiplying a float from an integer or a float from a float results in a float.
    - Division (/) - divides first number by the second number
        - Dividing a float from an integer or a float from a float results in a float.\
        - If two numbers can be evenly divided, it returns an integer whether the two numbers are float or integer.
        - Dividing by zero results in a fatal error. Suppose you still want to divide by an 0, you can use a function called `fdiv()` then pass in the two numbers accordingly. It returns `INF`
        - Basic Division rules still follows
    - Modulus (%) - returns the remainder of a division operation
        - Suppose you pass in floats as operands, the two will first be cast into integers before being operated on. Thus `echo 10.2 % 2.9` will return `0`. To get the proper float result, you can use the `fmod()` function the pass in the operands accordingly
        - Also, the sign of the remainder is dependent on the first operand (if first operand is negative, remainder will be negative and so on)
    - Exponentiation (**) - raises a number to the power of another number-

- PHP decides how to convert values based on the OPERATOR being used — not just the values themselves.

PHP Assignment Operators

- The = represents the assignment operator in PHP. It is used to assign a value to a variable.
- Basically, it assigns a values or an expression to a variable and returns the assigned value. Syntax

```
$variable = value;
```

- On the left side of the assignment operator is the variable name while on the right side is the value or expression to be assigned to the variable.
- When evaluating an assignment expression, PHP first evaluates the right side of the assignment operator to obtain the value to be assigned. Then, it assigns that value to the variable on the left side. Example

```
$first_name = "John";
$last_name = "Doe";
$full_name = $first_name . " " . $last_name; // Concatenate first and last names
echo $full_name; // Output: John Doe
```

Here, the assignemnt operator returns a value which is the value of the expression at the right side of the operator.

- Consider this piece of code

```
echo $full_name = 'Harry' . ' ' . 'Potter', '<br>';
echo $full_name, '<br>';
```

First, PHP executes code from left to right. Thus, it first evaluates the expression `$full_name = 'Harry' . ' ' . 'Potter'` which results in the value `Harry Potter`. Then, it assigns this value to the variable `$full_name`. Finally, it echoes the value of `$full_name`, which is `Harry Potter`. The second `echo` statement simply outputs the value of `$full_name` again, which is still `Harry Potter`.

Assignment tips

Double Assignment

```
$x = $y = 10;

echo $x; // 10
echo $y // 10
```

Complex Assignment

```
$x = ($y = 10) + 5;

echo $x; // 15
echo $y; // 10
```

Here, the expression in the parenthesis is first executed i.e assigning the variable `$y` to `10` the summing it up to `5` yields `15` which is assigned to variable `$x`. Know that this syntax is not recommended since it makes code less readable

Arithmetic Assignment Operators

- Suppose you want to add a value to an existing variable and update the variable with the new value. You can use the addition assignment operator (`+=`) to achieve this. Example

```
i = 0
i += 5; // Equivalent to i = i + 5
echo i; // Output: 5
```

Thus for any arithmetic operation, you can use the corresponding arithmetic assignment operator to update the variable with the result of the operation. The following are the arithmetic assignment operators in PHP:

- Addition assignment (`+=`) - adds a value to a variable and assigns the result to the variable
- Subtraction assignment (`-=`) - subtracts a value from a variable and assigns the result to the variable
- Multiplication assignment (`*=`) - multiplies a variable by a value and assigns the result to the variable
- Division assignment (`/=`) - divides a variable by a value and assigns the result to the variable
- Modulus assignment (`%=`) - calculates the modulus of a variable by a value and assigns the result to the variable
- Exponentiation assignment (`**=`) - raises a variable to the power of a value and assigns the result to the variable

- Notice that the arithmetic assignment operators perform the arithmetic operation on the current value of the variable and the specified value, then update the variable with the new result.
- Also, in the syntax, the variable appears at the left side of the operator while the value appears at the right side. Furthermore, for each arithmetic assignment operator, the arithmetic operation is performed and comes first before the assignment operation. Below is more of a general syntax of an arithmetic assignment operator

```
$variable operator= value;
```

- Where `operator` can be any of the arithmetic operators (+, -, *, /, %, **)
- The arithmetic assignment operators combine the arithmetic operation and the assignment operation into a single operation, making the code more concise and easier to read.

Concatenation Assignment Operator

PHP uses `.` for concatenation. To concatenate a string to an existing variable and update the variable with the new value, you can use the concatenation assignment operator (`.=`). Example

```
$message = "Hello";
$message .= ", World!"; // Equivalent to $message = $message . ", World!"
echo $message; // Output: Hello, World!
```

PHP Comparison Operators

- PHP comparsion operators are used to compare two values. They return a boolean value (true or false) based on the result of the comparison.
- The following are the comparison operators in PHP:
    - Equal (`==`) - returns true if two values are equal without considering the type (loose comparison => type juggling) else false
        - One of the exception to this is the following code

        ```
        var_dump(0 == 'hello');
        //bool(false)
        ```

        Here, since the string `'hello'` is not a numeric string, it converts the first operand to a string i.e `'0'` and does the comparison
    - Identical (`===`) - returns true if two values are equal and of the same type (strict comparison) else false. To avoid potential issues, use the identical operator for comparison
    - Not equal (`!=` or `<>`) - returns true if two values are not equal (with type juggling - loose comparison) else false
    - Not identical (`!==`) - returns true if two values are not equal or not of the same type (strict comparison) else false
    - Greater than (`>`) - returns true if the left value is greater than the right value else false
    - Less than (`<`) - returns true if the left value is less than the right value else false
    - Greater than or equal to (`>=`) - returns true if the left value is greater than or equal to the right value else false. Note that `>` comes first before `=`. If otherwise, it will be treated as the assignment operator and you will most likely get a syntax error
    - Less than or equal to (`<=`) - returns true if the left value is less than or equal to the right value else false. Similarly, note that `<` comes first before `=`. If otherwise you will most likely get a syntax error
    - Spaceship Operator (`<=>`) - returns `0` if first operand is equal to second operand - return `-1` if first operand is less than second operand - return `1` if first operand is greater than second operand
    - Null Coalescing Operator (`??`) - It is used as follow

```
$x = null;
$y = $x ?? 'Hello';

echo $y;
```

The above code will output `Hello`. The operator takes two operand - the `null` constant or an expression yielding null and the value or expression. By this, the operator will return the second operand only if the first is null or an expression yielding `null` else it returns the first operand

PHP AND Operator

- The logical AND operator (denoted by `and` or `&&`. Note that by case-insensitivity, And, AND and aNd are also valid but by convention, use `and` or `&&`) is used to combine two boolean expressions. It returns true if both expressions are true, otherwise it returns false.
- The following illustrates the result of the logical AND operator on two boolean values

```
| Expression 1 | Expression 2 | Result  |
|--------------|--------------|---------|
| true         | true         | true    |
| true         | false        | false   |
| false        | true         | false   |
| false        | false        | false   |
```

- `and` and `&&` are the same but they differ in their precedence. The `and` operator has a lower precedence than the `&&` operator. This means that when using `and`, it is evaluated after, while `&&` is evaluated before.

Short-Circuiting

- When the values of the first boolean operand is false, the second operand is not evaluated because the overall result will be false regardless of the value of the second operand since we know that `false && anything` is always false. This is known as Short-Circuiting. Thus, the `&&` or `and` operator is for Short-Circuiting logical AND operation.

PHP OR Operator

- The logical OR operator (denoted by `or` or `||`. Note that by case-insensitivity, Or, OR and oR are also valid but by convention, use `or` or `||`) is used to combine two boolean expressions. It returns true if at least one of the expressions is true, otherwise it returns false.
- The following illustrates the result of the logical OR operator on two boolean values

```
| Expression 1 | Expression 2 | Result  |
|--------------|--------------|---------|
| true         | true         | true    |
| true         | false        | true    |
| false        | true         | true    |
| false        | false        | false   |
```

- Similarly, `or` and `||` are the same but they differ in their precedence. The `or` operator has a lower precedence than the `||` operator. This means that when using `or`, it is evaluated after, while `||` is evaluated before.

Short-Circuiting

- When the values of the first boolean operand is true, the second operand (expression or value) is not evaluated because the overall result will be true regardless of the value of the second operand since we know that `true || anything` is always true. This is known as Short-Circuiting. Thus, the `||` or `or` operator is for Short-Circuiting logical OR operation.
- Practically, the or operator is used in the following pattern

```
function_name() || die("Error message");
```

Here, if `function_name()` returns false, the `die()` function is executed to terminate the script with an error message. If `function_name()` returns true, the `die()` function is not executed. Also, regardless of the return value, if the function executes successfully without errors, it returns true and the script continues to run.

The PHP OR gotchas

- Consider the following code snippet

```
$access_granted = false or true;
var_dump( $access_granted ); // Output: bool(false)
```

Here, the output is `bool(false)` because the assignment operator (`=`) has a higher precedence than the `or` operator. The expression is evaluated as follows

```
( $access_granted = false ) or true;
```

By this, it assigns the $access_granted variable the value false before the `or` operation is evaluated. Therefore, printing $access_granted outputs false.

- To achieve the expected result, you can use parentheses to explicitly define the order of evaluation as follows

```
$access_granted = ( false or true );
```

- This logical fuss applies to the `and` operator as well. Thus, always use parentheses when combining assignment with logical operators to avoid confusion. Or to rather play safe, use `&&` and `||` instead of `and` and `or` when combining with assignment.

PHP NOT Operator

- Unlike the AND and OR operators that are binary operators (require two operands), the NOT operator (denoted by `!`) is a unary operator (requires only one operand). It is used to negate a boolean expression. It returns true if the expression is false, and false if the expression is true.

- The following illustrates the result of the logical NOT operator on a boolean value

```
| Expression | Result  |
|------------|---------|
| true       | false   |
| false      | true    |
```

PHP Logical Operators Precedence

```
NOT ( ! ) > AND ( && ) > OR ( || ) > AND ( and ) > or ( or )
```

PHP Operators Precedence

```
Arithmetic Operators (highest):
( ) > ** > * / % > + -
> . > << >> > >>
Comparison Operators:
< <= > >= == != === !== <>
>= <=
Assignment Operators:
= += -= *= /= %= .= **=
Logical Operators (lowest):
and or
```

PHP Increment Operators

- The increment operator (++) is used to increase the value of a variable by 1. It can be used in two ways: as a prefix operator (before the variable) or as a postfix operator (after the variable).
- When used as a prefix operator, the value of the variable is incremented before it is used in an expression. Example

```php
$x = 5;
$y = ++$x; // $x is incremented to 6, then assigned to $y
echo $y; // Output: 6
```

- When used as a postfix operator, the value of the variable is incremented after it is used in an expression. Example

```php
$x = 5;
$y = $x++; // $y is assigned the value of $x (5), then $x is incremented to 6
echo $y; // Output: 5
```

Unexpected behaviours of Increment Operators

- When using the increment operator on a string that contains a (fully) numeric value, PHP will increment the numeric part of the string. Example

```php
$str = "99";
$str++;
echo $str; // Output: 100
```

- Array, Booleans and Resources cannot be incremented. Arrays will result in a fatal error while booleans and resources will issue a warning and have no effect.
- Decrementing a null value has no effect but incrementing a null value results to 1. Example

```php
$var = null;
$var++;
echo $var; // Output: 1
```

- Decrementing a string has no effect but incrementing a string increases the last character of the string. Same aplies to numeric strings (integer + text). Example

```php
$str = "abc";
$str++;
echo $str; // Output: abd
```

PHP Bitwise Operators

- PHP bitwise operators are used to perform bit-level operations on integers. They operate on the binary representation of the integers. The following are the bitwise operators in PHP:
    - AND (&) - performs a bitwise AND operation between two integers. Example

```php
$a = 5; // Binary: 0101
$b = 3; // Binary: 0011
$result = $a & $b; // Binary: 0001 (Decimal: 1)
echo $result; // Output: 1
```

This will return 1 because only the last bit in both numbers is 1.

- OR (|) - performs a bitwise OR operation between two integers. Example

```
$a = 5; // Binary: 0101
$b = 3; // Binary: 0011
$result = $a | $b; // Binary: 0111 (Decimal: 7)
echo $result; // Output: 7
```

This will return 7 because all bits that are 1 in either number are set to 1.

- XOR (^) - performs a bitwise XOR operation between two integers. Example

```
$a = 5; // Binary: 0101
$b = 3; // Binary: 0011
$result = $a ^ $b; // Binary: 0110 (Decimal: 6)
echo $result; // Output: 6
```

This will return 6 because only the bits that are different between the two numbers are set to 1.

- NOT (~) - performs a bitwise NOT operation on an integer. Example

```
$a = 5; // Binary: 0101
$result = ~$a; // Binary: 1010 (Decimal: -6)
echo $result; // Output: -6
```

- Left Shift (<<) - shifts the bits of an integer to the left by a specified number of positions. Example

```
$a = 5; // Binary: 0101
$result = $a << 1; // Binary: 1010 (Decimal: 10)
echo $result; // Output: 10
```

Here, the bits of 5 are shifted one position to the left, resulting in 10.

- Right Shift (>>) - shifts the bits of an integer to the right by a specified number of positions. Example

```
$a = 5; // Binary: 0101
$result = $a >> 1; // Binary: 0010 (Decimal: 2)
echo $result; // Output: 2
```

Here, rather than adding a 0 at the leftmost position, the leftmost bit is replicated (sign bit) to fill in the empty positions. Basically, for this operator, the specified number of bits are removed from the right

side of the binary representation of the integer.

Use-cases of Bitwise Operators

- The use cases of bitwise operators include:
  - Performing low-level operations on binary data
  - Manipulating individual bits in a number
  - Implementing certain algorithms that require bit-level operations
  - Setting, clearing, or toggling specific bits in a number
  - Performing efficient calculations on large sets of data
- Some practical examples of bitwise operations in PHP scenerio include:
  - Implementing permission systems where each permission is represented by a bit in an integer
  - Performing image processing operations on pixel data
  - Implementing cryptographic algorithms that require bit-level operations

PHP Array Operators

- PHP provides several array operators to perform operations on arrays. The following are the array operators in PHP:
  - Union (+) - combines two arrays into one array. If there are duplicate keys, the values from the first array are retained. Example

```
$array1 = array("a" => "Apple", "b" => "Banana");
$array2 = array("b" => "Orange", "c" => "Grapes");
$result = $array1 + $array2;
print_r($result);
```

This will output:

```
Array
(
    [a] => Apple
    [b] => Banana
    [c] => Grapes
)
```

Also

```
$array1 = array("Apple", "Banana");
$array2 = array("Orange", "Grapes");
$result = $array1 + $array2;
print_r($result);
```

This will output:

```
Array
(
    [0] => Apple
    [1] => Banana
)
```

- Equality (==) - returns true if two arrays have the same key-value pairs, regardless of the order of the elements and their types. Keys and Elements are compared. Example

```php
$array1 = array("a" => "Apple", "b" => "Banana");
$array2 = array("b" => "Banana", "a" => "Apple");
var_dump($array1 == $array2); // Output: bool(true)
```

- Identity (===) - returns true if two arrays have the same key-value pairs in the same order and of the same types. Keys and Elements are compared. Example

```php
$array1 = array("a" => "Apple", "b" => "Banana");
$array2 = array("b" => "Banana", "a" => "Apple");
var_dump($array1 === $array2); // Output: bool(false)
```

- Inequality (!= or <>) - returns true if two arrays do not have the same key-value pairs. Example

```php
$array1 = array("a" => "Apple", "b" => "Banana");
$array2 = array("a" => "Apple", "b" => "Orange");
var_dump($array1 != $array2); // Output: bool(true)
```

- Non-identity (!==) - returns true if two arrays do not have the same key-value pairs in the same order or of the same types. Example

```php
$array1 = array("a" => "Apple", "b" => "Banana");
$array2 = array("a" => "Apple", "b" => "Banana");
var_dump($array1 !== $array2); // Output: bool(false)
```

PHP Control Structures

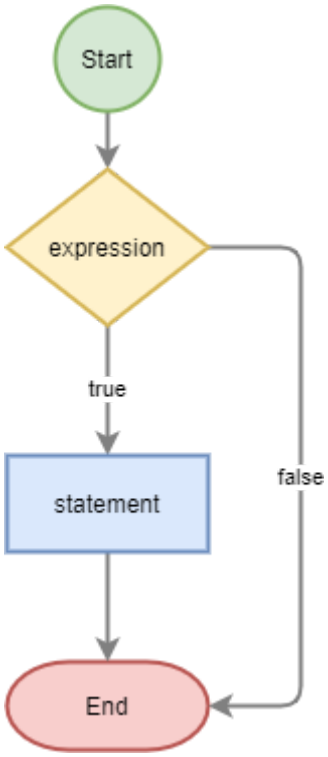
- A control structure is a block of code that determines the flow of execution based on certain conditions. It allows you to control the order in which statements are executed in your program.

PHP If Statement

- The `if` statement is a conditional statement that allows you to execute a block of code only if a specified condition or a logical expression is true. The syntax of the `if` statement in PHP is as follows:

```
if(expression) {
    // code to be executed if the expression is true
}
```

- Here, the `expression` is a boolean expression that evaluates to either true or false. It is executed first. If the expression is true, the code block inside the curly braces `{}` is executed. If the expression is false, the code block is skipped and the program continues with the next statement after the `if` block.



- The below diagram illustrates the flow of control in an `if` statement
- If you want to execute only a single statement when the condition is true, you can omit the curly braces. Example

```
$age = 18;
if ( $age >= 18 )
    echo "You are an adult.";
```

But still, it's a good practice to always use curly braces with the if statement even though it has a single statement to execute

Nesting If Statements

- This can be done as follows:

```
if ( condition1 ) {
    // code to be executed if condition1 is true
```

```
    if ( condition2 ) {
        // code to be executed if condition2 is true
    }
}
```

Embedded If Statements

- PHP provieds an alternative syntax for embedding `if` statements within HTML code. This syntax is particularly useful when you want to mix PHP code with HTML markup. The syntax for embedded `if` statements is as follows:

```
<?php if ( expression ): ?>
    <!-- HTML code to be executed if the expression is true -->
<?php endif; ?>
```

- If you require an `else` block, you can use the following syntax:

```
<?php if ( expression ): ?>
    <!-- HTML code to be executed if the expression is true -->
<?php else: ?>
    <!-- HTML code to be executed if the expression is false -->
<?php endif; ?>
```

- If you require an `elseif` block, you can use the following syntax:

```
<?php if ( expression1 ): ?>
    <!-- HTML code to be executed if expression1 is true -->
<?php elseif ( expression2 ): ?>
    <!-- HTML code to be executed if expression2 is true -->
<?php else: ?>
    <!-- HTML code to be executed if both expression1 and expression2 are false --
>
<?php endif; ?>
```

- Notice that the `if`, `elseif`, `else`, and `endif` keywords are used to define the structure of the embedded `if` statement. Also, they are written in a different element. The colon (`:`) is used to indicate the start of the code block, and the `endif;` keyword is used to close the `if` statement.
- After all conditions have been checked, the `endif;` keyword is used to close the `if` statement.

- Avoid the mistake of using curly braces `{}` with the embedded `if` statement.
- Also, avoid the use of the assignment operator (`=`) in place of the equality operator (`==`) in the expression of an `if` statement.

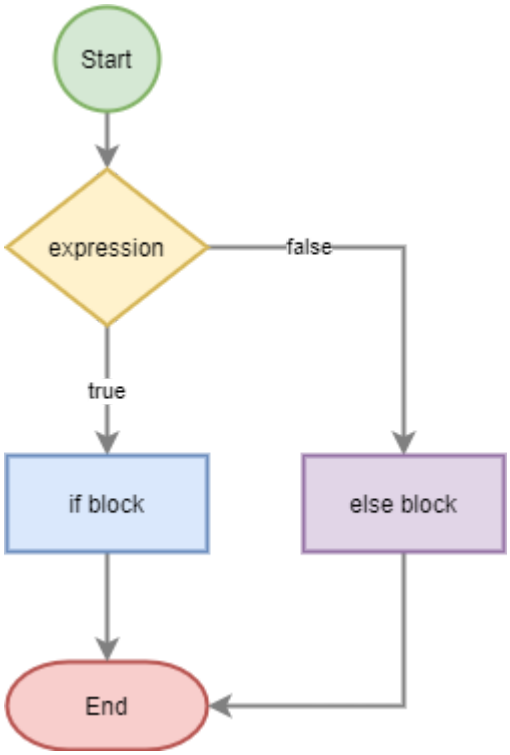
PHP If...Else Statement

- Suppose you want to execute a block of code when a condition is true and another block of code when the condition is false. You can use the `if...else` statement to achieve this. The syntax of the `if...else` statement in PHP is as follows:

```
if ( expression ) {
    // code to be executed if the expression is true
} else {
    // code to be executed if the expression is false
}
```

Here, if the expression is true, PHP executes the if block. Otherwise, it executes the else block.

- The flow of control in an `if...else` statement is illustrated below



PHP If...else statement in HTML

- Similar to the embedded `if` statement, you can also embed the `if...else` statement within HTML code using the following syntax:

```
<?php if (expression): ?>
// HTML code to be executed if expression is true
<?php else:>
// HTML code to be executed if expression is false
<?php endif ?> //super important - ends the conditonal block
```

- Note that you don't need to place a semicolon (😉 after the endif keyword because the endif is the last statement in the PHP block. The enclosing tag ?> automatically implies a semicolon

- An examples is as follows

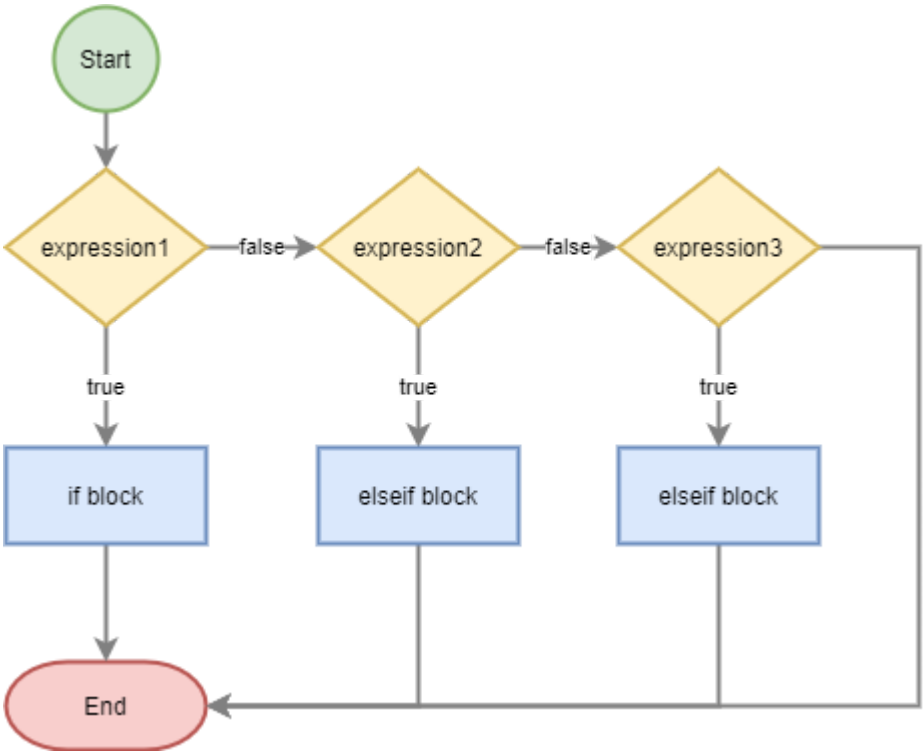
```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <meta />
    <title>Login</title>
  </head>
  <body>
    <?php $is_authenticated = true ?> <?php if ($is_authenticated): ?>
    <h1>Welcome, again!</h1>
    <a href="#">Log out</a>
    <?php else: ?>
    <h1>Welcome! Log in to your account</h1>
    <a href="#">Login</a>
    <?php endif ?>
  </body>
</html>
```

PHP If elseif statement

- The PHP If elseif statement is particularly useful when you have multiple conditions to check. It allows you to test several conditions sequentially and execute different blocks of code based on which condition is true. The syntax of the `if...elseif...else` statement in PHP is as follows:

```
if ( expression1 ) {
    // code to be executed if expression1 is true
} elseif ( expression2 ) {
    // code to be executed if expression2 is true
} else {
    // code to be executed if both expression1 and expression2 are false
}
```

This illustrates how the If elseif statement works



- Here, one a certain if block is executed, the rest are skipped even thought their expression is `true`.

PHP if elseif alternative syntax

- PHP supposrts an alternative syntax for the `elseif` statement. This involves exempting the curly braces and using colons (`:`) to indicate the start of each code block. The syntax is as follows:

```
if (expression1):
// code to be executed
elseif (expression2):
// code to be executed
elseif (expression3):
// code to be executed
else:
// code to be executed
endif;
```

- Colons indicates the start of each `if` statements
- Use the `endif` statement followed by a semi-colon to terminate the final conditional block (either the `else` or `elseif` block)
- This is similar to writing `if..else` statements in Python
- Now `elseif` and `else if` are the same in PHP. However, `elseif` is preferred as it is a single word and more concise. Also `elseif` should be used when embedding PHP within HTML. Also, if you decide to use the `else if` in the alternative syntax it yields an error. Example

```
<?php
```

```php
$x = 10;
$y = 20;

if ($x > $y) :
    echo 'x is greater than y';
else if ($x < $y):
    echo 'x is equal to y';
else:
    echo 'x is less than y';
endif;
```

The above code snipett yields an error. To fix it, use `elseif` instead of `else if` - remove the space between else and if.

- Use the elseif whenever possible to make your code more consistent.

PHP Ternary Operator

- The ternary operator is a shorthand way of writing an `if...else` statement. It is called the ternary operator because it takes three operands: a condition, a value to return if the condition is true, and a value to return if the condition is false. The syntax of the ternary operator in PHP is similar to that in JavaScript and is as follows:

```
condition ? value_if_true : value_if_false;
```

- Conside the following example

```php
$render = true
?
<<< TEXT
<h1>Welcome!</h1>
<a href="#">LogOut</a>
TEXT
:
<<< TEXT
<h1>Hello</h1>
<a href="#">Login</a>
TEXT;
echo $render;
```

Here, if the condition `$render` is true, the value of the first block (the HTML code for welcome message and logout link) is assigned to the variable `$render`. Otherwise, the value of the second block (the HTML code for hello message and login link) is assigned to `$render`. Finally, the value of `$render` is echoed.

- If an unassigned variable is used in the condition, it is treated as null which by type juggling gets converted to `null`.

The shorthand ternary operator

- From PHP 5.3 onwards, you can use the shorthand ternary operator (also known as the Elvis operator) to simplify the ternary operation when the value to return if the condition is true is the same as the condition itself. The syntax of the shorthand ternary operator is as follows:

```
$result = condition ?: value_if_false;
```

- Here is `condition` is `false`, the `$result` variable is assigned the `value_if_false`. If it is `true`, the `$result` variable is assigned the value produced by the `condition`
- The following example uses the shorthand ternary operator to assign the value of the `$path` to the `$url` if the $path is not empty. If the `$path` is empty, the ternary operator assigns the literal string '`/`' to the `$url`

```
$path = "\about"

$url = $path ?: "\"
echo $url; // Output: \about
```

- If `$path` is an empty string, null, false, 0 or not assigned a value, the output will be `\` instead.

Chaining Ternary Operators

- Tenary operators can be chained together to evaluate multiple conditions in a single expression. This is useful when you have several possible outcomes based on different conditions. In this case, each ternary operators are nested using parentheses `()`. The syntax for chaining (3) ternary operators is as follows:

```
condition1 ? value_if_true1 :
    ( condition2 ? value_if_true2 :
        ( condition3 ? value_if_true3 :
            value_if_false ) );
```

- Most times, chaining ternary operators can make your code hard to read and maintain. Thus, it is advisable to use `if...elseif...else` statements instead of chaining ternary operators when you have multiple conditions to check.
- However, if you still want to use chained ternary operators, ensure to use parentheses and indentations to clearly define the order of evaluation and improve readability.

PHP Switch

- The switch statement is particularly useful when the value of a variable needs to be compared against (or match to) multiple possible values in a boolean expression. It provides a more concise and readable

way to handle such scenarios compared to using multiple `if...elseif...else` statements.

- The syntax of the switch statement in PHP is as follows:

```
switch(expression){
  case value1:
    // code block 1;
    break;
  case value2:
    // code block 2;
    break;
  default:
    // code to be executed when expression don't fit any of the cases
}
```

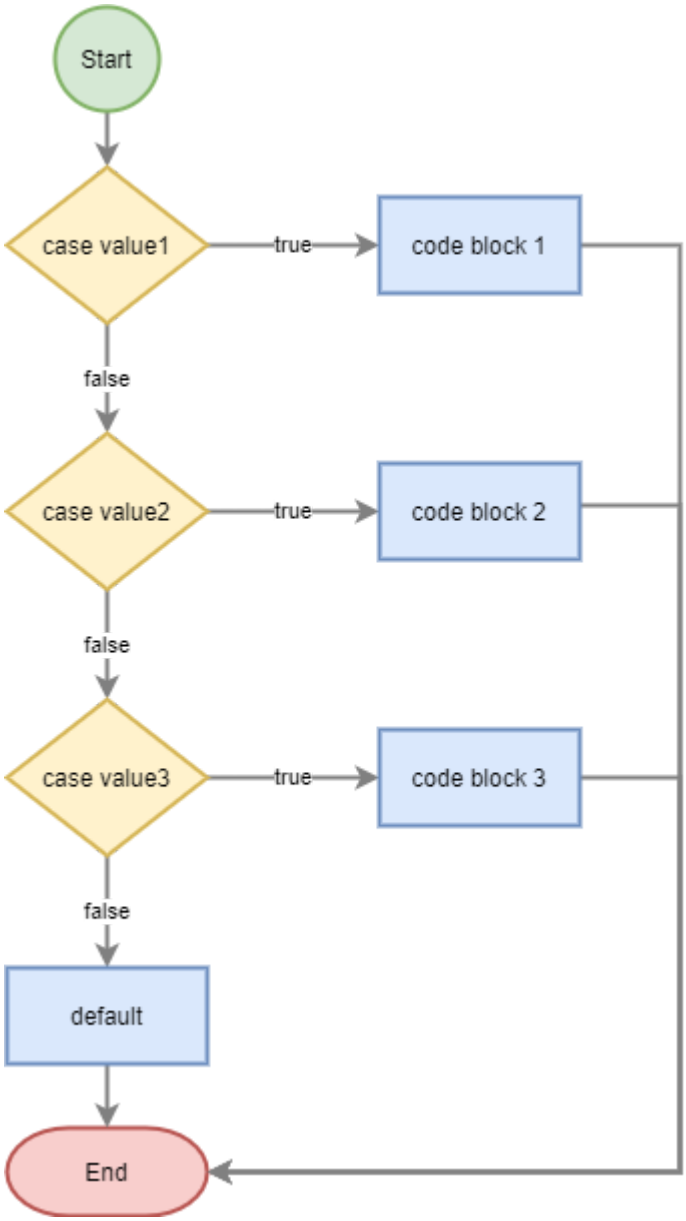
- It's alternative syntax

```
switch(expression):
  case value1:
    // code block 1;
    break;
  case value2:
    // code block 2;
    break;
  default:
    // code to be executed when expression don't fit any of the cases
endswitch;
```

is used when embedding PHP in HTML. Do not forget the `endswitch;` statement to terminate the switch block.

- Here, the `expression` is evaluated once and its value is compared against the values specified in each `case`. If a match is found, the corresponding code block is executed. The `break` statement is used to exit the switch statement after executing a case block. If no match is found, the code block under the `default` case is executed (if provided). If the `default` case is omitted and no match is found, no code block is executed.

- The flow of control in a switch statement is illustrated below



  Here, once a match is found, the corresponding case block is executed until a break statement is encountered. If no `break` statement is found, the execution will continue to the next case block(s) until a break is encountered or the switch statement ends. If no match is found, the default block is executed (if provided).

- Scenario: Suppose that you're building a website whose users have many roles like admin, editor, author, and subscriber. You can basically use an if...elseif...else statement to check the user role and display the appropriate content as

```
$role = 'editor';
$message = '';

if ($role === 'editor') {
    $message = 'Welcome! You have some pending articles to edit';
} elseif ($role === 'author') {
    $message = 'Welcome! Do you want to publish the draft article?';
} elseif ($role === 'admin') {
        $message = 'Welcome, admin!';
```

```
    } elseif ($role === 'subscriber') {
            $message = 'Welcome! Check out some new articles';
}

echo <<<TEXT
<h1>$message</h1>
TEXT;
```

- Instead of using the if...elseif...else statement, you can use the switch statement to achieve the same result more concisely as follows:

```
$role = 'editor';
$message = '';

switch ($role):
    case 'editor':
        $message = 'Welcome! You have some pending articles to edit';
        break;
    case 'author':
        $message = 'Welcome! Do you want to publish the draft article?';
        break;
    case 'admin':
         $message = 'Welcome, admin!';
         break;
    case 'subscriber':
        $message = 'Welcome! Check out some new articles';
         break;
    default:
        $message = 'Unauthorized!';
    endswitch;
```

or using the standard syntax

```
$role = 'editor';
$message = '';

switch ($role) {
    case 'editor':
        $message = 'Welcome! You have some pending articles to edit';
        break;
    case 'author':
        $message = 'Welcome! Do you want to publish the draft article?';
        break;
    case 'admin':
         $message = 'Welcome, admin!';
         break;
    case 'subscriber':
        $message = 'Welcome! Check out some new articles';
         break;
```

```
    default:
        $message = 'Unauthorized!';
}
```

Combining cases

- Since PHP executes a certain case block when a match is found till it encounters a break statement, you can combine multiple cases that share the same code block. Example: Suppose both the editor and author roles share the same welcome message. You can combine their cases as follows:

```
switch ($role):
    case 'editor':
    case 'author':
        $message = 'Welcome! Do you want to publish the draft article?';
        break;
    case 'admin':
         $message = 'Welcome, admin!';
         break;
    case 'subscriber':
        $message = 'Welcome! Check out some new articles';
         break;
    default:
        $message = 'Unauthorized!';
    endswitch;
```

By this, if the `$role` is either `editor` or `author`, the same welcome message will be assigned to the `$message` variable.

- Switch statements use loose comparison (==) when comparing the expression with the case values. This means that type juggling occurs during the comparison. By this, it does Type Juggling.

Using continue statement in switch

- In PHP 7.3 and later versions, you can use the `continue` statement within a switch statement to skip the current case and move to the next iteration of an enclosing loop. This is particularly useful when you want to skip certain cases based on specific conditions while iterating through a loop. Example:

```
for ($i = 0; $i < 5; $i++) {
    switch ($i) {
        case 1:
            echo "Skipping case 1\n";
            continue 2; // Skip to the next iteration of the for loop
        case 2:
            echo "Processing case 2\n";
            break;
        case 3:
            echo "Processing case 3\n";
            break;
```

```
            default:
                echo "Processing default case\n";
        }
    }
    /*
    Output:
    Processing default case
    Skipping case 1
    Processing case 2
    Processing case 3
    Processing default case
    */
```

switch() vs if...elseif...else

- The expression between the switch parentheses `switch(expression)` can only be a executed once while in the `if...elseif...else` statement, each condition is evaluated separately. Thus, if you have multiple conditions that depend on the same variable or expression, using a switch statement can be more efficient.
- Consider this example:

```
function x() {
    sleep(3);
    echo 'Done';
    return 4;
}

if(x() === 1) {
    echo 1;
} elseif(x() === 2) {
    echo 2;
} elseif(x() === 3) {
    echo 3;
} else {
    echo 4;
}
```

How this works is that the function `x()` is called multiple times (3 times) until a match is found or all conditions are evaluated. Each time the function is called, it sleeps for 3 seconds before returning a value. Thus, if no match is found, the total sleep time will be 9 seconds. To fix and optimize this, you can use a switch statement as follows:

```
function x() {
    sleep(3);
    echo 'Done';
    return 4;
}
```

```
switch(x()) {
    case 1:
        echo 1;
        break;
    case 2:
        echo 2;
        break;
    case 3:
        echo 3;
        break;
    default:
        echo 4;
}
```

By this, the function `x()` is called only once, and its return value is compared against the case values. Thus, the total sleep time will be only 3 seconds regardless of whether a match is found or not.

PHP sleep() Function

- The `sleep()` function in PHP is used to pause the execution of a script for a specified number of seconds. It is commonly used to introduce delays in the execution of a program, which can be useful in various scenarios such as rate limiting, simulating long-running processes, or waiting for external resources to become available. The syntax of the `sleep()` function is as follows:

```
sleep(seconds);
```

PHP Match Expression

- The `match` expression is a new feature introduced in PHP 8.0 that provides a more concise and expressive way to perform value comparisons and return results based on those comparisons. It is similar to the `switch` statement but has some key differences and advantages. The syntax of the `match` expression in PHP is as follows:

```
match (expression) {
    value1 => result1,
    value2 => result2,
    ...
    default => default_result,
};
```

- Here, the `expression` is evaluated once, and its value is compared against the specified values (`value1`, `value2`, etc.). These values can also be logical expressions that yields a value. If a match is found, the corresponding result (`result1`, `result2`, etc.) is returned. This result can also be any complex expression. If no match is found, the `default` result is returned (if provided). Unlike the

`switch` statement, the `match` expression uses strict comparison (`===`) for value comparisons, which means that both the value and type must match for a case to be considered a match.

- A use-case of the match expression is as follows:

```
match($role) {
    'admin' => 'Welcome, admin!',
    'editor' => 'Welcome! You have some pending articles to edit',
    'author' => 'Welcome! Do you want to publish the draft article?',
    'subscriber' => 'Welcome! Check out some new articles',
    default => 'Unauthorized!',
};
```

Since the match expression returns a value, you can directly assign its result to a variable as follows:

```
$message = match($role) {
    'admin' => 'Welcome, admin!',
    'editor' => 'Welcome! You have some pending articles to edit',
    'author' => 'Welcome! Do you want to publish the draft article?',
    'subscriber' => 'Welcome! Check out some new articles',
    default => 'Unauthorized!',
};
```

- PHP match expression has the following features:
  - It uses strict comparison (`===`) for value comparisons.
  - It returns a value, which can be directly assigned to a variable.
  - It does not require `break` statements to prevent fall-through behavior.
  - Each case must result in a value; you cannot have code blocks like in `switch` statements.

Match expression vs Switch statement

- Match expressions and switch statements are both used for conditional branching based on the value of an expression. However, there are some key differences between the two:
  - Comparison Type: Match expressions use strict comparison (`===`), while switch statements use loose comparison (`==`).
  - Return Value: Match expressions return a value, which can be directly assigned to a variable. Switch statements do not return a value.
  - Fall-through Behavior: Match expressions do not have fall-through behavior; each case must result in a value. Switch statements can have fall-through behavior if `break` statements are omitted. Suppose you have two cases that share the same code this can be done using match expression as follows:

```
$result = match($value) {
    'case1', 'case2' => 'Shared result for case1 and case2',
    'case3' => 'Result for case3',
```

```
        default => 'Default result',
    };
```

- Syntax: Match expressions have a more concise syntax compared to switch statements.
- Match expression is exhaustive while Switch statement is not. This means that all possible cases must be handled, either explicitly or through a default case. If a match expression does not cover all possible values and lacks a default case, it will result in an error at runtime.
- Switch statement takes multiple statements or code block for each case. For Match expression, each case must result in a single value.

PHP Loops

- A Loop is a control structure that allows you to execute a block of code repeatedly based on a specified condition. Loops are useful when you want to perform repetitive tasks or iterate over a collection of data. PHP provides several types of loops, including `for`, `while`, and `do...while` loops.

PHP for

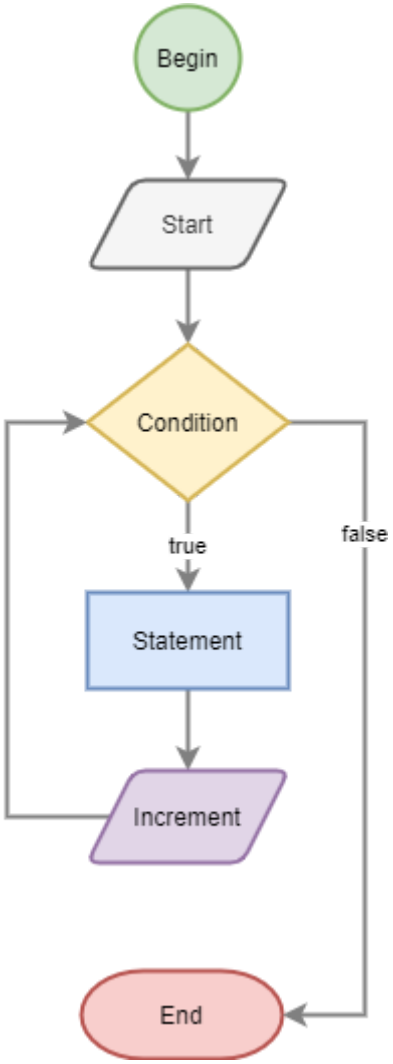
- The `for` statement allows you to execute a block of code repeatedly for a specified number of times. It is commonly used when the number of iterations is known beforehand. The syntax of the `for` statement in PHP is as follows:

```
for (start; condition; increment) {

}
```

- `start` statement is executed when the loop starts only not at each iteration. After that, it is never executed again.
- `condition` is mostly a boolean expression or value evaluated at the beginning of each iteration . If its `true`, the code in the block gets executed
- the `increment` expression is expressed at the end of each iteration
- In PHP, you can add multiple `start`, `condition` and `increment` expressions or just any expression by separating them with commas. Their execution will follow how `start`, `condition` and `increment` expressions are written. Example

```
for ( $i = 0, $j = 10; $i < 10, $j > 0; $i++, $j-- ) {
    echo "i: $i, j: $j\n";
}
```

This is mostly when you want to control multiple variables in a single `for` loop.



- The following illustrates the flow of control in a `for` loop
- Also, you can leave any of the three expressions (`start`, `condition`, `increment`) empty. If this happens, use the `break` statement to exit or terminate the loop at some point Example

```
for(;;) {
    // infinite loop
    // use break statement to exit the loop
    if ( some_condition ) {
        break;
    }
}
```

- Notice that even though `condition`, `start` and `increment` expressions are optional, the semicolons (`;`) separating them are mandatory.
- Also, don't forget to use the `break` statement to exit the loop when the `condition` expression is omitted; otherwise, it will result in an infinite loop.
- The increment operator `++` can be used rather than explicitly adding 1 to a variable. Example

```
for ( $i = 0; $i < 10; $i++ ) {
    echo "i: $i\n";
}
```

instead of

```
for ( $i = 0; $i < 10; $i += 1 ) {
    echo "i: $i\n";
}
```

`for` Loop performance issue

- Suppose you have a large array and you want to iterate over its elements using a `for` loop. If you use the `count()` function in the `condition` expression of the `for` loop, it will be called on every iteration, which can lead to performance issues. Example

```
$large_array = [/* large array elements */];

for($i = 0; $i < count($large_array); $i++) {
    // code to be executed
}
```

This can be inefficient because the `count()` function is called on every iteration of the loop. To optimize this, you can store the result of the `count()` function in a variable alongside the `start` statement before the loop starts and use that variable in the `condition` expression. Example

```
$large_array = [/* large array elements */];

for($i = 0, $len = count($large_array); $i < $len; $i++) {
    // code to be executed
}
```

Or you can store the length of the array in a variable before the loop starts as follows:

```
$large_array = [/* large array elements */];
$len = count($large_array);
for($i = 0; $i < $len; $i++) {
    // code to be executed
}
```

In practice, do not call any function in the `condition` expression of a loop if it can be avoided. Instead, store the result of the function in a variable before the loop starts and use that variable in the `condition` expression. This will improve the performance of your code, especially when dealing with large datasets.

PHP while

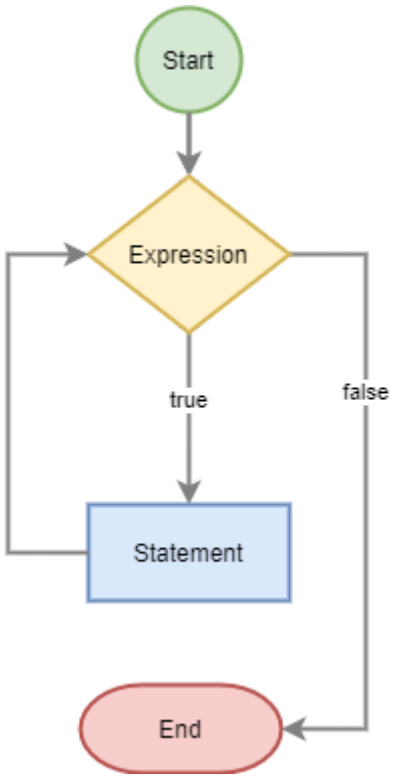
- The `while`statement executes a block of code as long as a specified condition (a boolean) is true. It is commonly used when the number of iterations is not known beforehand and depends on a certain condition being met. The syntax of the `while` statement in PHP is as follows:

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

- How it works
    - The `condition` is evaluated first. If it is true, the code block inside the curly braces `{}` is executed. After executing the code block, the `condition` is evaluated again. This process continues until the `condition` evaluates to false. When the `condition` is false, the loop terminates, and the program continues with the next statement after the `while` block.
- Since PHP evaluaates `condition` before executing the code block, the `while` loop is known as an Pretest loop.
- The `while` loop doesn't require curly braces `{}` if it contains only a single statement. Example

```
while (condition)
    // single statement to be executed
```

However, it's a good practice to always use curly braces `{}` with the `while` loop, even if it has a single statement to execute, to improve code readability and maintainability.



- The following illustrates the flow of control in a `while` loop

An alternative syntax for `while` loop

- An alternative syntax for the `while` loop is provided in PHP, which is particularly useful when embedding PHP code within HTML. This syntax uses a colon (`:`) to indicate the start of the loop block and the `endwhile;` statement to terminate the loop. The syntax is as follows:

```
while ( condition ):
    // code to be executed
endwhile;
```

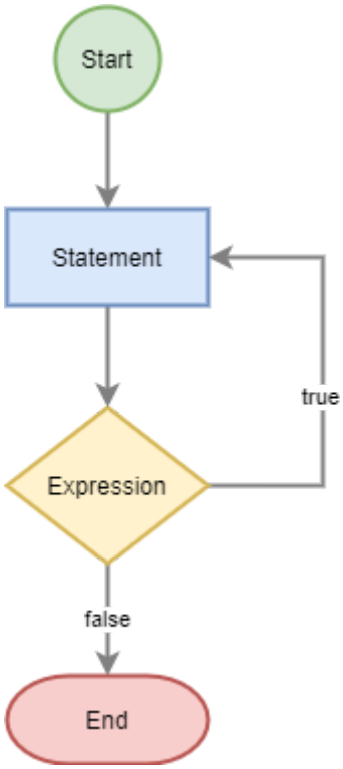
PHP do...while

- The `do...while` statement is similar to the `while` statement, but with a key difference: the code block is executed at least once before the condition is evaluated. This means that the code block will always run at least one time, regardless of whether the condition is true or false. The syntax of the `do...while` statement in PHP is as follows:

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

Here, the code is executed first before the `expression` is evaluated. If the `expression` evaluates to true, the code block is executed again. This process continues until the `expression` evaluates to false. When the `expression` is false, the loop terminates, and the program continues with the next statement after the `do...while` block.

- The semi-colon (`;`) after the `while` expression is mandatory in a `do...while` loop.



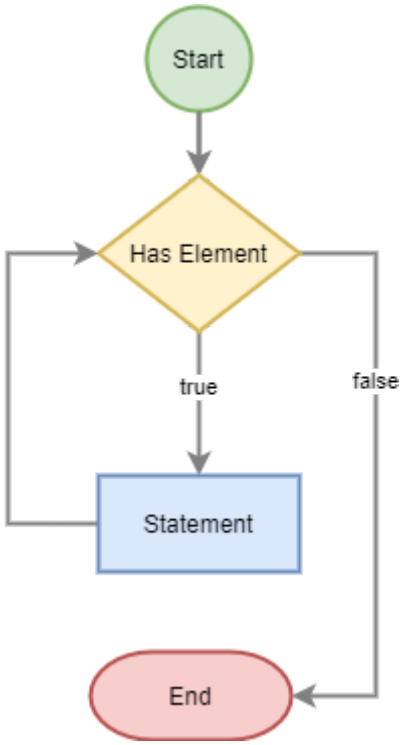
- The illustration of the flow of control in a `do...while` loop is as follows

do...while vs while

- PHP executes the code in the `do...while` loop at least once, even if the condition is false from the beginning. In contrast, the `while` loop may not execute the code block at all if the condition is false initially.
- The expression is evaluated at the end of each iteration in the `do...while` loop, while it is evaluated at the beginning of each iteration in the `while` loop.

PHP foreach

- The `foreach` statement is specifically designed for iterating over elements in arrays or objects. It provides a convenient way to loop through each element in an array or each property in an object without the need for manual indexing.



- This illustrates the flow of control in a `foreach` loop

PHP foreach with indexed arrays

Syntax:

```
foreach ( $array_name as $element ) {
    // code to be executed
}
```

- Here, if PHP encounters a `foreach` loop, it assigns the first element of the array to the variable `$element` and executes the code block. After executing the code block, it assigns the next element of the array to `$element` and executes the code block again. This process continues until all elements in the array have been processed. Then at the last element, the loop terminates, and the program continues with the next statement after the `foreach` block.

PHP foreach with associative arrays

- Syntax:

```
foreach($array_name as $key => $value) {
  //code to be executed
}
```

- Here, when PHP executes the `foreach` loop, it assigns the first key of the associative array to the variable `$key` and its corresponding value to the variable `$value`. Then, it executes the code block. After executing the code block, it assigns the next key to `$key` and its corresponding value to `$value`, and executes the code block again. This process continues until all key-value pairs in the associative array have been processed. Finally, when the last key-value pair is processed, the loop terminates, and the program continues with the next statement after the `foreach` block.

Manipulating array elements using `foreach`

- To manipulate array elements using the `foreach` loop, you can use the reference operator (`&`) before the value variable in the loop. This allows you to modify the actual elements of the array directly within the loop. Example

```
$numbers = [1, 2, 3, 4, 5];

foreach ($numbers as &$number) {
    $number *= 2; // Double the value of each element
}

print_r($numbers);
/*Output:
Array
(
    [0] => 2
    [1] => 4
    [2] => 6
    [3] => 8
    [4] => 10
)
*/
```

`foreach()` Behavior with empty arrays

- If the array is empty, the code block inside the `foreach` loop will not be executed at all. Example

Some `foreach()` Behaviours

- The variable used to hold the value in a `foreach` loop retains its value after the loop ends. Example

```
$names = ['John', 'Isaac', 'Gabriel'];

foreach($names as $name) {
    echo $name;
```

```
}

echo $name; // Output: Gabriel
```

- If you are using the `foreach()` loop to manipulate the array element by reference and the variable is assigned immediately after the `foreach` loop, its will overwrite the value of the last element processed in the loop. Example

```
$names = ['John', 'Isaac', 'Gabriel'];

foreach($names as &$name) {
    echo $name;
}

$name = 'Michael';

print_r($names);
/*Output:
Array
(
    [0] => John
    [1] => Isaac
    [2] => Michael
)
```

- To avoid this issue, you can unset the reference variable after the `foreach` loop as follows:

```
$names = ['John', 'Isaac', 'Gabriel'];
foreach($names as &$name) {
    echo $name;
}
unset($name); // Unset the reference variable
$name = 'Michael';
print_r($names);
/*Output:
Array
(
    [0] => John
    [1] => Isaac
    [2] => Gabriel
)
*/
```

PHP `implode()` function

- The `implode()` function in PHP is used to join array elements into a single string. It takes two parameters: a separator string and an array. The separator string is placed between each element of the

array in the resulting string. The syntax of the `implode()` function is as follows:

```
implode(separator, array);
```

- Here, the `separator` is a string that will be used to separate the elements in the resulting string, and `array` is the array whose elements you want to join together.
- An example of using the `implode()` function is as follows:

```
$fruits = ['Apple', 'Banana', 'Orange'];
$result = implode(', ', $fruits);

echo 'The fruits are: ' . $result;
// Output: The fruits are: Apple, Banana, Orange
```

- For best practices, always check if the input variable is an array before using the `implode()` function to avoid potential errors.

PHP `json_encode()` function

- The `json_encode()` function in PHP is used to convert a PHP variable (such as an array or object) into a JSON (JavaScript Object Notation) string. JSON is a lightweight data interchange format that is easy for humans to read and write, and easy for machines to parse and generate. The syntax of the `json_encode()` function is as follows:

```
json_encode(value, options, depth);
```

- Here, the `value` is the PHP variable that you want to convert to a JSON string. The `options` parameter is optional and allows you to specify various options for encoding, such as pretty-printing the JSON output. The `depth` parameter is also optional and specifies the maximum depth of the nested structures to be encoded.
- An example of using the `json_encode()` function is as follows:

```
$data = [
    'name' => 'John Doe',
    'age' => 30,
    'city' => 'New York'
];
$json_string = json_encode($data, JSON_PRETTY_PRINT);
echo $json_string;
/* Output:
{
    "name": "John Doe",
    "age": 30,
    "city": "New York"
```

```
}
*/
```

PHP `break` statement

- The `break` statement terminates the execution of the `for`, `do...while`, `while`, and `switch` statement
- Practically, we use the `break` statement alongside the `if` statement to specify the condition for the termination of the loop
- Also, by default, the `break` statement terminates only the innermost loop or switch statement in which it is placed.

```
if (condition){
  break
}
```

PHP `break` statement with nested loops

- When using nested loops (a loop inside another loop), the `break` statement only terminates the innermost loop in which it is placed. The outer loop continues to execute as normal. Example

```
for ($i = 1; $i <= 3; $i++) {
    echo "Outer loop iteration: $i\n";
    for ($j = 1; $j <= 5; $j++) {
        if ($j == 3) {
            break; // This will break only the inner loop
        }
        echo "  Inner loop iteration: $j\n";
    }
}
// Output:
// Outer loop iteration: 1
//   Inner loop iteration: 1
//   Inner loop iteration: 2
// Outer loop iteration: 2
//   Inner loop iteration: 1
//   Inner loop iteration: 2
// Outer loop iteration: 3
//   Inner loop iteration: 1
//   Inner loop iteration: 2
```

- In this example, when `$j` equals 3, the `break` statement terminates the inner loop. The outer loop continues to the next iteration.
- Now, to break out the both the outer and inner loops, you can specify the number of levels to break out of by providing an optional numeric argument to the `break` statement. Example

```
for ($i = 1; $i <= 3; $i++) {
    echo "Outer loop iteration: $i\n";
    for ($j = 1; $j <= 5; $j++) {
        if ($j == 3) {
            break 2; // This will break out of both the inner and outer loops
        }
        echo "  Inner loop iteration: $j\n";
    }
}
```

- Here, the `break 2;` statement terminates both the inner and outer loops when `$j` equals 3.
- Basically, the numeric argument specifies how many nested levels of loops to break out of. If no argument is provided, it defaults to 1, meaning only the innermost loop is terminated.

- Consider this a similar example with switch statement

```
$grade = 'B';
while (true) {
  switch ($grade) {
      case 'A':
          echo "Excellent!";
          break 2; // This will break out of the switch statement and any
enclosing loop (if present) - here, the while loop
      case 'B':
          echo "Good job!";
          break;
      case 'C':
          echo "You passed.";
          break;
      default:
          echo "Invalid grade.";
  }
}
```

- Consider another similar example with nested switch statements

```
$level1 = 'A';
$level2 = 'B';
switch ($level1) {
    case 'A':
        echo "Level 1: A\n";
        switch ($level2) {
            case 'A':
                echo "Level 2: A\n";
                break 2; // This will break out of both switch statements
            case 'B':
                echo "Level 2: B\n";
                break;
```

```
            default:
                echo "Level 2: Invalid\n";
        }
        break;
    case 'B':
        echo "Level 1: B\n";
        break;
    default:
        echo "Level 1: Invalid\n";
}
```

- Consider another example with nested loops

```
for ($i = 0; $i < 6; $i++) {
    for ($j = 0; $j < 4; $j++) {
        if ($j === 3 || $i === 3) {
            echo "Coordinate containing 3 has been found! Exiting inner loop
now...<br>";
            break;
        }

        echo "($i,$j) <br>";
    }
}
```

PHP `continue` Statement

- The `continue` statement is used to skip the current iteration of a loop and move to the next iteration. It is commonly used when you want to skip certain iterations based on a specific condition.
- It is used to skip all remaining statement that follow it in the current iteration of a loop and proceed to the next iteration of the loop.
- The syntax of the `continue` statement in PHP is as follows:

```
if (condition) {
    continue;
}
```

PHP Functions

- A function is a named block of code that is reusable and performs a specific task. Functions are used to organize code into reusable modules, making it easier to read, maintain, and debug. In PHP, functions are defined using the `function` keyword followed by the function name and parentheses `()`. The syntax for defining a function in PHP is as follows:

```
function function_name() {
    // code to be executed
}
```

- In this syntax:
    - First, specify the function name followed by the function keyword. The function's name must start with a letter or underscore followed by zero or more letters, underscores, and digits.
    - Second, define one or more statements inside the function body. The function body starts with the `{` and ends with `}`.
- To call or invoke a function, you simply use the function name followed by parentheses `()`. That is

```
function_name();
```

- This is only appropriate when a function has no parameter

PHP Function Parameters

- Practically, functions often require input values to perform their tasks. These input values are called parameters (or arguments). You can define parameters in the function definition by specifying them inside the parentheses `()`. Example

```
function function_name($parameter_1, $parameter_2, ...) {

}
```

- In the function body, you can use the parameters as variables to perform operations or calculations. When calling the function, you provide the actual values (arguments) for the parameters. In this function body, paramters are called local variables because they are only accessible within the function.
- Scenario: Suppose you want to create a function that calculates the sum of two numbers. You can define the function with two parameters as follows:

```
function sum ($x, $y) {
  echo $x + $y;
}
```

- Now, when calling a function with a parameter, you simply need to specify an argument which is more of the actual value to be worked on with the the function body. To call the above `sum` function, we do this

```
sum(1, 2); //Output: 3
```

- In the above code, the value of `1` is assigned to the parameter `$x`, and the value of `2` is assigned to the parameter `$y`. The function then calculates the sum of `$x` and `$y` and echoes the result.

Parameters vs Arguments

- Parameters are simply the local variables (or dummy variables) specified when defining a function.
- Arguments, on the other hand, are the actual values passed to the function when calling it.

PHP `return` Statement

- Basically, functions can return values. To do this, we use the `return` statement. The `return` statement is used to specify the value that a function should return to the caller. When PHP encounters a `return` statement within a function, it immediately exits the function and sends the specified value back to the point where the function was called.
- This value can be literarily any valid data type in PHP, such as integers, strings, arrays, objects, etc. It can even be an expression that evaluates to a value.
- At default, functions return `NULL` if no `return` statement is specified.

Function Declaration

- Normally, in PHP a function can be defined before call or after call. Exceptions to this includes: + Declaring functions conditionally - If a function is to be declared conditionally and it is called before declaration, it will result in an error + Inner functions - Suppose you declare a function inside another function (nested function), the inner function will not be available until the outer function is called. Thus, if you try to call the inner function before calling the outer function, it will result in an error. Practically, do not declare functions conditionally or as inner functions unless necessary to avoid such errors.
- Other things to know regarding function declaration
    - Function names are case-insensitive. This means that you can call a function using different cases (uppercase or lowercase) without any issues.
    - You cannot redeclare a function with the same name in the same scope. If you try to do so, it will result in a fatal error.
    - Functions can be declared inside other functions (nested functions). However, the inner function will only be accessible within the scope of the outer function.

HTML Code inside a function

- You can include HTML code inside a PHP function using echo or heredoc syntax. Example using echo

```
function displayGreeting($name) {
    echo "<h1>Hello, $name!</h1>";
    echo "<p>Welcome to our website.</p>";
}
```

- Or you can follow this syntax

```
<?php function function_name ?>
  <!-- HTML code here -->
<?php endfunction ?>
```

- The above can be written in PHP as follows:

```
<?php
function displayGreeting($name) { ?>
  // HTML code here
<?php } ?>
```

`return` statement in a PHP script

- The `return` statement can be used in a PHP script outside of a function (a global scope) to terminate the execution of the script and optionally return a value to the environment calling the script. By this, when the script is included using `include` or `require` or `include_once` or `require_once`, that expression will return the returned value.
- Example Assume this is a sum.php file

```
function sum(...$x) : int
{
    return array_sum($x);
}
return sum(1, 2, 3); // returns 6 and terminates the script
```

Suppose this script is then included in another PHP file as follows:

```
$result = include 'sum.php';
echo $result; // Output: 6
```

- When the `return` statement is encountered in a PHP script, it immediately stops the execution of the script and returns control to the calling environment, which is typically the web server or command line interface that initiated the script. Codes after the return statement will not be executed.

PHP Function Parameters

- Functions can have zero or more parameters
- In case where your function has multiple parameters, separate them with commas `,`
- The `concat()` function is used to concatenate two strings. It takes two parameters: `$str1` and `$str2`. The function returns the concatenated result of the two strings. Specifying 1 error brings about an error
- It's a good argument to list arguments vertically when the arguments list becomes long for better readability

Trailing Comma in Function Parameters

- From PHP 8 onwards, you can include a trailing comma after the last parameter in a function definition. This means that you can add a comma after the last parameter without causing a syntax error. Example

```php
function exampleFunction($param1, $param2, $param3,) {
    // Function body
}
```

- The trailing comma is optional and does not affect the functionality of the function. However, it can be useful when adding new parameters to the function in the future, as it allows you to add new parameters without modifying the previous line.
- Similarly, from PHP 7.3 onward, you can also use trailing commas in function calls. Example

```php
exampleFunction('value1', 'value2', 'value3',);
```

Passing Arguments by Value

- When a variable is passed as an argument to a function, the value of the variable is copied to the corresponding parameter in the function. This means that any changes made to the parameter inside the function do not affect the original variable outside the function. This is known as passing by value.

Passing Arguments by Reference

- When a variable is passed as an argument to a function by reference, the function receives a reference (or pointer) to the original variable instead of a copy of its value. This means that any changes made to the parameter inside the function will directly affect the original variable outside the function. To pass an argument by reference, you need to prefix the parameter name with an ampersand (`&`) in the function definition. Example

```php
function incrementByReference( &$number ) {
    $number++;
}
$value = 5;
incrementByReference( $value );
echo $value; // Output: 6
```

- Here, since we passed the `$value` variable by reference using the `&` symbol, any changes made to the `$number` parameter inside the `incrementByReference` function directly affect the original `$value` variable outside the function. Thus, after calling the function, the value of `$value` is incremented to `6`. But if the paramter wasn't assigned a value by reference, `$value` will still be unchanged and the output would have been `5`
- So reassigning the parameter inside the function will also change the original variable outside the function.

- What & is neccesarily doing is making the parameter an alias to the argument passed in. Thus, any changes made to the parameter inside the function will directly affect the original variable outside the function.

PHP Default Function Parameters

- In PHP, you can define default values for function parameters. This means that if a caller does not provide a value for a parameter when calling the function, the default value will be used instead. To define a default value for a parameter, you simply assign a value to the parameter in the function definition. Example

```
function function_name($param1 = default_value, $param2 = default_value, ...) {

}
```

- This is practically useful when you already know that a certain parameter will mostly have a specific value. Thus, you can set that value as the default value for the parameter.
- Default Parameters must be at the end

Default Arguments

- Default arguments are the actual values assigned to the parameters when defining a function with default parameters. These default arguments are used when the caller does not provide a (default) value for the corresponding parameter when calling the function.
- Thus a default argument exists when a function parameter has a default value

PHP Named Arguments

- The named arguments feature in PHP 8.0 upward allows you to pass an argument to a function based on the parameter name, rather than the parameter position. This means that you can specify the name of the parameter when calling a function, making it easier to understand which argument corresponds to which parameter, especially when dealing with functions that have many parameters or optional parameters.
- The syntax for using named arguments is as follows:

```
function function_name($param1, $param2, $param3...) {
    // function body
}

function_name(param2: value2, param1: value1, param3: value3);
```

- Notice that when referencing the parameters by name, the order of the arguments does not matter. You can specify the arguments in any order you like. Also, the $ sign is omitted and colons (:) are used

to separate the parameter names from their corresponding values.

Mixing Positional and Named Arguments

- PHP allows you to mix positional and named arguments when calling a function. However, there are some rules to follow:
- Positional arguments must come before named arguments. This means that if you are using both positional and named arguments in a function call, all positional arguments must be specified first, followed by the named arguments else we get an error.
- Also, with named arguments, you can skip optional parameters that have default values by simply not specifying them in the function call. Example

```
function createUser($name, $age = 18, $country = 'USA') {
    echo "Name: $name, Age: $age, Country: $country";
}
createUser(name: 'Alice', country: 'Canada'); // Age will use the default value of
18
```

PHP Variable Scope

- The scope of a variable determines which part of the code can access or modify that variable. In PHP, there are 4 types of variable scopes:
    - Local Scope
    - Global Scope
    - Static Scope
    - Function Parameter Scope

Local Variable

- When you defined a variable within the body of a function, that variable is said to have a local scope. This means that the variable can only be accessed and modified within the function where it is defined - it cannot be accessed outside that function. Example

```
function greet() {
    $message = "Hello, World!"; // Local variable
    echo $message;
}

echo $message; // This will result in an error because $message is not defined in
the global scope
```

- Also, Local scoped variables only exist during the execution of the function. Once the function completes its execution, the local variables are destroyed and their values are lost.

Global Variable

- When you define a variable outside of any function, that variable is said to have a global scope. This means that the variable can be accessed and modified from anywhere in the script, even when the script is included in another script and also including inside functions.
- Now, for a global variable to be accessible inside a function, you need to use the `global` keyword to import the global variable into the local scope of the function. Example

```
$message = "Hello, World!"; // Global variable
function greet() {
    global $message; // Accessing the global variable
    echo $message;
}
greet(); // Output: Hello, World!
```

PHP `$GLOBALS` Array

- PHP stores all global variables in a special associative array called `$GLOBALS`. This array allows you to access global variables from anywhere in the script, including inside functions, without needing to use the `global` keyword such that the variable name is the key of the array while the value is the value of the variable. Example

```
$message = "Hello, World!"; // Global variable

function greet() {
    echo $GLOBALS['message']; // Accessing the global variable using $GLOBALS
array
}
greet(); // Output: Hello, World!
```

- It is important to note that using global variables can lead to code that is harder to read and maintain, as it can create dependencies between different parts of the code. Therefore, it is generally recommended to minimize the use of global variables and instead pass data to functions through parameters whenever possible. It is a bad practice!

Superglobal Variables

- These are basiclally bult-in global variables in PHP that are always accessible, regardless of scope.
- They provide information about the PHP script's environment, server, and user input.
- Some common superglobal variables include `$_GET`, `$_POST`, `$_SESSION`, `$_COOKIE`, and `$_SERVER`.

- Below shows a table of some common superglobal variables in PHP and their meanings:
    - $GLOBALS | An associative array containing references to all global variables in the script. The variable names are the keys of the array.
    - $_SERVER | An associative array containing information about the server and execution environment.
    - $_GET | An associative array of variables passed to the current script via the URL parameters (query string). Basically used to collect data sent in the URL.It returns data from a GET request.

- $_POST | An associative array of variables passed to the current script via the HTTP POST method. It is used to collect data sent in a POST request.
- $_COOKIE | An associative array of variables passed to the current script via HTTP cookies. It returns data from an HTTP cookies.
- $_FILES | An associative array of items uploaded to the current script via the HTTP POST method. It returns data from a POST file upload.
- $_ENV | An associative array of variables passed to the current script via the environment method.
- $_REQUEST | An associative array that contains the contents of `$_GET`, `$_POST`, and `$_COOKIE`. It is used to collect data after submitting an HTML form.
- $_SESSION | An associative array containing session variables available to the current script. It is used to store and retrieve data across multiple pages during a user's session.

Static Variables

- A Static variable is a local variable that retains its value between function calls. This means that the value of the static variable persists even after the function has completed its execution, and it can be accessed and modified in subsequent calls to the same function.
- To declare a static variable in PHP, you use the `static` keyword before the variable name inside the function. Example

```
function counter() {
    static $count = 0; // Static variable
    $count++;
    echo $count;
}
counter(); // Output: 1
counter(); // Output: 2
counter(); // Output: 3
```

- In this example, the `$count` variable is declared as static inside the `counter` function. Each time the function is called, the value of `$count` is incremented by 1 and echoed. Since `$count` is static, it retains its value between function calls, resulting in the output of `1`, `2`, and `3` on successive calls to the `counter` function.

- Another case of this static variable is displayed by the following code

```
function getValue() {
    $value = someHeavyFunction();

    echo $value;
}

function someHeavyFunction() {
    sleep(2);

    return 10;
```

```
}

getValue();
getValue();
getValue();
```

- In this example, the `someHeavyFunction()` simulates a time-consuming operation by sleeping for 2 seconds before returning a value of `10`. Each time the `getValue()` function is called, it calls `someHeavyFunction()`, resulting in a delay of 2 seconds for each call. To fix this, we can use a static variable to store the result of `someHeavyFunction()` so that it is only called once. Here is the modified code:

```php
function getValue() {
    static $value = null;

    if ($value === null) {
        $value = someHeavyFunction();
    }

    echo $value;
}
```

Here, the `$value` variable is declared as static inside the `getValue()` function. The first time `getValue()` is called, it checks if `$value` is `null`, and if so, it calls `someHeavyFunction()` to get the value and stores it in `$value`. On subsequent calls to `getValue()`, the stored value is not not `null` and is used directly, avoiding the delay caused by calling `someHeavyFunction()` again.

Function Parameter Scope

- Function Parameter Scope refers to the scope of variables that are defined as parameters in a function. These parameters are local to the function and can only be accessed and modified within the function body. Example

```php
function greet($name) { // $name is a parameter with local scope
    echo "Hello, $name!";
}
greet("Alice"); // Output: Hello, Alice!
greet("Bob");   // Output: Hello, Bob!
```

PHP `declare`

- The `declare` construct in PHP is used to set execution directives for a block of code or an entire script. It allows you to specify certain behaviors or settings that affect how the code is executed.
- It does not call a function or produce an output. Rather, it controls how PHP code is interpretted and executed.

- The syntax of the `declare` construct is as follows:

```
declare (directive = value) {
    // code to be executed
}
```

- Here, the `directive` is a specific setting that you want to apply to the code block.
- `value` is the value you want to assign to the directive. This value can be an integer, boolean, or other appropriate data type depending on the directive being used.
- The code block starts with the `{` and ends with `}`.
- The `directive` can be one of several options, such as `ticks`, `encoding`, or `strict_types`. Each directive has its own specific purpose and behavior.
- It can also be applied to the entire script without the curly braces as follows:

```
declare (directive = value);
```

By this, the directive will apply to any code that comes after it in the script.

PHP `declare(ticks=N)`

- The `ticks` directive is used to specify the number of ticks that should occur before a registered tick handler function is called.
- A tick is a low-level event that occurs during the execution of a PHP script per N statement, and it allows you to execute code at specific intervals during the script's execution.
- To use the `ticks` directive, you need to register a tick handler function using the `register_tick_function()` function. This function takes the name of the tick handler function as its argument.
- The tick handler function will be called every N ticks (i.e after N statements), where N is the value specified in the `declare(ticks=N)` directive.
- Example

```
declare(ticks = 2);

function tick_handler() {
    print "Executed! \n";
}

register_tick_function('tick_handler');

$age = 19;

$message = match($age) {
            18 => 'Diamond',
            40 => 'Saphaire',
```

```
            50 => 'Golden',
            60 => 'Silver',
            default => "You age isn't special!"
        };

echo $message;
```

How it works

- In this example, the `tick_handler()` function is registered as a tick handler using the `register_tick_function()`
- This function will be called every time a tick occurs during the execution of the script.
- The `declare(ticks=2)` directive specifies that a tick should occur every 2 statements.
- As the script executes, the `tick_handler()` function will be called every 2 statements, resulting in the output "Executed!" being printed multiple times throughout the script's execution.

- Practically, you can use the `ticks` directive to perform tasks such as profiling, debugging, or monitoring the execution of your PHP code at specific intervals.

PHP `declare(encoding='encoding_name')`

- The `encoding` directive is used to specify the character encoding for a block of PHP code. This directive allows you to define the encoding that should be used for string literals and other character data within the specified code block.
- To use the `encoding` directive, you need to specify the desired encoding name as the value of the directive. Example

```
declare(encoding='UTF-8') {
    // code to be executed with UTF-8 encoding
}
```

PHP `declare(strict_types=1)`

- The `strict_types` directive is used to enable strict typing in PHP. When strict typing is enabled, PHP will enforce type declarations for function parameters and return values in that current script, meaning that the types of the argument passed later into the function must match exactly as specified in the function definition.
- If the script is later included in another script without strict typing enabled, the strict typing rules will not apply to the included script and arguments can be coerced to match the expected types.

PHP Type Hints

- Adding a string and an integer together results in an error
- By type hints you can enforce the types for function parameters and return values.
- Type hints help catch type-related errors early in the development process, making your code more robust and easier to maintain.

- It is mostly used in function definitions to specify the expected data types of parameters and return values.
- To add type hints to function parameters, you simply specify the desired type before the parameter name in the function definition. That is

```
function function_name( type $parameter_1, type $parameter_2, ...) {

}
```

where

- `type` can be any valid data type in PHP, such as `int`, `float`, `string`, `array`, `bool`, `callable`, `iterable`, `object`, or a class/interface name.
- `parameter_1`, `parameter_2`, ... are the names of the parameters.
- By this, PHP will enforce that the arguments passed to the function match the specified types. If an argument of a different type (that cannot be coerced) is passed, a `TypeError` will be thrown.
- Note that in the function block, the type of the parameter can be changed and won't result in an error

PHP Type Hints for Return Values

- To specify the return type of a function, you add a colon (`:`) followed by the desired return type after the closing parenthesis of the parameter list in the function definition. That is

```
function function_name( type $parameter_1, type $parameter_2, ... ) : return_type
{

}
```

where

- `return_type` can be any valid data type in PHP, such as `int`, `float`, `string`, `array`, `bool`, `callable`, `iterable`, `object`, or a class/interface name.
- By this, PHP will enforce that the value returned by the function matches the specified return type. If a value of a different type (that cannot be coerced) is returned, a `TypeError` will be thrown.
- From PHP 7.0 onward, if a function does not return any value, you can specify the return type as `void`. This indicates that the function is not expected to return any value. By this `NULL` will be returned. If you return a value, you get an error even though you return null. Example

```
function function_name( type $parameter_1, type $parameter_2, ... ) : void {

}
```

The Union Type

- From PHP 8.0 onward, if a function return a value that can be of multiple types or accepts arguments of multiple types, you can specify a union type for the return type or parameter. This is done by separating the possible types with a pipe (`|`) symbol. Example

```
function function_name( type | type1 $parameter_1, type | type2 $parameter_2, ...
) : type1 | type2 | type3 {

}
```

Thus, the type of value returned depends on the type of arguments passed to the function.

The mixed type

- From PHP 8.0 onward, you can use the `mixed` type hint to indicate that a function parameter or return value can be of any type. This is useful when you want to allow flexibility in the types of values that can be passed to or returned from a function. The `mixed` type is equivalent to

```
object|resource|array|string|int|float|bool|null
```

The nullable type

- The following defines a function that converts a string to uppercase.

```
function toUpperCase( string $input ) : string
{
    return strtoupper($input);
}
```

- In the above code, if you pass null as an argument, you'll get a TypeError because the function expects a string argument. To allow null values, you can use the nullable type hint by prefixing the (parameter and return) type with a question mark (`?`). Example

```
function toUpperCase( ?string $input ) : ?string
{
    if ( $input === null ) {
        return null;
    }
    return strtoupper( $input );
}
```

- Generally, for any defined function of a certain type(s)

```
function function_name( type $parameter_1, type $parameter_2, ... ) : return_type
{

}
```

you can make it nullable as follows

```
function function_name (?type $parameter_1, ?type $parameter_2, ...): ?return_type
{
  if ($parameter_1 != null) {
    return $paramter_1;
  }

  return null;
}
```

- In the above general snipett, we add the `?` to the type of the parameter. The ?return_type allows you to pass the return type of the function argument or null.
- The nullable type was introduced in PHP 7.1
- Note that the `mixed` type has already included the `null` type for return values.So there is no need to do this

```
?mixed
```

Doing this will result in an error.

PHP Stict Types

- To enable strict typing in a PHP file, you can use the `declare(strict_types=1);` directive at the very beginning of the file, before any other code. This directive tells PHP to enforce strict type checking for function calls and return values within that file.
- Adding strict types means that PHP will not perform type coercion when passing arguments to functions or returning values from functions. Instead, it will throw a `TypeError` if the types do not match exactly. Example

```
declare(strict_types=1);
function add(int $a, int $b): int {
    return $a + $b;
}
echo add(1, 2); // Output: 3
echo add(1.5, 2.5); // Throws TypeError
```

PHP Strict Types: the Special case

- PHP has a special case when the target type is `float` and the argument type is `int`. In this case, PHP allows the conversion from `int` to `float` even in strict mode. This means that if a function expects a `float` parameter and you pass an `int` argument, PHP will automatically convert the `int` to a `float` without throwing a `TypeError`. Example

```
declare(strict_types=1);
function calculateArea(float $radius): float {
    return 3.14 * $radius * $radius;
}
echo calculateArea(5); // Output: 78.5
```

- Thus if your parametrs are of type float, you can still pass integers without causing a TypeError.

PHP strict_types and include

- The `include` and `require` statements in PHP are used to include and evaluate files in the current script. When using these statements, the `strict_types` directive is not inherited from the included file to the including file or vice versa. Each file has its own `strict_types` setting, and they do not affect each other.

PHP Variadic Functions

- A variadic function is a function that can accept a variable number of arguments. That is, there no fixed number of parameters that must be passed to the function when it is called. Instead, you can pass any number of arguments, and the function will handle them accordingly.
- In PHP, to allow a function accept a variable number of arguments, we use the `func_get_args()` function.
- The `func_get_args()` function returns an array containing all the arguments passed to the function. Example

```
function sum() {
    $args = func_get_args(); // Get all arguments as an array
    $total = 0;
    foreach ($args as $num) {
        $total += $num; // Sum up the arguments
    }
    return $total;
}
echo sum(1, 2, 3); // Output: 6
echo sum(4, 5, 6, 7, 8); // Output: 30
```

Here, the `args` variable is an array that contains all the arguments passed to the `sum` function. The function then iterates over the array and calculates the total sum of the arguments.

- Notice that no parameters are defined in the function definition. This is because the function can accept any number of arguments.
- From PHP 5.6 onward, you can also use the `...` (splat) operator to define variadic functions. This operator allows you to specify a parameter that can accept a variable number of arguments. Example

```
function sum(...$numbers) {// Get all arguments as an array. Those are stored in
$numbers
    $total = 0;
    foreach ($numbers as $num) {
        $total += $num; // Sum up the arguments
    }
    return $total;
}
echo sum(1, 2, 3); // Output: 6
echo sum(4, 5, 6, 7, 8); // Output: 30
```

Here, the `...$numbers` parameter basically acts as an array that contains all the arguments passed to the `sum` function. The function then iterates over the array and calculates the total sum of the arguments.

- The splat operator is more concise and easier to read than using `func_get_args()`. Also, from PHP 7, it allows you to specify type hints for the variadic parameter. Example

```
function sum(int ...$numbers) {; // Get all arguments of the type int as an array
    $total = 0;
    foreach ($numbers as $num) {
        $total += $num; // Sum up the arguments
    }
    return $total;
}
echo sum(1, 2, 3); // Output: 6
echo sum(4, 5, 6, 7, 8); // Output: 30
```

- Now, if a function has multiple parameters, only the last parameter can be variadic. That is

```
function function_name($param1, $param2, ...$variadic_param) {

}
```

- If a variadic parameter is not defined last, we get a syntax error.

```
Fatal error: Only the last parameter can be variadic in...
```

- Also, a function can have only one variadic parameter.

PHP Arrays

- An array is basically a list/collection of elements (values) that are stored under a single variable name.
- PHP Arrays are of two types
    - Indexed Arrays
    - Associative Arrays
- The keys in an Associative array are usually strings, while the keys in an Indexed array are usually integers from 0 to the number of elements present in the array minus one.
- Arrays are used to store multiple values in a single variable, making it easier to manage and manipulate related data.

Creating Arrays

- To define a array, you can either use the `array()` function or the short array syntax `[]`. For readability, the short array syntax is preferred.
- You use the `array()` construct as follows:

```
$array_name = array( element1, element2, element3, ... );
```

- Or you can use the short array syntax as follows:

```
$array_name = [ element1, element2, element3, ... ];
```

- The short syntax is known as the JSON notation.

Displaying Arrays

- To show the contents of an array, you can use the `print_r()` function or the `var_dump()` function. Example
- Using `var_dump()` in this manner

```
$array_name = [ "Apple", "Banana", "Orange" ];
var_dump( $array_name );
```

yields this output

```
  [0]=>
  string(5) "Apple"
  [1]=>
  string(6) "Banana"
  [2]=>
  string(6) "Orange"
}
```

```
- Using `print_r()` in this manner

```php
$array_name = [ "Apple", "Banana", "Orange" ];
print_r( $array_name );
```

yields this output

```
Array
(
    [0] => Apple
    [1] => Banana
    [2] => Orange
)
```

- To make the output of `print_r()` more readable in a web browser, you can wrap it in `<pre>` HTML tags as follows:

```
echo "<pre>";
print_r( $array_name );
echo "</pre>";
```

Accessing Elements in an Array

- To access individual elements in an array, you use the index (for indexed arrays) or the key (for associative arrays) inside square brackets `[]`. Example

```
$array_name = [ "Apple", "Banana", "Orange" ];
echo $array_name[1]; // Output: Banana
```

- Here, we access the second element of the array (index 1) and echo its value, which is "Banana".
- For associative arrays, you access elements using their keys. Example

```
$array_name = [ "name" => "John", "age" => 30, "city" => "New York" ];
echo $array_name["age"]; // Output: 30
```

- Unlike strings, you cannot use negative indexes to access elements from the end of an array in PHP. Attempting to do so will result in an undefined array key error. Same applies when you try to access an index that is out of bounds (greater than the highest index in the array).

Adding Elements to an Array
```

- To add an element to an indexed array, you use the following syntax:

```
$array_name[] = new_element;
```

- Here, PHP calculates the highest integer index currently in the array and adds 1 to it and assigns it as the new index of the new element. Thus, the new element is added to the end of the array.

Changing Array Elements

- The following syntax is used to change an element in an array:

```
$array_name[index_or_key] = new_value;
```

- Here, you specify the index (for indexed arrays) or key (for associative arrays) of the element you want to change, and assign it a new value.

Removing Elements from an Array

- To remove an element from an array, you can use the `unset()` function.
- Its syntax is as follows:

```
unset( $array_name[index_or_key] );
```

- This function can also take multiple elements
- Example

```
$array_name = [ "Apple", "Banana", "Orange" ];
unset( $array_name[1] ); // Removes the element at index 1 (Banana)
print_r( $array_name ); // Output: Array ( [0] => Apple [2] => Orange )
```

- Note that after removing an element, the array keys are not automatically reindexed. Thus, if you try to access the removed index, you will get an undefined index error.
- Consider this example

```
$even = [2, 40 => 4, 6];
unset($even[0], $even[40], $even[41]);
print_r($even);

$even[] = 83;

print_r($even);
```

By the result in the above example, whenever `unset()` is used to remove all elements in an array, and you try to append an element to the array using `array[] = new_element` or `array_push($array, new_element)`, the largest indexed at that point is retained and the new index of the new element will be the `largest index (before removal) + 1`

- To reindex the array after removing an element, you can use the `array_values()` function. Example

```
$array_name = [ "Apple", "Banana", "Orange" ];
unset( $array_name[1] ); // Removes the element at index 1 (Banana)
$array_name = array_values( $array_name ); // Reindex the array
print_r( $array_name ); // Output: Array ( [0] => Apple [1] => Orange )
```

- The `array_values()` function returns a new array with the values from the original array, reindexed with sequential integer keys starting from 0.
- This function does not work for an associative array. Also, it doesn't modify the original array; it returns a new array with reindexed values.
- Thus, whenever unset is used to remove an element from an indexed array, it is a good practice to use `array_values()` to reindex the array if you plan to access the elements by their indexes later.

Getting the Length / Size of an Array

- To get the length (number of elements) of an array, you can use the `count()` function. Example

```
$score = [ 90, 85, 78, 92 ];
$length = count( $score );
echo $length; // Output: 4
```

PHP Associative Arrays

- An associative array is an array that uses named keys (strings or integers) to identify its elements instead of zero-indexed numeric keys.
- You can create an associative array using the same syntax as indexed arrays, but instead of using numeric indexes, you use string keys. Example

```
$assoc_array = [
  "first" => "Apple",
  "second" => "Banana",
  "third" => "Orange"
];
```

- Consider another

```
$even_indexed_array = [
        2 => 2,
        4 => 4
];

print_r($even_indexed_array);
```

Here, the keys are integers but they are not zero-indexed. If a key that is neither string nor integer is used, PHP tries to convert it to either a string or an integer.

- Consider this code such that there is both a named key and an indexed key

```
$arr_name = [
    a,
    b,
    "5" => c,
    d
]

print_r($arr_name);
// Output: Array ( [0] => a [1] => b [5] => c [6] => d )
```

Here, the indexed after the named index is automatically set to the highest integer index plus one.

- To append an element, to an associative array, you have to specify the key for the new element. That is:

```
$array_name["new_key"] = new_value;
```

This element will be added at the last of the array.

- If you have multiple keys that are the same, the last one will overwrite the previous ones. Example

```
    $versions = [
        "PHP 7" => "7.4",
        "PHP 8" => "8.0",
        "PHP 8" => "8.1" // This will overwrite the previous "PHP 8" key
    ]
    print_r($versions);
    // Output: Array ( [PHP 7] => 7.4 [PHP 8] => 8.1 )
```

- Now notice that PHP perfroms type juggling when using keys in an associative array. Thus, if you use a key that is an integer or a string that can be converted to an integer, PHP will treat it as an integer key. Example

```
$assoc_array = [
  "1" => "Apple", // This key is treated as integer 1
  2 => "Banana", // This key is an integer
  "three" => "Orange" // This key is a string
];
print_r( $assoc_array );
```

- Consider this example

```
$assoc_array = [
  "1" => "Apple", // This key is treated as integer 1
  2 => "Banana", // This key is an integer
  "three" => "Orange", // This key is a string
  true => "Mango", // This key is treated as integer 1 (overwrites "Apple")
  1 => "Grapes", // This key is an integer (overwrites "Mango")
  1.5 => "Pineapple" // This key is treated as integer 1 (overwrites "Grapes")
];

print_r( $assoc_array );
// Output: Array ( [1] => Pineapple [2] => Banana [three] => Orange )
```

- Use an associative array to reference elements by names (keys) instead of numbers (indexes).

PHP Multidimensional Arrays

- A multidimensional array is an array that contains one or more arrays as its elements. This allows you to create complex data structures that can represent more than one dimension of data.
- For example, you can create a two-dimensional array (an array of arrays) to represent tasks and number of hours spent on each task for different days of the week.

```
$tasks = [
  ['Learn PHP programming', 2],
  ['Practice PHP', 2],
  ['Work', 8]
]
```

Removing Elements from a Multidimensional Array

- To remove an element from a multidimensional array, you can use the `unset()` function along with the appropriate indexes or keys to specify the element you want to remove. Example

```
$tasks = [
  ['Learn PHP programming', 2],
  ['Practice PHP', 2],
```

```
  ['Work', 8]
];

unset( $tasks[1][0] ); // Removes only the first element of the second sub-array
('Practice PHP')
print_r( $tasks ); // Output: Array ( [0] => Array ( [0] => Learn PHP programming
[1] => 2 ) [1] => Array ( [0] => [1] => 2 ) [2] => Array ( [0] => Work [1] => 8 )
)
```

PHP Array Functions

PHP `array_splice()` function

- The `array_splice` function can also be used to remove elements from a array. It takes the array, the starting index, and the number of elements to remove as arguments.
- It's syntax is as follows:

```
array_splice( array, start_index, number_of_elements_to_remove );
```

- Consider this example

```
$tasks = [
  ['Learn PHP programming', 2],
  ['Practice PHP', 2],
  ['Work', 8]
];
array_splice( $tasks, 1, 1 ); // Removes only the second element (['Practice PHP',
2])
print_r( $tasks ); // Output: Array ( [0] => Array ( [0] => Learn PHP programming
[1] => 2 ) [1] => Array ( [0] => Work [1] => 8 ) )
```

- Generally, `array_splice` is more useful when you want to remove multiple elements or a range of elements from a multidimensional array.
- Practically, use `array_splice` when you want to remove element(s) from an Indexed Array

PHP `array_splice()` with Associative array

- Whenever `array_splice()` is used with an Associative array, the keys are reindexed to numeric indexes starting from 0. Example

```
$assoc_array = [
  "first" => "Apple",
  "second" => "Banana",
  "third" => "Orange"
];
```

```
array_splice( $assoc_array, 1, 1 ); // Removes only the second element ("Banana")
print_r( $assoc_array ); // Output: Array ( [0] => Apple [1] => Orange )
```

Iterating over a Multidimensional Array using foreach

- To iterate over a multidimensional array using `foreach`, you can use nested `foreach` loops. The outer loop iterates over the main array, while the inner loop iterates over each sub-array. Example

```
$tasks = [
  ['Learn PHP programming', 2],
  ['Practice PHP', 2],
  ['Work', 8]
];

foreach ( $tasks as $task ) {
    foreach ( $task as $detail ) {
        echo $detail . " ";
    }
    echo "\n"; // New line after each task
}
```

Accessing Elements in a Multidimensional Array

- We follow this syntax to access elements in a multidimensional array:

```
$array_name[index1][index2]...
```

Sorting a Multidimensional Array

- To sort a multidimensional array in PHP, you can use the `usort()` function along with a custom comparison function. The `usort()` function sorts an array by values using a user-defined comparison function. It takes two parameters: the array to be sorted and the comparison function.
- The `usort()` has the following syntax:

```
usort( array, comparison_function );
```

- The `comparison_function` is a callback function that defines the sorting logic. It takes two parameters (elements from the array) and returns:
  - A negative value if the first element should come before the second element.
  - A positive value if the first element should come after the second element.
  - Zero if both elements are considered equal.
- If the `comparison_function`can be an Anonymous function (Closure) or a Named function.
- If it is an Anonymous function, it is defined directly within the `usort()` call as follows:

```
usort( $array, function ( $a, $b ) {
    // Comparison logic here
} );
```

- If it is a Named function, it is defined separately and then passed as a string to `usort()` as follows:

```
function compare( $a, $b ) {
    // Comparison logic here
}
usort( $array, 'compare' );
```

- Consider this example where we sort a multidimensional array based on the second element of each sub-array (number of hours spent on each task):

```
$tasks = [
  ['Learn PHP programming', 2],
  ['Practice PHP', 2],
  ['Work', 8]
];

usort( $tasks, function ( $a, $b ) {
    return $a[1] <=> $b[1]; // Compare based on the second element (hours)
} );
```

PHP array_unshift() function

- To prepend one or more elements to the beginning of an (Indexed) array, you can use the `array_unshift()` function. This function adds the specified elements to the start of the array and shifts the existing elements to higher indexes.
- The `array_unshift()` function:
    - Adds elements to the beginning of an array
    - Shifts existing elements to the right
    - Reindexes numeric keys
- It returns the new number of elements in the array after the elements have been added.
- Also, it modifies the original array directly.
- The syntax of the `array_unshift()` function is as follows:

```
array_unshift( array &$array, mixed  $value1, mixed  $value2, ... );
```

- When multiple values are provided, they are added to the beginning of the array in the order they are specified.
- Example:

```
$fruits = [ "Banana", "Orange" ];
array_unshift( $fruits, "Apple", "Mango" );
print_r( $fruits ); // Output: Array ( [0] => Apple [1] => Mango [2] => Banana [3]
=> Orange )
```

Preceding an element to the beginning of a Associative array

- To prepend an element to the beginning of an associative array in PHP, we reassign the array as follows:

```
$array_name = [ "new_key" => new_value ] + $array_name ;
```

- Here, we create a new array with the new key-value pair and then use the `+` operator to combine it with the existing associative array. This effectively adds the new element to the beginning of the array.
- This is called array union in PHP.
- Similarly, to prepend elements to the end of an associative array, you can do this:

```
$array_name = $array_name + [ "new_key" => new_value ];
```

Notice that the new value and key pair is defined after the existing array.

PHP array_push() function

- The PHP `array_push()` function is used to add one or more elements to the end of an (Indexed) array. This function appends the specified elements to the end of the array and increases the size of the array accordingly.
- The `array_push()` function:
  - Adds elements to the end of an array
  - Increases the size of the array
  - Modifies the original array directly
- It returns the new number of elements in the array after the elements have been added.
- The syntax of the `array_push()` function is as follows:

```
array_push( array &$array, mixed  $value1, mixed  $value2, ... );
```

- It has the same functionality as using the `[]` operator to append elements to an array.
- Use `[]` to add a single element to the end of an array as follows while `array_push()` is used to add multiple elements.

```
$array_name[] = new_value; // Adds a single element to the end of the array
array_push( $array_name, new_value1, new_value2, ... ); // Adds multiple elements
to the end of the array
```

PHP array_pop() function

- The `array_pop()` function in PHP is used to remove and return the last element from an array. This function modifies the original array by removing the last element and returns the value of that element.
- It's syntax is as follows:

```
array_pop($array_name);
```

- If the array is empty, `array_pop()` returns `NULL`.

Removing the last element from an Associative array

- Using `array_pop()` on an associative array simply return the value to the last key.

PHP array_shift() function

- The `array_shift()` function in PHP is used to remove and return the first element from an array. This function modifies the original array by removing the first element and returns the value of that element.
- It's syntax is as follows:

```
array_shift($array_name);
```

- Similarly, if the array is empty, `array_shift()` returns `NULL`.

`array_shift()` with an array of mixed keys

- When `array_shift()` is used with an array that has both indexed and associative keys, it removes and returns the first element based on the internal order of the array, which is determined by the order in which the elements were added to the array and the elements get reindexed.
    - If one of the keys is at an index different from the normal order, it will be reassigned a new key
    - If one of the keys is string, it isn't reindexed

PHP array_keys() function

- The `array_keys()` function in PHP is used to retrieve all the keys from an array. This function returns a new array containing the keys of the original array.
- It's syntax is as follows:

```
array_keys( array $array, mixed $search_value = null, bool $strict = false ) :
array
```

- Here,
    - `array` is the input array from which you want to retrieve the keys.

    - `search_value` (optional) is a value to search for in the array. If provided, only the keys corresponding to this value will be returned as elements in the array.
    - `strict` (optional) is a boolean flag that determines whether to use strict comparison (type and value) when searching for the `search_value`. The default is `false` which will cause type juggling during the search.

- Consider this example:

```
$fruits = [
    "a" => "Apple",
    "b" => "Banana",
    "c" => "Orange",
    "d" => "Banana"
];
$keys = array_keys( $fruits, "Banana" );
print_r( $keys ); // Output: Array ( [0] => b [1] => d )
```

- Condider this other example with the `search_value` parameter and `strict`:

```
$values = [
    "x" => 10,
    "y" => "10",
    "z" => 20
];
$keys = array_keys( $values, 10, true );
print_r( $keys ); // Output: Array ( [0] => x )
```

Here, since we set `strict` to `true`, only the key "x" is returned because it corresponds to the integer value `10`, while "y" corresponds to the string value `"10"`. Type matters in strict comparison.

- Notice that if the `search_value` parameter is not provided, the `array_keys()` function will return all the keys from the input array.
- Thus the `strict` parameter is only useful when the `search_value` parameter is provided.

PHP array_key_exists() function

- The `array_key_exists()` function in PHP is used to check if a specific key exists in an array. This function returns `true` if the key is found in the array, and `false` otherwise.
- It's syntax is as follows:

```
array_key_exists( mixed $key, array $array ) : bool
```

Here, the `key` comes before the array.

- The function searches for the specified `key` in the first dimension of the array only. It does not search in nested arrays.

PHP array_key_exists() vs isset()

- `isset()` works similarly to `array_key_exists()`, but there is a key difference:
  - `isset()` returns `false` if the key exists but its value is `NULL`.
  - `array_key_exists()` returns `true` if the key exists, regardless of its value (even if it is `NULL`).
- The syntax of `isset()` is as follows:

```
isset($vars['key']);
```

- Consider this example:

```
$device = [
    'on' => true,
    'off' => null
];

var_dump(array_key_exists('off', $device));// Output: bool(true)
var_dump(isset($device['off']));// Output: bool(false)
```

- Stick to `array_key_exists()` when you want to check for the existence of a key in an array, regardless of its value. Use `isset()` when you want to check if a key exists and its value is not `NULL`.

PHP in_array() function

- The `in_array()` function in PHP is used to check if a specific value exists in an array. This function returns `true` if the value is found in the array, and `false` otherwise.
- It's syntax is as follows:

```
in_array( mixed $needle, array $haystack, bool $strict = false ) : bool
```

Here,

- `$needle` is the varaible that holds value you want to search for in the array.
- `haystack` is the input array in which you want to search for the value.
- `strict` (optional) is a boolean flag that determines whether to use strict comparison (type and value) when searching for the `needle`. The default is `false`. This is useful to avoid type juggling during the search.

- Consider this example:

```
$device = [
    'on' => true,
    'off' => null
];
```

```
var_dump(in_array(false, $device)); // Output: bool(true) because null is
considered equal to false in non-strict comparison
var_dump(in_array(false, $device, true)); // Output: bool(false) because null is
not identical to false in strict comparison
```

PHP array_merge()

- The `array_merge()` function is used to merge one or two arrays together. It returns a new array that contains element from the two input arrays
- It appends elements of the next array to the last element of the previous array
- It's syntax is as follows

```
array_merge( array $array1, array $array2, ... ) : array
```

- Example

```
$even = [2, 4, 6, 8];
$odd = [1, 3, 5, 7];

array_merge($even, $odd);
print_r(array_merge($even, $odd));
/* Output:
Array
(
    [0] => 2
    [1] => 4
    [2] => 6
    [3] => 8
    [4] => 1
    [5] => 3
    [6] => 5
    [7] => 7
)*/
```

Using `array_merge()` with an Associative Array with string keys

- Consider the following code snippet

```
$odd_days_hours = [
    'Monday' => 4,
    'Wednesday'=> 2,
    'Friday' => 9
];

$even_days_hours = [
```

```
    'Tuesday' => 7,
    'Thursday' => 4
];

print_r(array_merge($odd_days_hours, $even_days_hours));

/*Output
Array
(
    [Monday] => 4
    [Wednesday] => 2
    [Friday] => 9
    [Tuesday] => 7
    [Thursday] => 4
)
*/
```

Notice that the order is kept. The elements of the next array is appended to the last element of the previous array

- Now, consider another code snippet

```
$odd_days_hours = [
    'Monday' => 4,
    'Wednesday'=> 2,
    'Friday' => 9
];

$even_days_hours = [
    'Tuesday' => 7,
    'Thursday' => 4,
    'Monday' => 3
];

print_r(array_merge($odd_days_hours, $even_days_hours));

/*Output
Array
(
    [Monday] => 3
    [Wednesday] => 2
    [Friday] => 9
    [Tuesday] => 7
    [Thursday] => 4
)
*/
```

Notice that in this example, the key `'Monday'` is reassigned. Thus, if a key reoccurs in the next array, its value in the previous array is ignored and the value in the next is assigned to it instead. But it keeps to the position it happens to be in the previous

- Now, notice that to merge two associaitve arrays together, you can simply do this

```
$odd_days_hours = [
    'Monday' => 4,
    'Wednesday'=> 2,
    'Friday' => 9
];

$even_days_hours = [
    'Tuesday' => 7,
    'Thursday' => 4,
    'Monday' => 3
];

print_r(($odd_days_hours + $even_days_hours));
```

But the difference of this and using the `array_merge()` function is that it whenever the key reappears in the next array with a different value, it is totally ignored. Rather, it keeps to the key and its value in the previous

- From PHP 7.4, PHP returns an empty array whenever `array_merge()` is called with no arguments. Before PHP 7.4, it resulted in a warning.

PHP `array_sum()` function

- The `array_sum()` function in PHP is used to calculate the sum of all the values in an array. It takes an array as input and returns the total sum of its elements.
- It's syntax is as follows:

```
array_sum( array $array ) : float|int
```

PHP `array_sum()` with an Assocative Array containing an array

- Consider the following snippet

```
$nums = [1,3, 'even' => [2, 4]];

$sum = array_sum($nums);

echo $sum;
// Output: 4
```

Why is the output `4`?

- The `array_sum()` function only sums up the top-level numeric values in the array. Suppose the `'even'` key is equal to `2` instead of an array, then the output would be `6` since it would sum up `1 + 3 + 2 =`

`6`.

- If the array contains nested arrays, `array_sum()` does not recursively sum the values within those nested arrays. It only considers the top-level numeric values.

PHP `array_sum()` with non-numeric values

- When using `array_sum()` with an array that contains non-numeric values, PHP will attempt to convert those values to numbers before performing the summation. Here are some rules PHP follows for type conversion:
  - Strings that represent valid numbers (e.g., "10", "3.14") will be converted to their numeric equivalents.
  - Boolean values will be converted to `1` for `true` and `0` for `false`.
  - Non-numeric strings (e.g., "apple", "hello") will be converted to `0`.
  - Null values will also be treated as `0`.

PHP `array_chunk()` function

- The `array_chunk()` function in PHP is used to split an array into smaller arrays (chunks) of a specified size. It takes an array and a chunk size as input and returns a multidimensional array containing the chunks.

- It's syntax is as follows:

```
array_chunk( array $array, int $size, bool $preserve_keys = false ) : array
```

where

- `array` is the input array to be split into chunks.
- `size` is the size of each chunk (number of elements in each chunk).
- `preserve_keys` (optional) is a boolean flag that determines whether to preserve the original array keys in the chunks. The default is `false`, which means the keys will be reindexed in each chunk.

- Consider this example:

```
$numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9];

$chunks = array_chunk($numbers, 3);
print_r($chunks);
/* Output:
Array
(
    [0] => Array
        (
            [0] => 1
            [1] => 2
            [2] => 3
        )
```

```
    [1] => Array
        (
            [0] => 4
            [1] => 5
            [2] => 6
        )

    [2] => Array
        (
            [0] => 7
            [1] => 8
            [2] => 9
        )

)
*/
```

- Consider the same example such that keys are preserved

```php
$numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9];

$chunks = array_chunk($numbers, 3, true);
print_r($chunks);
/* Output:
Array
(
    [0] => Array
        (
            [0] => 1
            [1] => 2
            [2] => 3
        )

    [1] => Array
        (
            [3] => 4
            [4] => 5
            [5] => 6
        )

    [2] => Array
        (
            [6] => 7
            [7] => 8
            [8] => 9
        )

)
*/
```

PHP `array_combine()` function

- The `array_combine()` function in PHP is used to create a new associative array by combining two arrays: one for the keys and another for the values. It takes two arrays as input and returns a new associative array where the elements from the first array are used as keys and the elements from the second array are used as values.
- It's syntax is as follows:

```
array_combine( array $keys, array $values ) : array
```

where

- `keys` is the array containing the keys for the new associative array.
- `values` is the array containing the values for the new associative array.
- Consider this example:

```
$keys = ['name', 'age', 'city'];
$values = ['Alice', 30, 'New York'];
$associative_array = array_combine($keys, $values);
print_r($associative_array);
/* Output:
Array
(
    [name] => Alice
    [age] => 30
    [city] => New York
)
*/
```

- Know that if the size of the array don't match (i.e are not the same), we get an error

PHP `array_search()` function

- The `array_search()` function in PHP is used to search for a specific value in an array and return the first corresponding key if the value is found. If the value is not found, it returns `false`otherwise.
- It's syntax is as follows:

```
array_search( mixed $needle, array $haystack, bool $strict = false ) :
int|string|false
```

where

- `needle` is the value you want to search for in the array.

- `haystack` is the input array in which you want to search for the value.
- `strict` (optional) is a boolean flag that determines whether to use strict comparison (type and value) when searching for the `needle`. The default is `false`. This is useful to avoid type juggling during the search.

- Consider this example:

```
$fruits = [
    "a" => "Apple",
    "b" => "Banana",
    "c" => "Orange",
    "d" => "Banana"
];
$key = array_search("Banana", $fruits);
echo $key; // Output: b
```

PHP `array_diff()` function

- The `array_diff()` function in PHP is used to compare two or more arrays and return the values from the first array that are not present in any of the other arrays. It performs a value-based comparison (not keys) and returns an array containing the differences.

- It's syntax is as follows:

```
array_diff( array $array1, array $array2, array ...$arrays ) : array
```

- Consider this example:

```
$array1 = [ "a", "b", "c", "d" ];
$array2 = [ "b", "d", "e" ];
$difference = array_diff( $array1, $array2 );
print_r( $difference ); // Output: Array ( [0] => a [2] => c )
```

PHP `array_diff_assoc()` function

- The `array_diff_assoc()` function in PHP is used to compare two or more associative arrays and return the key-value pairs from the first array that are not present in any of the other arrays. It performs a key-value based comparison and returns an array containing the differences.

- It's syntax is as follows:

```
array_diff_assoc( array $array1, array $array2, array ...$arrays ) : array
```

- Consider this example:

```
$array1 = [ "a" => "Apple", "b" => "Banana", "c" => "Cherry", "d" => "Date" ];
$array2 = [ "b" => "Banana", "c" => "Date" ];
$difference = array_diff_assoc( $array1, $array2 );
print_r( $difference ); // Output: Array ( [a] => Apple [c] => Cherry [d] => Date
)
```

PHP `array_diff_key()` function

- The `array_diff_key()` function in PHP is used to compare the keys of two or more arrays and return the key-value pairs from the first array whose keys are not present in any of the other arrays. It performs a key-based comparison and returns an array containing the differences.
- It's syntax is as follows:

```
array_diff_key( array $array1, array $array2, array ...$arrays ) : array
```

- Consider this example:

```
$array1 = [ "a" => "Apple", "b" => "Banana", "c" => "Cherry", "d" => "Date" ];
$array2 = [ "b" => "Banana", "c" => "Date" ];
$difference = array_diff_key( $array1, $array2 );
print_r( $difference ); // Output: Array ( [a] => Apple [d] => Date )
```

PHP Spread Operator (. . .)

- The spread operator (. . .) in PHP is used to unpack arrays or objects into individual elements. It allows you to expand an array or object into a list of arguments when calling a function or creating a new array.
- The spread operator can be used in the following contexts:
  - Function Calls
  - Array Creation

Using the Spread Operator in Function Calls

- When calling a function, you can use the spread operator to pass the elements of an array as individual arguments to the function. Example

```
function sum($a, $b, $c) {
    return $a + $b + $c;
}
```

```
$numbers = [1, 2, 3];
sum(...$numbers); // Equivalent to sum(1, 2, 3)
```

- If the number of elements in the array does not match the number of parameters in the function, a `TypeError` will be thrown.
- If only one parameter is expected in the function, it only works with the first element of the array. Example

```
$even = [2, 4, 6, 8];

function multiply_by_2($n) {
    return $n * 2;
}

echo multiply_by_2(...$even);
// Output: 4 because only the first element (2) is used
```

Using the Spread Operator in Array Creation

- You can use the spread operator to create a new array by unpacking elements from existing arrays. Example

```
$array1 = [1, 2, 3];
$array2 = [4, 5, 6];
$new_array = [...$array1, ...$array2]; // Result: [1, 2, 3, 4, 5, 6]
```

- Consider another example

```
$fruits = ['Apple', 'Banana'];
$foods = ['Carrot', 'Broccoli', ...$fruits];
print_r($foods); // Output: Array ( [0] => Carrot [1] => Broccoli [2] => Apple [3]
=> Banana )
```

Note that the spread operator maintains the order of elements when unpacking arrays.

Using spread operator with a generator

- In the following example, first, we define a generator that returns even numbers between 2 and 10. Then, we use the spread operator to spread out the returned value of the generator into an array:

```
<?php

function even_number()
{
```

```
    for($i =2; $i < 10; $i+=2){
        yield $i;
    }
}

$even = [...even_number()];

print_r($even);
/* Output:
(
    [0] => 2
    [1] => 4
    [2] => 6
    [3] => 8
)*/
```

- How it works:
    - The `even_number()` function is a generator that yields even numbers from 2 to 8. The `yield` keyword allows the function to return a value and pause its execution, maintaining its state for the next call.
    - When we use the spread operator (`...`) with the generator function call `even_number()`, it iterates over the yielded values and unpacks them into a new array. The `even_number()` generator produces the values one by one, and the spread operator collects them into the `$even` array.
    - The resulting `$even` array contains all the even numbers generated by the `even_number()` function.

Using spread operator with a Traversable object

- PHP allows you to apply the spread operator not only to an array but also to any object that implements the Traversable interface. For example:

```
<?php

class RGB implements IteratorAggregate
{
    private $colors = ['red', 'green', 'blue'];

    public function getIterator(): Traversable
    {
        return new ArrayIterator($this->colors);
    }
}

$rgb = new RGB();
$colors = [...$rgb];

print_r($colors);
```

How it works:

- The `RGB` class implements the `IteratorAggregate` interface, which requires the implementation of the `getIterator()` method. This method returns an instance of `ArrayIterator`, which allows iteration over the `$colors` array.
- When we create an instance of the `RGB` class and use the spread operator (`...`) with it, PHP calls the `getIterator()` method to obtain the iterator for the object. The spread operator then iterates over the colors provided by the iterator and unpacks them into a new array.
- The resulting `$colors` array contains the colors 'red', 'green', and 'blue'.

Spread Operator and named arguments

- You can call a function using named arguments. Example

```php
<?php


function format_name(string $first, string $middle, string $last): string
{
    return $middle ?
        "$first $middle $last" :
        "$first $last";
}

echo format_name(
    first: 'John',
    middle: 'V.',
    last: 'Doe'
); // John V. Doe
```

By this, the order of the arguments does not matter.

- Consider the following as a general syntax for using named arguments

```
function_name(paramter_name1: value1, parameter_name2: value2, ...);
```

Note that the dollar sign `$` is not used when specifying the parameter names.

- The spread operator can be used along with named arguments as follows:

```php
<?php


function format_name(string $first, string $middle, string $last): string
{
    return $middle ?
        "$first $middle $last" :
```

```
        "$first $last";
}


$names = [
  'middle' => 'V.',
    'first' => 'John',
    'last' => 'Doe'
];

echo format_name(...$names); // John V. Doe
```

Here, the keys of the associative must correspond to the parameter names of the function.

PHP rand() function

- The `rand(min, max)` function generates a random integer between the specified minimum and maximum values. Example

```
$min = 1;
$max = 10;
$random_number = rand($min, $max);
echo $random_number; // Output: A random integer between 1 and 10 including 1 and
10
```

PHP List

- The `list()` function in PHP is used to assign values from an array to a list of variables in a single operation. It is often used in conjunction with the `array` construct or when working with functions that return arrays.
- The syntax of the `list()` function is as follows:

```
list( $var1, $var2, ... ) = array;
```

where,

- `$var1`, `$var2`, etc. are the variables that will receive the values from the array.
- `array` is the array from which the values will be extracted.

Using a list to skip array elements

- The following example uses the list to assign the first and the third element to variables. Here, it skips the second element:

```php
$prices = [100, 0.1, 0.05];

list($buy_price, , $discount) = $prices;
echo "The price is $buy_price with the discount of $discount";
// Output: The price is 100 with the discount of 0.05
```

Using the nested list to assign variables

- The following example uses a nested list to assign variables from a multidimensional array:

```php
$elements = ['body', ['white','blue']];
list($element, list($bgcolor, $color)) = $elements;

var_dump($element, $bgcolor, $color);
/* Output:
string(4) "body"
string(5) "white"
string(4) "blue"
*/
```

Using PHP List with an Associative Array

- From PHP 7.1, you can use the `list()` function with associative arrays by specifying the keys of the array elements you want to assign to variables. Example

```php
$person = [
    'first_name' => 'John',
    'last_name' => 'Doe',
    'age' => 25
];

list(
    'first_name' => $first,
    'last_name' => $last,
    'age' => $age) = $person;

var_dump($first, $last, $age);
/* Output:
string(4) "John"
string(3) "Doe"
int(25)
*/
```

Here, we specify the keys of the associative array in the `list()` function to assign their corresponding values to the variables `$first`, `$last`, and `$age`.

- This allows for more flexibility when working with associative arrays, as you can directly map the keys to variables without relying on their order in the array.
- The following can be considered as the general syntax for using `list()` with an associative array:

```
list(
    'key1' => $var1,
    'key2' => $var2,
    ...
) = $associative_array;
```

PHP Array Destructuring

- Array destructuring in PHP allows you to extract values from an array and assign them to individual variables in a concise manner. This feature is new and was introduced in PHP 7.1 and provides a more readable way to work with arrays.
- The syntax for array destructuring is similar to the `list()` function but uses square brackets `[]` instead of the `list()` keyword. Example

```
[$var1, $var2, ...] = $array;
```

- For Associative arrays, you can use the following syntax:

```
['key1' => $var1, 'key2' => $var2, ...] = $associative_array;
```

- Example of PHP Array Destructuring using an Associative Array:

```
$person = [
    'first_name' => 'John',
    'last_name' => 'Doe',
    'age' => 25
];

['first_name' => $first, 'last_name' => $last, 'age' => $age] = $person;
var_dump($first, $last, $age);
/* Output:
string(4) "John"
string(3) "Doe"
int(25)
*/
```

- Example of PHP Array Destructuring using an Indexed Array:

```php
$coordinates = [10, 20, 30];

[$x, $y, $z] = $coordinates;
var_dump($x, $y, $z);
/* Output:
int(10)
int(20)
int(30)
*/
```

Skipping Elements during Destructuring

- You can skip elements during destructuring by leaving the corresponding variable position empty. Example

```php
$values = [100, 200, 300];

[$first, , $third] = $values;
var_dump($first, $third);
/* Output:
int(100)
int(300)
*/
```

Swapping Variables using Destructuring

- Generally, to swap the values of two variables, you would need a temporary variable as follows:

```php
$a = 5;
$b = 10;
$temp = $b;
$b = $a;
$a = $temp;

echo "a: $a, b: $b"; // Output: a: 10, b: 5
```

- But with array destructuring, you can swap the values of two variables in a single line without needing a temporary variable. Example

```php
$a = 5;
$b = 10;

[$a, $b] = [$b, $a];
echo "a: $a, b: $b"; // Output: a: 10, b: 5
```

Here, we create an array with the values of `$b` and `$a`, and then destructure it back into `$a` and `$b`, effectively swapping their values.

PHP `parse_url()` function

- The `parse_url()` function in PHP is used to parse a URL and return its components as an associative array. This function breaks down the URL into its various parts, such as scheme, host, port, path, query, and fragment.

```
$url = "https://www.example.com:8080/path/to/resource?query=php#
section1";
$components = parse_url($url);
print_r($components);
/*
Output:
Array
(
    [scheme] => https
    [host] => www.example.com
    [port] => 8080
    [path] => /path/to/resource
    [query] => query=php
    [fragment] => section1
)
*/
```

PHP Array `sort()` function

- The `sort()` function in PHP is used to sort the elements of an array in ascending order. This function modifies the original array and sorts its elements based on their values.
- The syntax of the `sort()` function is as follows:

```
sort( array &$array, int $flags = SORT_REGULAR ) : bool
```

where,

- `array` is the input array that you want to sort.
- `flags` (optional) is a parameter that determines the sorting behavior. It can take various values such as `SORT_REGULAR`, `SORT_NUMERIC`, `SORT_STRING`, etc. The default is `SORT_REGULAR`.

Sorting an array of strings

- The following example demonstrates how to sort an array of strings using the `sort()` function:

```
$fruits = [ "Banana", "Orange", "Mango" ];
sort( $fruits );
/* Output:
Array
(
    [0] => Banana
    [1] => Mango
    [2] => Orange
)
*/
```

Here, sorting is done according to the ASCII values of the characters in the strings.Also, the function use the `SORT_STRING` flag by default when sorting strings. By this flag, the array elements will be considered as strings during the sorting process.

Sorting an array of strings case-insensitively

- To sort an array of strings in a case-insensitive manner, you can use the `SORT_FLAG_CASE` flag along with the `SORT_STRING` flag. Example:

```
$fruits = ['apple', 'Banana', 'orange'];
sort($fruits, SORT_STRING | SORT_FLAG_CASE);

print_r($fruits);
/* Output:
Array
(
    [0] => apple
    [1] => Banana
    [2] => orange
)
*/
```

without the `SORT_FLAG_CASE` flag, the sorting would be case-sensitive, and uppercase letters would be sorted before lowercase letters based on their ASCII values.

```
$fruits = ['apple', 'Banana', 'orange'];
sort($fruits, SORT_STRING);
/* Output:
Array
(
    [0] => apple
    [1] => Banana
    [2] => orange
)
*/
```

Sorting an array using Natural Ordering

- To sort an array using natural ordering, you can use the `SORT_NATURAL` flag along with the `sort()` function. Natural ordering sorts strings in a way that is more intuitive for humans, especially when dealing with numbers within strings. Example:

```
  $files = ['file10.txt', 'file2.txt', 'file1.txt'];
  sort($files, SORT_NATURAL);
  print_r($files);
/* Output:
Array
(
    [0] => file1.txt
    [1] => file2.txt
    [2] => file10.txt
)
*/
```

Here, the `SORT_NATURAL` flag ensures that the files are sorted in a way that considers the numeric values within the strings, resulting in a more human-friendly order. Without the `SORT_NATURAL` flag, the sorting would be based on ASCII values, leading to an incorrect order:

```
  $files = ['file10.txt', 'file2.txt', 'file1.txt'];
  sort($files);
  print_r($files);
/* Output:
Array
(
    [0] => file1.txt
    [1] => file10.txt
    [2] => file2.txt
)
*/
```

Here, the sorting is done based on ASCII values, which places 'file10.txt' before 'file2.txt' because '1' comes before '2' in ASCII order.

Sorting an array of mixed data types

- When sorting an array that contains mixed data types (e.g., integers, strings, floats), the `sort()` function uses either the `SORT_STRING` or `SORT_NUMERIC` flag based on the types of the elements in the array. By default, it uses `SORT_REGULAR`, which compares the elements based on their values without considering their types.

Using `SORT_STRING` flag

- When using the `SORT_STRING` flag, all elements are treated as strings during the sorting process

- Boolean valeu `true` is converted to the string `1` while `false` is converted to an empty string `""`.

Example:

```
$mixed = [10, '5', 3.14, 'apple', 2, '20', 'banana', true, .5];
sort($mixed, SORT_STRING);
print_r($mixed);
/* Output:
Array ( [0] => 0.5 [1] => 1 [2] => 10 [3] => 2 [4] => 20 [5] => 3.14 [6] => 5 [7]
=> apple [8] => banana )
*/
```

Using `SORT_NUMERIC` flag

- When using the `SORT_NUMERIC` flag, all elements are treated as numbers during the sorting process. Non-numeric strings are considered to the integer `0` thus, they appear first. Example:

```
$mixed = [10, '5', 3.14, 'apple', 2, '20', 'banana', true, .5];
sort($mixed, SORT_STRING);
print_r($mixed);
/* Output:
Array ( [0] => apple [1] => banana [2] => 0.5 [3] => 1 [4] => 2 [5] => 3.14 [6] =>
5 [7] => 10 [8] => 20 )
*/
```

PHP `rsort()` function

- The `rsort()` function in PHP is used to sort the elements of an array in descending order. This function modifies the original array and sorts its elements based on their values.
- The syntax of the `rsort()` function is as follows:

```
rsort( array &$array, int $flags = SORT_REGULAR ) : bool
```

where,

- `array` is the input array that you want to sort.
- `flags` (optional) is a parameter that determines the sorting behavior. It can take various values such as `SORT_REGULAR`, `SORT_NUMERIC`, `SORT_STRING`, etc. The default is `SORT_REGULAR`.

- Example of using `rsort()` function:

```
$numbers = [ 5, 2, 8, 1, 4 ];
rsort( $numbers );
/* Output:
Array
```

```
(
    [0] => 8
    [1] => 5
    [2] => 4
    [3] => 2
    [4] => 1
)
*/
```

- `rsort()` function is similar to `sort()` function only with the exception that `rsort()` sorts in descending order

PHP `ksort()` function

- The `ksort()` function sorts elements by keys. Mainly useful for associative arrays
- It's syntax is as follow:

```
ksort( array &$array, int $flags = SORT_REGULAR ) : bool
```

- To combine the flag values, you can use the `|` operator. Example:

```
$fruits = [
    "b" => "Banana",
    "a" => "apple",
    "c" => "Orange"
];
ksort( $fruits, SORT_STRING | SORT_FLAG_CASE );
print_r( $fruits );
/* Output:
Array
(
    [a] => apple
    [b] => Banana
    [c] => Orange
)
*/
```

Here, the `ksort()` function sorts the associative array `$fruits` by its keys in a case-insensitive manner using the `SORT_STRING` and `SORT_FLAG_CASE` flags. The resulting array is sorted alphabetically by the keys 'a', 'b', and 'c'.

PHP `krsort()` function

- The `krsort()` function in PHP is used to sort the elements of an array by their keys in descending order. This function modifies the original array and sorts its elements based on their keys.

- The syntax of the `krsort()` function is as follows:

```
krsort( array &$array, int $flags = SORT_REGULAR ) : bool
```

- Example of using `krsort()` function:

```
$fruits = [
    "b" => "Banana",
    "a" => "apple",
    "c" => "Orange"
];
krsort( $fruits );
print_r( $fruits );
/* Output:
Array
(
    [c] => Orange
    [b] => Banana
    [a] => apple
)
*/
```

PHP `usort()` function

- The `usort()` function in PHP is used to sort an array by values using a user-defined comparison function. This function allows you to define your own sorting logic based on specific criteria.
- The syntax of the `usort()` function is as follows:

```
usort( array &$array, callable $callback ) : bool
```

where,

- `array` is the input array that you want to sort.
- `callback` is a user-defined comparison function that determines the sorting order. This function should take two parameters (elements from the array to compare) and return:
    - A negative value if the first element should come before the second element.
    - A positive value if the first element should come after the second element.
    - Zero if both elements are considered equal.

Sorting an array of numbers

- Consider this example where we sort an array of numbers in ascending order using `usort()`:

```php
$numbers = [2, 4, 36, 43, 9, 2, 0];

usort($numbers, function ($x, $y) {
    if($x === $y) {
        return 0;
    }

    return $x < $y ? -1 : 1;
});

print_r($numbers);
/* Output:
Array
(
    [0] => 0
    [1] => 2
    [2] => 2
    [3] => 4
    [4] => 9
    [5] => 36
    [6] => 43
)
*/
```

How it works:

- The `usort()` function takes the `$numbers` array and a comparison function as arguments.
  - The comparison function compares two elements `$x` and `$y` from the array.
  - If `$x` is equal to `$y`, it returns `0`, indicating that they are equal.
  - If `$x` is less than `$y`, it returns `-1`, indicating that `$x` should come before `$y`.
  - If `$x` is greater than `$y`, it returns `1`, indicating that `$x` should come after `$y`.
- The `usort()` function uses this comparison logic to sort the `$numbers` array in ascending order.

- To sort in descending order, you can modify the comparison function as follows:

```php
$numbers = [2, 4, 36, 43, 9, 2, 0];
usort($numbers, function ($x, $y) {
    if($x === $y) {
        return 0;
    }

    return $x > $y ? -1 : 1;
});
print_r($numbers);
/* Output:
Array
(
    [0] => 43
    [1] => 36
    [2] => 9
```

```
    [3] => 4
    [4] => 2
    [5] => 2
    [6] => 0
)
*/
```

- From PHP 7.0, you can use the spaceship operator (`<=>`) to simplify the comparison function as follows:

```php
$numbers = [2, 4, 36, 43, 9, 2, 0];
usort($numbers, function ($x, $y) {
    return $x <=> $y; // Ascending order
});
print_r($numbers);
/* Output:
Array
(
    [0] => 0
    [1] => 2
    [2] => 2
    [3] => 4
    [4] => 9
    [5] => 36
    [6] => 43
)
*/
```

Here, the spaceship operator (`<=>`) compares `$x` and `$y` and returns `-1`, `0`, or `1` based on their relationship, making the code more concise.

- If the callback is simple, you can use an arrow function like this

```php
$numbers = [2, 4, 36, 43, 9, 2, 0];
usort($numbers, fn($x, $y) => $x <=> $y);
print_r($numbers);
/* Output:
Array
(
    [0] => 0
    [1] => 2
    [2] => 2
    [3] => 4
    [4] => 9
    [5] => 36
    [6] => 43
)
*/
```

How it works:

- The arrow function `fn($x, $y) => $x <=> $y` is a concise way to define the comparison logic for sorting.
- It compares two elements `$x` and `$y` from the array using the spaceship operator (`<=>`), which returns:
  - A negative value if `$x` is less than `$y`.
  - A positive value if `$x` is greater than `$y`.
  - Zero if both elements are equal.

Sorting an Array of String by Length

- Consider this example where we sort an array of strings by their length using `usort()`:

```
$names = [ 'Alex', 'Peter',  'John' ];
usort($names, fn($x,$y) => strlen($x) <=> strlen($y));

var_dump($names);
/* Output:
Array
(
    [0] => Alex
    [1] => John
    [2] => Peter
)
*/
```

Sorting an Array of Objects

- Consider this example where we sort an array of objects by a specific property using `usort()`:

```
class Person
{
    public $name;
    public $age;

    public function __construct(string $name, int $age)
    {
        $this->name = $name;
        $this->age = $age;
    }
}

$group = [
    new Person('Bob', 20),
    new Person('Alex', 25),
    new Person('Peter', 30),
];

usort($group, fn($x, $y) => $x->age <=> $y->age);
```

```
print_r($group);
/* Output:
Array
(
    [0] => Person Object
        (
            [name] => Bob
            [age] => 20
        )

    [1] => Person Object
        (
            [name] => Alex
            [age] => 25
        )

    [2] => Person Object
        (
            [name] => Peter
            [age] => 30
        )

)
*/
```

How it works:

- The `usort()` function takes the `$group` array and a comparison function as arguments.
- The comparison function compares the `age` property of two `Person` objects `$x` and `$y`.
- The spaceship operator (`<=>`) is used to compare the ages, allowing the `usort()` function to sort the array of `Person` objects in ascending order based on their age.

Using a Static Method as a callback

- A Static Method is a method that belongs to the class itself rather than to any specific instance of the class. You can use a static method as a callback function in `usort()`. Example:

```
class Person
{
    public $name;

    public $age;

    public function __construct(string $name, int $age)
    {
        $this->name = $name;
        $this->age = $age;
    }
}
```

```
class PersonComparer
{
    public static function compare(Person $x, Person $y)
    {
        return $x->age <=> $y->age;
    }
}

$group = [
    new Person('Bob', 20),
    new Person('Alex', 25),
    new Person('Peter', 30),
];

usort($group, ['PersonComparer', 'compare']);

print_r($group);
```

Here, we define a static method `compare` in the `PersonComparer` class that compares the `age` property of two `Person` objects. We then pass this static method as a callback to the `usort()` function to sort the array of `Person` objects by age.

- Know that, the `usort()` function sorts an array (indexed or associative) and doesn't preserve the keys. Rather, it reindexes the element

PHP asort

- The `asort()` function sorts the elements (values) of an associative array in ascending order. Unlike sort functions, `asort()` maintains the index association
- It's syntax is as follow

```
asort(array &$array, int $flags = SORT_REGULAR) : bool
```

Example

```
$fruits = [
    "b" => "Banana",
    "a" => "Apple",
    "c" => "Orange"
];
asort( $fruits, SORT_STRING );
print_r( $fruits );
/* Output:
Array
(
    [a] => Apple
    [b] => Banana
```

```
    [c] => Orange
)
*/
```

PHP `arsort()` function

- The `arsort()` function in PHP is used to sort the elements (values) of an associative array in descending order while maintaining the index association.
- The syntax of the `arsort()` function is as follows:

```
arsort( array &$array, int $flags = SORT_REGULAR ) : bool
```

PHP `uasort()` function

- The `uasort()` function in PHP is used to sort an array by values using a user-defined comparison function while maintaining the index association (That is, the keys are preserved). This function allows you to define your own sorting logic based on specific criteria.
- The syntax of the `uasort()` function is as follows:

```
uasort( array &$array, callable $callback ) : bool
```

where

- `array` is the input array that you want to sort.
- `callback` is a user-defined comparison function that determines the sorting order. This function should take two parameters (elements from the array to compare) and return:
    - A negative value if the first element should come before the second element.
    - A positive value if the first element should come after the second element.
    - Zero if both elements are considered equal.
- Example of using `uasort()` function:

```
$countries = [
    'China' => ['gdp' => 12.238 , 'gdp_growth' => 6.9],
    'Germany' => ['gdp' => 3.693 , 'gdp_growth' => 2.22 ],
    'Japan' => ['gdp' => 4.872 , 'gdp_growth' => 1.71 ],
    'USA' => ['gdp' => 19.485, 'gdp_growth' => 2.27 ],
];

// sort the country by GDP
uasort($countries, function ($x, $y) {
    return $x['gdp'] <=> $y['gdp'];
});
```

```
// show the array
foreach ($countries as $name => $stat) {
    echo "$name has a GDP of {$stat['gdp']} trillion USD with a GDP growth rate of
{$stat['gdp_growth']}%" . '<br>';
}
/* Output:
Germany has a GDP of 3.693 trillion USD with a GDP growth rate of 2.22%
Japan has a GDP of 4.872 trillion USD with a GDP growth rate of 1.71%
China has a GDP of 12.238 trillion USD with a GDP growth rate of 6.9%
USA has a GDP of 19.485 trillion USD with a GDP growth rate of 2.27%
*/
```

PHP `uksort()` function

- The `uksort()` function in PHP is used to sort an array by keys using a user-defined comparison function while maintaining the index association (That is, the values are preserved). This function allows you to define your own sorting logic based on specific criteria.
- The syntax of the `uksort()` function is as follows:

```
uksort( array &$array, callable $callback ) : bool
```

- Example of using `uksort()` function:

```
$fruits = [
    "B" => "Banana",
    "a" => "Apple",
    "c" => "Orange"
];

uksort($fruits, fn ($x, $y) => $x <=> $y);
print_r($fruits);
/* Output:
Array ( [B] => Banana [a] => Apple [c] => Orange )
*/
```

PHP Arrow functions

- Arrow functions, introduced in PHP 7.4, provide a more concise syntax for defining anonymous functions (also known as closures). They are particularly useful for short functions that consist of a single expression.
- The syntax of an arrow function is as follows:

```
fn (parameter_list) => expression;
```

where,

- `parameter_list` is a comma-separated list of parameters that the function takes.
- `expression` is a single expression that the function evaluates and returns.
- An advantage of arrow functions is that they automatically capture variables from the surrounding scope without needing to use the `use` keyword. Example

```
$multiplier = 2;
$numbers = [1, 2, 3, 4, 5];

$multiplied = array_map(fn($n) => $n * $multiplier, $numbers);

print_r($multiplied);
/* Output:
Array ( [0] => 2 [1] => 4 [2] => 6 [3] => 8 [4] => 10 )
*/
```

- A limitation of arrow functions is that they can only contain a single expression and cannot have multiple statements or a block of code. If you need to perform more complex operations, you should use traditional anonymous functions.

PHP `isset()`

- The `isset()` is a language construct in PHP is used to determine if a variable is set and is not `NULL`. It returns `true` only if the variable exists and has a value other than `NULL`, and `false` if the variable does not exist or exist but has its value set to `null`.
- The syntax of the `isset()` function is as follows:

```
isset( mixed $var, mixed ...$vars ) : bool
```

where,

- `var` is the variable to be checked.
- `vars` (optional) is additional variables to be checked.

PHP `isset()` with multiple variables

- When using `isset()` with multiple variables, it returns `true` only if all the specified variables are set and not `NULL`. If any of the variables is not set or is `NULL`, it returns `false`.
- Example of using `isset()` with multiple variables:

```php
$var1 = "Hello";
$var2 = 42;
$var3 = null;

if (isset($var1, $var2, $var3)) {
    echo "All variables are set and not null.";
} else {
    echo "One or more variables are not set or are null.";
}

/* Output:
One or more variables are not set or are null.
*/
```

Also, one an unset variable is found, the preceding variable are not evaluated

PHP `empty()` construct

- The `empty()` construct accepts a variable and returns true is the variable is empty. Otherwise, it returns `false`
- It's syntax is as follows

```php
empty(mixed $var) : bool
```

- A variable is said to be empty if it does not exist or if its value equal `false` (or equal to any of false, (int) 0, (float) 0.0, -0.0, (string) "0", (array) [], or NULL)
- The `empty($var)` is essentially the same as this expression that uses `isset()` and equality operator `==`

```php
!isset($v) ||  $v == false
```

Example of using `empty()` construct:

```php
$var1 = "";
if (empty($var1)) { //true
    echo "The variable is empty.";
} else {
    echo "The variable is not empty.";
}
/* Output:
The variable is empty.
*/
```

- `empty()` is a language construct in PHP not a function.
- `empty()` will not throw an error even though the input variable isn't defined
- In practice, you use the `empty()` construct in the situation where you're unsure if a variable exists.

For example, suppose you receive an array `$data` from an external source, e.g., an API call or a database query.

To check if the `$data` array has an element with the key 'username' and it is not empty, and you may use the following expression:

```
if (!empty($data['username'])) {
    // The 'username' key exists and is not empty
    $username = $data['username'];
} else {
    // The 'username' key does not exist or is empty
    $username = 'Guest';
}
```

Similarly, this can be done with `isset()` as follows:

```
if (isset($data['username']) && $data['username']) {
    // The 'username' key exists and is not empty
    $username = $data['username'];
} else {
    // The 'username' key does not exist or is empty
    $username = 'Guest';
}
```

- `isset()` and `empty()` are not opposite of each other

PHP `is_null()`

- The `is_null()` construct accepts a variable and returns true if it's value equals `null`. Otherwise, it returns `false`

```
is_null(mixed $var): bool
```

- `is_null()` also returns `true` if a variable does not exist. Afterwards, it isuses a warning

```
var_dump(is_null($undefined_var)); // true with a warning
```

The warning is as follows:

```
Notice: Undefined variable: $undefined_var in ...
```

PHP `is_null()` with array

- `is_null()` returns `false` if a certain key does not exist in an array. Example:

```
$data = ['name' => null];
var_dump(is_null($data['age'])); // false
```

Thereafter, we have this error

```
Notice: Undefined index: link
```

PHP Anonymous function

- An anonymous function (also known as a closure or lambda function) is a function that is defined without a name. Anonymous functions are often used as callback functions or for creating functions on the fly.
- The syntax of an anonymous function is as follows:

```
function ($paramter_1, $parameter_2, ...) {
    // function body
};
```

- Since the function has no name, it is typically assigned to a variable or passed as an argument to another function.

Anonymous function assigned to a variable

- Example of an anonymous function assigned to a variable:

```
$sum = function($a, $b) {
  return $a + $b;
};

// Calling the function
echo $sum(10, 21); // Output: 31
```

- If we dump the variable `$sum`, we get the following output:

```
var_dump($sum);
/* Output:
object(Closure)#1 (1) { ["parameter"]=> array(2) { ["$a"]=> string(10) "" ["$b"]=>
string(10) "" } }
*/
```

- That output indicates that the variable `$sum` holds an instance of the `Closure` class, which represents the anonymous function. Note: `Closure` in PHP is not the same as the closure in other programming languages such as JavaScript or Python.

Passing an anonymous function as an argument

- Example of passing an anonymous function as an argument to another function:

```
echo '<br>';
print_r($numbers);
echo '<br>';

function sort_by_size($x, $y) {
    return $x <=> $y;
}

usort($numbers, 'sort_by_size');

echo '<br>';
print_r($numbers);
echo '<br>';
/* Output:
Array ( [0] => 1932 [1] => 1003 [2] => 3428 [3] => 253 [4] => 901 )
Array ( [0] => 253 [1] => 901 [2] => 1003 [3] => 1932 [4] => 3428 )
```

Notice that when the function was to be passed as an argument, we used its name and wrapped it in quotes.

- The above is basically passing a function as an argument to another function
- Due to the fact that the function `sort_by_size` may be used once, we can define it as an anonymous function as follows:

```
usort($numbers, function($x, $y) {
    return $x <=> $y;
});
```

Scope of the anonymous function

- By default, an anonymous function cannot access variables from its parent scope
- To fix this, we use the `use` construct
- Example of using the `use` construct to access variables from the parent scope:

```
$name = "John Doe";

$greet = function() use ($name) {
    return "Hello, $name!";
};
echo $greet(); // Output: Hello, John Doe!
```

- Variables passed using the `use` construct are passed by value. To pass them by reference, prefix the variable name with an ampersand (`&`)
- Example of passing variables by value using the `use` construct:

```
$count = 0;
$increment = function() use ($count) {
    $count++;
    return $count;
};
echo $increment(); // Output: 1
echo $count; // Output: 0
```

- Example of passing variables by reference using the `use` construct:

```
$count = 0;
$increment = function() use (&$count) {
    $count++;
    return $count;
};

echo $increment(); // Output: 1
echo $count; // Output: 1
```

Here, by using the `&` symbol before `$count` in the `use` construct, we pass `$count` by reference. As a result, when we increment `$count` inside the anonymous function, it also updates the value of `$count` in the parent scope.

- An Anonymous function such that you can access variables outside its scope is called a Closure

Returning an anonymous function from another function

- Example of returning an anonymous function from another function:

```php
function multiplier($x)
{
    return function ($y) use ($x) {
        return $x * $y;
    };
}

$double = multiplier(2);
echo $double(100) . '<br>'; // 200

$tripple = multiplier(3);
echo $tripple(100) . '<br>'; // 300
```

How it works:

- The `multiplier` function takes a parameter `$x` and returns an anonymous function.
- The returned anonymous function takes a parameter `$y` and uses the `use` construct to access the `$x` variable from the parent scope.
- When we call `multiplier(2)`, it returns an anonymous function that multiplies its input by `2`. We assign this function to the variable `$double`.
- Similarly, when we call `multiplier(3)`, it returns an anonymous function that multiplies its input by `3`. We assign this function to the variable `$tripple`.

PHP Arrow function

- Arrow functions were introduced in PHP 7.4 as a more concise syntax for anonymous functions.
- It's syntax is as follows:

```php
fn ($parameter_1, $parameter_2, ...) => expression;
```

where,

- `parameter_1, parameter_2, ...` is a comma-separated list of parameters that the function takes.
- `expression` is a single expression that the function evaluates and returns. It can't take more than one Notice that there is no need for the `return` keyword.
- The above syntax is the same as the below Anonymous function

```php
function ($parameter_1, $parameter_2, ...) {
    return expression;
};
```

- Unlike anonymous functions, arrow functions can also access variables from the parent scope using the `use` construct. However, arrow functions automatically capture variables from the parent scope by value, so there is no need to explicitly use the `use` construct.

- Also, arrow function needs to either be assigned to a variable or passed explicitly as an argument to another function.

PHP Variable Functions

- In PHP, variable functions allow you to call a function using a variable that contains the name of the function as a string. This feature provides flexibility in function calls, enabling dynamic function invocation based on runtime conditions.
- To use variable functions, you simply assign the name of the function to a variable and then use that variable followed by parentheses to call the function.

Using variable functions to call a method

- The variable functions allow you to call the methods of an object. The syntax for calling a method using a variable function is as follows:

```
$this->$variable($arguments)
```

where,

- `$variable` is a variable that contains the name of the method as a string.
- `$arguments` are the arguments to be passed to the method.

- Example of using variable functions to call a method:

```
class Calculator {
    public function add($a, $b) {
        return $a + $b;
    }

    public function subtract($a, $b) {
        return $a - $b;
    }
}
$calc = new Calculator();
$operation = 'add';
$result = $calc->$operation(10, 5);
echo $result; // Output: 15
$operation = 'subtract';
$result = $calc->$operation(10, 5);
echo $result; // Output: 5
```

- Know that variable functions will not work with PHP language constructs such as `echo`, `isset()`, `empty()`, etc. Example

```php
$func = 'isset';
$var = null;
var_dump($func($var)); // This will result in a fatal error
```

PHP `callable` type

- In PHP, the `callable` type is a special type that represents a function or method that can be called. It can be a string containing the name of a function, an array containing an object and method name, or a closure (anonymous function).
- Basically when a function is used as argument to another function and it is called within that function, it is called a callback function and it is introduced using the `callable` type.
- The `callable` type is often used as a type hint in function or method parameters to indicate that the argument should be a callable function or method.
- Consider this example as one of the use-case of a callable type is as follows

```php
function sum(callable $callback, ...$x) {
    return $callback($x);
}

echo sum('array_sum', 2, 3, 4); // Output: 9
```

- Consider another example

```php
function multiply(callable $callback, $x) {
    return $callback($x);
}

function double($y) {
    return $y * 2;
}

function triple($y) {
    return $y * 3;
}

function quad($y) {
    return $y * 4;
}

echo multiply('double', 4), '<br>'; // Output: 8
echo multiply('triple', 4), '<br>'; // Output: 12
echo multiply('quad', 4), '<br>'; // Output: 16
```

How it works:

- The `multiply` function takes a callable function as its first parameter and a number as its second parameter.
- It then calls the provided callable function with the number as an argument and returns the result.
- The functions `double`, `triple`, and `quad` are defined to perform different multiplication operations.
- When we call the `multiply` function with different callable functions (`double`, `triple`, and `quad`), it executes the corresponding multiplication logic and returns the results.
- Rather than using strings to represent the function names, you can also use anonymous functions or arrow functions as callable arguments. Example:

```
function multiply(callable $callback, $x) {
    return $callback($x);
}

echo multiply(fn($y) => $y * 2, 4), '<br>'; // Output: 8
echo multiply(fn($y) => $y * 3, 4), '<br>'; // Output: 12
echo multiply(fn($y) => $y * 4, 4), '<br>'; // Output: 16
```

Closure vs Callable

- A `Closure` is a specific type of `callable` that represents an anonymous function (a function without a name) that can capture variables from its surrounding scope. In PHP, closures are instances of the `Closure` class.
- On the other hand, `callable` is a broader type that encompasses any valid function or method that can be called, including named functions, static methods, and closures.
- The above means that all closures are callables, but not all callables are closures.
- Also the previous code snippet can be modified for closures as follows

```
function multiply(Closure $callback, $x) {
    return $callback($x);
}

echo multiply(fn($y) => $y * 2, 4), '<br>'; // Output: 8
echo multiply(fn($y) => $y * 3, 4), '<br>'; // Output: 12
echo multiply(fn($y) => $y * 4, 4), '<br>'; // Output: 16
```

By this, all callbacks that must be passed into the `multiply()` function must be an anonymous function not a named function else, we get a fatal error.

PHP `is_callable()` function

- The `is_callable()` function in PHP is used to check if a variable can be called as a function. It returns `true` if the variable is a valid callable (i.e., a function or method that can be invoked), and `false` otherwise.

- The syntax of the `is_callable()` function is as follows:

```
is_callable( mixed $var, bool $syntax_only = false, string &$callable_name = null
) : bool
```

where,

- `var` is the variable to be checked.
- `syntax_only` (optional) is a boolean parameter that, when set to `true`, checks only the syntax of the callable without verifying its existence.
- `callable_name` (optional) is a string variable that, if provided, will be set to the name of the callable if it is valid.

- Example of using `is_callable()` function:

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

$func = 'greet';
if (is_callable($func)) {
    echo $func('John'); // Output: Hello, John!
} else {
    echo "The function is not callable.";
}
```

PHP call_user_func_array()

- The `call_user_func_array()` function in PHP is used to call a callback function with an array of parameters. This function is particularly useful when you want to pass a variable number of arguments to a function or when the arguments are stored in an array.
- The syntax of the `call_user_func_array()` function is as follows:

```
call_user_func_array(callable $callback, array $args): mixed
```

where,

- `callback` is the function to be called. It can be a string containing the function name, an array containing an object and method name, or a closure.
- `args` is an array of parameters to be passed to the callback function.

- Example of using `call_user_func_array()` function:

```
$evens = [2, 4, 6];
```

```
echo call_user_func_array(fn ($x, $y, $z) => $x + $y, $evens);
// Output: 6
```

How it works:

- The `call_user_func_array()` function takes a callback function (in this case, an arrow function that sums two numbers) and an array of arguments (`$evens`).
- It calls the callback function with the elements of the `$evens` array as individual arguments. Here, only the first two elements are sum since the callback requires only two parameters

PHP `array_map()` function

- The `array_map()` function in PHP is used to apply a callback function to each element of one or more arrays. It returns a new array containing the results of applying the callback function to each element.
- The syntax of the `array_map()` function is as follows:

```
array_map(callable $callback, array $array1, array ...$arrays): array
```

where,

- `callback` is the function to be applied to each element of the array(s). It can be a string containing the function name, an array containing an object and method name, or a closure.
- `array1, arrays` are the input arrays to be processed. You can pass multiple arrays, and the callback function will receive corresponding elements from each array as arguments.
- Know that whenever you pass a single array as an argument to the `array_map()` function, the callback function should accept only one parameter. If you pass multiple arrays, the callback function should accept as many parameters as there are arrays. By this, you can perform operations that involve multiple arrays.
- Also, know that if a single array is passed, keys are preserved in the resulting array. However, if multiple arrays are passed, the resulting array will have re-indexed keys starting from `0`.
- Also, if you passing multiple arrays of different lengths, the resulting array will have a length equal to the longest input array and the last elements of the resulting array will be `0` since the last elements of the shorter arrays are considered as `null`

Using the PHP array_map() function with an array of objects

- The following code snippet defines a class with three properties `id`, `username`, and `email`. It then creates an array of objects of that class and uses the `array_map()` function to extract the `email` property from each object in the array.

```
class User {
    public $username;

    public $email;
```

```
    public $id;

    public function __construct (int $id, string $username, string $email) {
        $this -> id = $id;
        $this -> username = $username;
        $this -> email = $email;
    }
};

$users = [
    new User(1, 'John Doe', 'johndoe@gmail.com'),
    new User(2, 'Harray Matthews', 'harraymat@gmail.com'),
    new User(3, 'Collins Fox', 'collfox@gmail.com')
];

$usernames = array_map(fn ($user) => $user -> username, $users);

echo '<br>';
print_r($usernames);
echo '<br>';
/* Output:
Array ( [0] => John Doe [1] => Harray Matthews [2] => Collins Fox )
*/
```

Using a static method as a callback in array_map()

- You can also use a static method as a callback function in the `array_map()` function. Example:

```
class Square {
    public $length;

    public static function area($length) {
        return $length ** 2;
    }
};

$lengths = [2, 3, 4];

$areas = array_map('Square::area', $lengths);

echo '<br>';
print_r($areas);
echo '<br>';
/* Output:
Array ( [0] => 4 [1] => 9 [2] => 16 )
*/
```

How it works:

- The `Square` class defines a public static method `area()` that calculates the area of a square given its length.
- An array `$lengths` is created containing the lengths of three squares.
- The `array_map()` function is used to apply the static method `Square::area` to each element of the `$lengths` array.
- The result is stored in the `$areas` array, which contains the areas of the squares.

- Note that the syntax for passing a public static method to the array_map() function is as follows:

```
'className::staticMethodName'
```

Passing `null` as callback to `array_map()`

- When `null` is passed as the callback function to the `array_map()` function, it returns an array containing an individual array of elements from each of the input arrays whose keys tally. Example:

```
$evens = [2, 4];
$odds = [1, 3, 5];

echo '<br>';
print_r(array_map(null, $evens, $odds));
echo '<br>';
/*
Output:
Array ( [0] => Array ( [0] => 2 [1] => 1 ) [1] => Array ( [0] => 4 [1] => 3 ) [2]
=> Array ( [0] => [1] => 5 ) )
*/
```

PHP `array_filter()` function

- The `array_filter()` function in PHP is used to filter elements of an array using a callback function. It returns a new array containing only the elements that satisfy the condition defined in the callback function.
- The syntax of the `array_filter()` function is as follows:

```
array_filter(array $array, ?callable $callback = null, int $mode = 0): array
```

where,

- `array` is the input array to be filtered.
- `callback` (optional) is a function that defines the filtering condition. It should return `true` for elements that should be included in the result and `false` for elements that should be excluded. If no callback is provided, all truthy values will be included.

- `mode` (optional) is a flag that determines what arguments are passed to the callback function. It can take the following values:
  - `ARRAY_FILTER_USE_KEY`: Passes the key of each element to the callback function.
  - `ARRAY_FILTER_USE_BOTH`: Passes both the value and the key of each element to the callback function.
  - Default is `0`, which means only the value of each element is passed to the callback function.
- Example of using `array_filter()` function:

```
$codes = ['#$288IF1', '#$102OJ9', '#$504XD0', '#$192PL0', '#$453ND4', '#$548DA6'];

$filtered_codes = array_filter($codes, fn ($x) => $x % 2 === 0 ? true : false,
ARRAY_FILTER_USE_KEY);

echo '<br>';
print_r($filtered_codes);
echo '<br>';
/* Output:
Array ( [0] => #$288IF1 [2] => #$504XD0 [4] => #$453ND4 )
*/
```

How it works:

- The `array_filter()` function takes the `$codes` array and a callback function as arguments.
- The callback function checks if the key of each element is even using the modulo operator (`%`). If the key is even, it returns `true`, indicating that the element should be included in the result; otherwise, it returns `false`.
- The `ARRAY_FILTER_USE_KEY` flag is used to indicate that the keys of the array

Using a method of a class as callback

- The following example demonstrates the use of a class method as a callback function in the `array_filter()` function:

```
$indexes = [1, 2, 3, 4, 5, 6];

class Odd {
    public function isOdd($x) {
        return !($x % 2 === 0);
    }
}

$filtered_indexes = array_filter($indexes, [new Odd, 'isOdd']);

echo '<pr>';
print_r($filtered_indexes);
echo '<pr>';
/* Output:
```

```
Array ( [0] => 1 [2] => 3 [4] => 5 )
*/
```

How it works:

- The `Odd` class defines a method `isOdd()` that checks if a number is odd.
- An instance of the `Odd` class is created using `new Odd`.
- The `array_filter()` function is called with the `$indexes` array and the method `isOdd` of the `Odd` class instance as the callback function.

- The following syntax can be considered as a general syntax for using a class method as a callback in the `array_filter()` function

```
array_filter($array, [new ClassName, 'methodName']);
```

Using the static method of a class as callback

- The following demonstrates the use of a static method as the callback of the `array_filter()` function:

```
$indexes = [1, 2, 3, 4, 5, 6];

class Odd {
    public static function isOdd($x) {
        return !($x % 2 === 0);
    }
}

$filtered_indexes = array_filter($indexes, ['Odd', 'isOdd']);

echo '<pr>';
print_r($filtered_indexes);
echo '<pr>';
/* Output:
Array ( [0] => 1 [2] => 3 [4] => 5 )
*/
```

A general syntax for this can be considered as follows

```
array_filter($array, ['ClassName', 'staticMethodName']);
```

Using a magic method as callback

- From PHP 5.3, you can necessarily use the `__invoke()` magic method as a callable as follows

```php
$mixed_numbers = [-2, -4, 5, -2, 5, 5, 3];

class Positive {
    public function __invoke($number) {
        return $number > 0;
    }
}

$filtered_numbers = array_unique(array_filter($mixed_numbers, new Positive()));

echo '<pre>';
print_r($filtered_numbers);
echo '<pre>';
/* Output:
Array
(
    [2] => 5
    [6] => 3
)
*/
```

How it works:

- The `Positive` class defines the `__invoke()` magic method, which checks if a number is positive.
- An instance of the `Positive` class is created using `new Positive()`.
- The `array_filter()` function is called with the `$mixed_numbers` array and the `Positive` class instance as the callback function.
- The `array_unique()` function is then used to remove duplicate positive numbers from the filtered result.
- The result is stored in the `$filtered_numbers` array, which contains only unique positive numbers.
- The following syntax can be considered as a general syntax for using a magic method as a callback in the `array_filter()` function

```php
array_filter($array, new ClassName());
```

For a magic method, you just need to call an instance of the class since the `__invoke()` method is called automatically when the object is used as a function.

Passing elements to the callback function

- By default, the `array_filter()` function passes only the values of the array to the callback function. However, you can change this behavior by using the `mode` parameter.
- The `mode` parameter can take the following values:
    - `ARRAY_FILTER_USE_KEY`: Passes the key of each element to the callback function.
    - `ARRAY_FILTER_USE_BOTH`: Passes both the value and the key of each element to the callback function.
- Example of using the `mode` parameter in the `array_filter()` function:

```
$codes = ['#$288IF1', '#$102OJ9', '#$504XD0', '#$192PL0', '#$453ND4', '#$548DA6'];
$filtered_codes = array_filter($codes, fn ($x, $y) => $y % 2 === 0 ? true : false,
ARRAY_FILTER_USE_KEY);
echo '<br>';
print_r($filtered_codes);
echo '<br>';
/* Output:
Array ( [0] => #$288IF1 [2] => #$504XD0 [4] => #$453ND4 )
*/
```

- Example using `ARRAY_FILTER_USE_BOTH` mode:

```
$codes = ['#$288IF1', '#$102OJ9', '#$504XD0', '#$192PL0', '#$453ND4', '#$548DA6'];
$filtered_codes = array_filter($codes, fn ($x, $y) => $y % 2 === 0 && strlen($x) >
7, ARRAY_FILTER_USE_BOTH);
echo '<br>';
print_r($filtered_codes);
echo '<br>';
/* Output:
Array ( [0] => #$288IF1 [2] => #$504XD0 )
*/
```

How it works:

- In the first example, the `ARRAY_FILTER_USE_KEY` mode is used, so the callback function receives the key of each element as the second parameter (`$y`). The function checks if the key is even and returns `true` for even keys, resulting in a filtered array containing only elements with even keys.
- In the second example, the `ARRAY_FILTER_USE_BOTH` mode is used, so the callback function receives both the value (`$x`) and the key (`$y`) of each element. The function checks if the key is even and if the length of the value is greater than 7. It returns only those elements that satisfy both conditions.
- `$x` parameter represents the values while `$y` parameter represents the keys of the array elements.
- Know that this `ARRAY_FILTER_USE_BOTH` allows the keys to be preserved in the filtered array. By this if only the element is used to check for the condition, and the returned array is printed after filtering, the keys of the original array are preserved in the filtered array.

```
$codes = ['#$88IF1', '#$102OJ9', '#$504XD0', '#$192L0', '#$453ND4', '#$548DA6'];

echo '<br>';
print_r($codes); // Ouput the original array
echo '<br>';

$filtered_codes = array_filter($codes, fn ($x, $y) => strlen($x) > 7,
ARRAY_FILTER_USE_BOTH);
echo '<br>';
print_r($filtered_codes);
echo '<br>';
/*
```

```
Ouput:
Array ( [0] => #$88IF1 [1] => #$102OJ9 [2] => #$504XD0 [3] => #$192L0 [4] =>
#$453ND4 [5] => #$548DA6 )
Array ( [1] => #$102OJ9 [2] => #$504XD0 [4] => #$453ND4 [5] => #$548DA6 )
*/
```

Keys are preserved in this case. Suppose you want to reindex the filtered array, you can use the `array_values()` function as follows:

```
$reindexed_filtered_codes = array_values($filtered_codes);
```

- Know that if no callback function is provided to the `array_filter()` function, it will remove all falsy values from the array. Falsy values include `false`, `0`, `0.0`, `""` (empty string), `NULL`, and empty arrays.

PHP `array_reduce()` function

- The `array_reduce()` function reduces the values of an array into a single value by iteratively applying a callback function to each element of the array.
- Suppose you have an array of numbers and you want to calculate their sum. You can use the `array_reduce()` function to achieve this.
- The syntax of the `array_reduce()` function is as follows:

```
array_reduce(array $array, callable $callback, mixed $initial = null): mixed
```

where,

- `array` is the input array to be reduced.
- `callback` is a function that defines how to reduce the array. It should take two parameters: the accumulator (the current reduced value) and the current array element. The function should return the updated accumulator.
- `initial` (optional) is the initial value of the accumulator (which is the first paramter of the callback function). If not provided, the first element of the array is used as the initial value.

- Example of using `array_reduce()` function:

```
$words = ['Sage', 'The', 'Warrior'];

$smashed_words = array_reduce($words, fn ($previous, $current) => $previous .
$current);

echo '<br>';
echo $smashed_words;
echo '<br>';
```

```
/* Output: SageTheWarrior
*/
```

How it works:

- The `array_reduce()` function takes the `$words` array and a callback function as arguments.
- The callback function concatenates the `previous` accumulated string with the `current` word from the array.
- The result is stored in the `$smashed_words` variable, which contains the concatenated string "SageTheWarrior".

- The `previous` parameter is special. It should not be manipulated when passed to the callback function. It holds the accumulated value from the previous iteration.
- Consider this example

```
$carts = [
    ['item' => 'A', 'qty' => 2, 'price' => 10],
    ['item' => 'B', 'qty' => 3, 'price' => 20],
    ['item' => 'C', 'qty' => 5, 'price' => 30]
];

$total_item = array_reduce($carts, fn($x, $y) => $x['qty'] + $y['qty']);

echo '<br>';
echo $total_item;
echo '<br>';
```

The above code will throw an error since in the first iteration, the `$x` parameter will be equal to the first element of the array which is an array itself. To fix this, we need to do the following:

```
$total_item = array_reduce($carts, fn($x, $y) => $x + $y['qty']);
```

- If the input array is empty and the `initial` parameter is not provided, the `array_reduce()` function will return `NULL`. Example:

```
$empty_array = [];
$result = array_reduce($empty_array, fn($x, $y) => $x + $y);
var_dump($result); // Output: NULL
```

- However, if the `initial` parameter is provided, the `array_reduce()` function will return the value of the `initial` parameter. Example:

```
$empty_array = [];
$result = array_reduce($empty_array, fn($x, $y) => $x + $y, 0);
```

```
var_dump($result); // Output: int(0)
```

PHP `include` construct

- The `include` construct in PHP is used to include (or load) and evaluate a specified file during the execution of a script. It allows you to reuse code from other files, making your code more modular and easier to maintain.
- It's syntax is as follows:

```
include 'file.php';
```

where,

- `file.php` is the path to the file you want to include. It can be a relative or absolute path.
- When the `include` construct is encountered, PHP will read the specified file and execute its code as if it were part of the calling file. If it cannot find the specified file, it will generate a warning but continue executing the rest of the script.

```
Warning: include(functions.php): failed to open stream: No such file or directory
in ... on line ..
Warning: include(): Failed opening 'functions.php' for inclusion
(include_path='\xampp\php\PEAR') in ... on line ..
```

- General structure for a PHP project

```
.
├── index.php
├── functions.php
├── inc
│   ├── footer.php
│   └── header.php
└── public
    ├── css
    │   └── style.css
    └── js
        └── app.js
```

Thus, by convention, you include the `header.php` and `footer.php` files in the `inc` directory

PHP include & variable scopes

- When you include a file, all the variables defined in that file inherit the variable scope of the line on which the include occurs. They becomes a global variable in the file that includes it.

- Example of using `include` construct and variable scopes:
- For example, the following defines the $title and $content variables in the `functions.php`:

```
<?php

// functions.php

$title = 'PHP include';
$content = 'This shows how the PHP include construct works.';
```

When you include the functions.php in the index.php file, the $title and $content variables become the global variables in the index.php file. And you can use them as follows:

```
<?php include 'inc/header.php'; ?>


<?php include_once 'functions.php'; ?>

<h1><?php echo $title; ?></h1>
<p><?php echo $content; ?></p>


<?php include 'inc/footer.php'; ?>
```

Notice that the file containing the reusable variable is included before it is used in that file

- But if the `functions.php` file is included in a function, the variables defined in the `functions.php` file will be local to that function only. Example:

```
<?php include 'inc/header.php'; ?>
<?php include_once 'functions.php'; ?>

<?php
    function render_article()
    {
        include 'functions.php';

        return "<article>
                    <h1>$title</h1>
                $content
                </article>";
    }

    echo render_article();
?>
```

```php
<?php include 'inc/footer.php'; ?>
```

In this case, the `$title` and `$content` variables are local to the `render_article()` function and cannot be accessed outside of it.

- Also, It's important to note that all functions, classes, interfaces, and traits defined in the included file will have a global scope.

PHP `include_once` construct

- The `include_once` construct in PHP is similar to the `include` construct, but it ensures that the specified file is included only once during the execution of a script. If the file has already been included, it will not be included again and it returns `true`
- Simply put, the `include_once` loads the file just once, regardless of how many times the file is included.
- Suppsose you have a file named `functions.php` and you want to include it in multiple places in your script. Using `include_once` ensures that the file is included only once, preventing potential issues such as function redefinitions or variable overwrites.
- It's syntax is as follows:

```php
include_once 'file.php';
```

Adding `include_once 'file.php'` again in that sam file won't neccessarily yeild an error. Instead, it will simply return `true` without including the file again.

- Example of using `include_once` construct:

```php
<?php
include_once 'functions.php';
include_once 'functions.php'; // This will not include the file again
echo 'Functions file included successfully.';
?>
/* Output:
Functions file included successfully.
*/
```

- `include` and `include_once` returns `1` on successful inclusion of the file and `false` on failure.

include vs include_once

- The main difference between `include` and `include_once` is that `include_once` checks if the file has already been included, and if so, it does not include it again if it has been once included. This is useful to prevent issues such as function redefinitions or variable overwrites when the same file is included multiple times.

PHP `require` construct

- The `require` construct in PHP is used to include and evaluate a specified file during the execution of a script. It is similar to the `include` construct, but with one key difference: if the specified file cannot be found or included, `require` will generate a fatal error and halt the execution of the script.
- It's syntax is as follows:

```
require 'file.php';
```

- Practically, you often use the `require` construct to load the code from libraries. Since the libraries contain the required functions to execute the script, it’s better to use the `require` construct than the `include` construct.
- PHP `require` is not a function, but a language construct. Sometimes, the `require`construct can be used similarly like functions as follows

```
require('file.php');
```

The above is the same as `require 'file.php'`. The parenthesis present in this expression are optional. They are not part of the `require` construct syntax. Instead they belong to the file path expression that is being loaded

PHP `require_once` construct

- Similar to the `include_once` construct, the `require_once` construct in PHP ensures that the specified file is included only once during the execution of a script. If the file has already been included, it will not be included again. It is different from `include_once` such that if the file cannot be found, it will generate a fatal error and halt the execution of the script.

require vs require_once

- The main difference between `require` and `require_once` is that `require_once` checks if the file has already been included, and if so, it does not include it again if it has been once included. This is useful to prevent issues such as function redefinitions or variable overwrites when the same file is included multiple times.

include vs require

- The main difference between `include` and `require` is how they handle errors when the specified file cannot be found or included.
- `include` generates a warning and continues executing the rest of the script.
- `require` generates a fatal error and halts the execution of the script.

Use-cases

- Best use-cases for include and require is for code readability and structure

- Suppose you want to include the content of a PHP file in a string, you can use the `include` construct within an output buffer as follows:

```
ob_start();
include 'file.php';
$content = ob_get_clean();

echo $content;
```

How it works: - The `ob_start()` function starts output buffering, which means that any output generated by the included file will be captured in the buffer instead of being sent to the browser. - The `include 'file.php';` statement includes the specified file, and any output generated by that file is captured in the output buffer. - The `ob_get_clean()` function retrieves the contents of the output buffer and clears it. The captured output is then assigned to the `$content` variable. - Thus the `$content` variable can be treated as a string

PHP `__DIR__`

- The `__DIR__` magic constant in PHP returns the directory of the file in which it is used. It provides an absolute path to the directory, making it useful for including files or working with file paths relative to the current script's location.
- The syntax of the `__DIR__` magic constant is as follows:

```
__DIR__
```

Technically, `__DIR__` is equivalent to the following expression:

```
dirname(__FILE__)
```

- Suppose you have a project directory with the following structure

```
.
├── inc
│   ├── footer.php
│   └── header.php
└── index.php
```

- If you use the `__DIR__` magic constant in the `index.php` file, it will return the absolute path to the directory containing the `index.php` file. Example:

```php
// index.php
echo __DIR__;
/* Output:
C:\path\to\your\project
*/
```

- If you use the `__DIR__` magic constant in the `header.php` file, it will return the absolute path to the directory containing the `header.php` file. Example:

```php
// inc/header.php
echo __DIR__;
/* Output:
C:\path\to\your\project\inc
*/
```

Use DIR when:

- You are including a file that is in a different directory than the script that's running.
- Your file might be included from multiple locations in your project (like header.php being used in public/index.php and admin/dashboard/index.php).
- You want to make your includes robust and portable.

```php
include __DIR__ . '/inc/header.php';
```

By this, irrespective of where the script is being run from, the correct path to `header.php` will always be resolved.

You don't really need DIR when:

- The file you're including is in the same directory as the script you're currently running.
- You are in a very small project where the paths are simple and you won't move files around.
- `__DIR__` allows you to reference files relative to the included script, not where the initial application execution is

PHP Variables variables

- In PHP, variable variables allow you to use the value of a variable as the name of another variable. This means that you can dynamically create and access variables based on the values of other variables.
- It's use-case includes scenarios where you want to create variable names dynamically based on user input, configuration settings, or other runtime conditions.
- The syntax for variable variables is as follows:

```
$$variable_name
```

where,

- `$variable_name` is the name of the variable whose value will be used as the name of another variable.
- Example of using variable variables:

```
$var_name = 'greeting';
$$var_name = 'Hello, World!';
echo $greeting; // Output: Hello, World!
```

How it works:

- The variable `$var_name` is assigned the string value `'greeting'`.
- The variable variable `$$var_name` is then created, which is equivalent to `$greeting`. It is assigned the value `'Hello, World!'`.
- Finally, when we echo `$greeting`, it outputs the value `'Hello, World!'

PHP mail() function

- The `mail()` function in PHP is used to send emails directly from a script. It allows you to send simple text-based emails or HTML emails with attachments.
- On Linux or Unix systems, you can configure the `mail()` function to use the `sendmail` or `Qmail` program to send messages
- On Windows, you can install the `sendmail` and set the `sendmail_path` in `php.ini` file to point at the executable file.
- However, it's more convenient to set the SMTP (Simple Mail Transfer Protocol) server with a port and sendmail_from in the php.ini file on Windows like this:

```
[mail function]
SMTP = smtp.yourisp.com
smtp_port = 25
sendmail_from = youremail@yourdomain.com
```

where,

- `smtp.yourisp.com` is the SMTP server of your email provider. Your email provider is your ISP (Internet Service Provider) in most cases which is the company that provides you with internet access.
- `25` is the port number used by the SMTP server. The default port for SMTP is `25`. However, some email providers may use different ports for secure connections, such as `465` for SSL or `587` for TLS.
- `youremail@yourdomain.com` is the email address that will appear in the "From" field of the emails sent using the `mail()` function.

- If the SMTP server requires authentication, you can add the following lines for the account to authenticate:

```
auth_username=smtp_user
auth_password=smpt_password
```

where,

- `smtp_user` is the username for the SMTP server authentication.
- `smtp_password` is the password for the SMTP server authentication.

- Once the configuration is ready, you need to restart the webserver.
- The syntax of the `mail()` function is as follows:

```
mail(
    string $to,
    string $subject,
    string $message,
    array|string $additional_headers = [],
    string $additional_params = ""
): bool
```

where

- `$to` is the reciever of the email
- `$subject` is the subject of the email
- `$message` is the body of the email. It can be plain text or HTML. If $message is plain text, you use a CRLF (\r\n) to separate lines. Each line should not exceed 70 character
- `$additional_headers` (optional) is used to specify additional headers for the email, such as "From", "Cc", "Bcc", and "Reply-To". It can be a string or an array of strings. If the header comes from an untrusted source, you should always sanitize it for security
- `$additional_params` allows you to pass additional flags as the command-line options to the sendmail program The `mail()` function returns `true` if the mail was accepted for delivery. It doesn't mean that the mail is successfully reached the intended receiver. If an error occurred, the `mail()` function return `false`

Using the PHP mail() function to send a plain text email example

```
<?php

$subject = 'This is a test email';

$message = <<<MSG
    Hi,
    This is a simple email.
    It's sent from PHP.
```

```
MSG;

wordwrap($message, 70, "\r\n");

mail('contact@phptutorial.net', $subject, $message);
```

How it works:

- The `$subject` variable is assigned the subject of the email.
- The `$message` variable is assigned the body of the email using a heredoc syntax for better readability.
- The `wordwrap()` function is used to ensure that the lines in the message do not exceed 70 characters, which is a recommended practice for email formatting. It manipulates the `$message` variable directly.
- The `mail()` function is called with the recipient's email address, subject, and message to send the email.

Using the PHP mail() function to send a mail with extra headers example

- The following example uses the mail() function to send a mail with additional headers like From, Reply-To, and X-Mailer

```
<?php

$to      = 'contact@phptutorial.net';
$subject = 'This is a test email';
$message = 'Hi there';

$headers[] = 'From: john.doe@example.com';
$headers[] = 'Reply-To: john.doe@example.com';
$headers[] = 'X-Mailer: PHP/' . phpversion();


mail($to, $subject, $message, implode('\r\n', $headers));
```

From PHP 7.2 or later, you can pass the headers as an array like this:

```
<?php

$to      = 'contact@phptutorial.net';
$subject = 'This is a test email';
$message = 'Hi there';

$headers = [
    'From' => 'john.doe@example.com',
    'Reply-To' => 'john.doe@example.com',
    'X-Mailer' => 'PHP/' . phpversion()
];
```

```
mail($to, $subject, $message,  $headers);
```

Using the PHP mail() function to send HTML email example

- To send HTML mail, you need to set the `Content-type` for the header like this:

```
<?php

$to      = 'contact@phptutorial.net';
$subject = 'This is a test email';
$message = '<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Email</title>
</head>

<body>

    <h1>This is HTML mail</h1>

</body>

</html>';


$headers = [
    'MIME-Version' => '1.0',
    'Content-type' => 'text/html; charset=utf8',
    'From' => 'john.doe@example.com',
    'Reply-To' => 'john.doe@example.com',
    'X-Mailer' => 'PHP/' . phpversion()
];


if (mail($to, $subject, $message,  $headers)) {
    echo 'email was sent.';
} else {
    echo 'An error occurred.';
}
```

Guide: Using PHP mail() with Gmail SMTP on XAMPP (Windows)

1. Overview

Goal: Use PHP's `mail()` on XAMPP (Windows) to send email through Gmail's `SMTP server` (via XAMPP's sendmail helper), instead of trying to send to `localhost:25`.

2. Prepare your Gmail account

1. Turn on 2-Step Verification (2FA) for your Google account.
2. Generate a Gmail app password:

- Go to your Google Account → Security.
- With 2-Step Verification enabled, open App passwords. You can necessarily search for it
- Create a new app password (e.g. "XAMPP PHP mail". It can be anything) and copy the 16-character code. The code usually comes with spaces - use it without those spaces

This app password is what you use in config, not your normal Gmail password.

3. Configure php.ini in XAMPP

File: `C:\xampp\php\php.ini`

- Find the `[mail function]` section.
- Comment out direct localhost mail settings if they exist:

```
text
[mail function]
;SMTP=localhost
;smtp_port=25
```

- Set the sendmail_path (make sure the path is correct for your XAMPP):

```
text
sendmail_path = "C:\xampp\sendmail\sendmail.exe -t"
```

- Enable OpenSSL extension (remove ; at the start if present):

```
text
extension=openssl
; or on some XAMPP versions:
;extension=php_openssl.dll  -> change to: extension=php_openssl.dll
```

- Save the file.
- Restart Apache from XAMPP Control Panel after every change to php.ini.

4. Configure sendmail.ini in XAMPP

File: `C:\xampp\sendmail\sendmail.ini`

- Set Gmail SMTP and auth details (example):

```
text
smtp_server = smtp.gmail.com
smtp_port = 587
smtp_ssl = tls

auth_username = your_email@gmail.com
auth_password = your_app_password
```

Notes:

auth_username is your Gmail address.

auth_password is the app password from step 2 (without spaces), not your normal Gmail password.

- Save the file when done.

5. Write a simple test script

File: e.g. `C:\xampp\htdocs\testmail.php`

```
php
<?php
$to      = "receiver@example.com"; // where you want to receive the email
$subject = "XAMPP mail() test";
$message = "Hello, this is a test email from XAMPP.";
$headers = "From: your_email@gmail.com\r\n";

if (mail($to, $subject, $message, $headers)) {
    echo "Mail sent";
} else {
    echo "Mail failed";
}
```

Important details:

- From: should usually match the Gmail you used in auth_username.
- \r\n is used as line ending in headers.
- $to can be any address you can check (another email or even the same Gmail).

6. Test and debug

- Start Apache and MySQL in XAMPP.
- Open `http://localhost/testmail.php` in your browser.
- If it prints “Mail sent”:
- Check the inbox of receiver@example.com (also check Spam).
- If mail() returns false:

```
php
$result = mail($to, $subject, $message, $headers);
if (!$result) {
    var_dump(error_get_last());
}
```

Use error_get_last() to see any internal error message.

7. Security and best practices

- Never share or commit your app password (sendmail.ini, php.ini) to GitHub or anywhere public.
- For real projects, consider using a mail library (like PHPMailer, Symfony Mailer, etc.) with SMTP directly; it gives better error messages, HTML email support, attachments, etc.
- Keep this guide as a reference; when something stops working, re-check:
    - php.ini mail section
    - sendmail.ini SMTP settings
    - Gmail app password and 2FA status

That Apache was restarted after changes

If you like, you can paste this into a text file like xampp-mail-guide.txt and store it alongside your XAMPP projects.

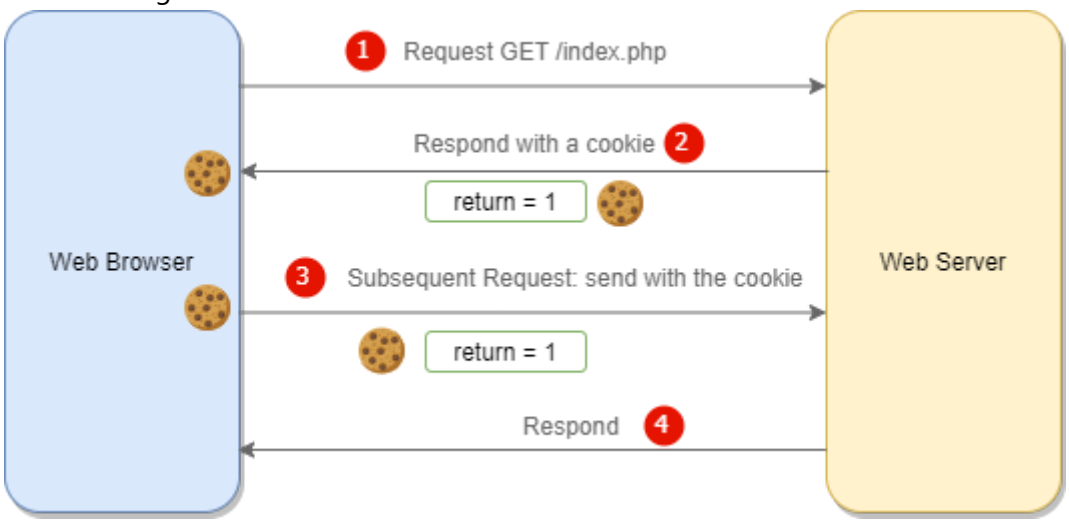
PHP Cookies

- The web works based on HTTP protocol. The HTTP protocol is stateless that is it doesn’t remember anything about the previous requests.
- When a web browser requests a page from a web server, the web server responds with the page content. Later, if the same web browser requests the same page again, and the web server has no information that the request is from the same web browser
- Cookies solve this stateless challenge
- A cookie is a small piece of data that a web server sends to a web browser to check if two requests come from the same web browser. The web browser stores the cookie and sends it back to the web

server with each subsequent request to the same server. This allows the web server to remember information about the user, such as their preferences, login status, and other data.
- Cookies are also known as web cookies, HTTP cookies, or browser cookies.

Why cookies?

- Cookies are commonly used for:
  - Session management: Cookies can be used to store session IDs, allowing users to remain logged in as they navigate through different pages of a website.
  - Personalization: Cookies can store user preferences, such as language settings or theme choices, to enhance the user experience.
  - Tracking: Cookies can be used to track user behavior across different websites for analytics and advertising purposes. For example, on an E-commerce website, you can use cookies to record the products that users previously viewed. Later, you can use this information to recommend the related products that users might be interested in.
- The following illustrates how cookies work:



How it works:

- When a user visits a website, the web server sends a cookie to the user's web browser.
- The web browser stores the cookie and sends it back to the web server with each subsequent request to the same server.
- The web server can then use the information stored in the cookie to remember the user's preferences, login status, and other data. Technically speaking:
  - First, the web browser sends a request to the web server. Suppose the web server doesn't have any information about the web browser. The web server creates a cookie with a name `return` and a value `1` and attaches the cookie to the HTTP response header. To create a cookie, you'll use the `setcookie()` function.
  - Second, the web browser stores the cookie.
  - Third, the web browser sends the second request with the stored cookie in the header of the HTTP request to the web server. On the web server, PHP can access the cookie via the `$_COOKIE` superglobal variable and do something accordingly
  - Finally, the web server responds with the content of the request. Typically, it responds to the web browser with the content based on the value of the cookie.

- A web browser can store a cookie with a maximum size of 4KB, but this limit varies between browsers.
- A cookie has an expiration date. Typically, web browsers store cookies for a specific duration, and the web server can specify the expiration time for a cookie.

- A cookie also stores the web address (URL) that indicates the URL that created the cookie. The web browser can send back the cookie that was originally set by the same URL. In other words, a website won't be able to read a cookie set by other websites.
- Most modern web browsers allow users to choose to accept cookies. Therefore, you should not wholly rely on cookies for storing critical data.

Setting a cookie in PHP

- The `setcookie()` function allows you to send HTTP header to create a cookie on a browser.
- It's syntax is as follows:

```
<?php

setcookie (
    string $name ,
    string $value = "" ,
    int $expires = 0 ,
    string $path = "" ,
    string $domain = "" ,
    bool $secure = false ,
    bool $httponly = false
): bool
```

where,

- `name` is the name of the cookie.
- `value` is the value of the cookie. It is stored on the user's computer. You can store any string value in a cookie. However, you should avoid storing sensitive information in cookies since cookies are stored on the user's computer.
- `expires` is the expiration time of the cookie. It is specified as a Unix timestamp. If this parameter is set to `0` or not set, the cookie will expire at the end of the session (when the browser closes). You can use the `time()` function to get the current time and add the number of seconds you want the cookie to last. For example, to set a cookie that expires in one hour, you can use `time() + 3600`.
- `path` specifies the path on the server where the cookie will be available. If set to `/`, the cookie will be available within the entire domain. If set to `/folder/`, the cookie will only be available within that folder and its subfolders.
- `domain` specifies the domain that the cookie is available to. If not specified, it defaults to the host of the current document.

```
https://example.com/page1.html    → host = "example.com"
```

- `secure` indicates whether the cookie should only be transmitted over a secure HTTPS connection. If set to `true`, the cookie will only be sent over HTTPS. Also, a secured HTTPS connection means the data between your browser and the website is encrypted using SSL/TLS, so no one can read or tamper with your cookies, passwords, or other information while it's being transmitted.

- `httponly` indicates whether the cookie should only be accessible through the HTTP protocol. If set to `true`, the cookie will not be accessible via JavaScript, which can help mitigate certain types of cross-site scripting (XSS) attacks.
- From PHP 7.3.0, you can also set cookies using an array of options as follows:

```
setcookie (
    string $name ,
    string $value = "" ,
    array $options = [] ) : bool
```

where,

- `options` is an associative array that can contain the following keys: `expires`, `path`, `domain`, `secure`, `httponly`, and `samesite`. The `samesite` can take a value of `None`, `Lax`, or `Strict`. It is . If you use any other key, the setcookie() function will raise a warning.
- The `setcookie()` function must be called before any output is sent to the browser.
- The setcookie() function returns true if it successfully executes. Notice that it doesn't indicate whether the web browser accepts the cookie. The setcookie() function returns false if it fails.

`$_COOKIE` superglobal variable

- The `$_COOKIE` superglobal variable in PHP is an associative array that contains all the cookies that have been sent to the server by the client's web browser. Each cookie is represented as a key-value pair, where the key is the name of the cookie and the value is the value of the cookie.
- To access a cookies by name, you use the following syntax

```
$_COOKIE['cookie_name'];
```

- Suppose the cookie name has spaces or dots `.`, you replace them by an underscore `_`
- To check if a cookies is set, you use the `isset()` function

```
if(isset($_COOKIE['cookie_name'])) {
    // cookie is set
} else {
    // cookie is not set
}
```

- `$_COOKIE` is a superglobal variable, which means it is accessible from any scope within a PHP script, including functions and classes, without needing to use the `global` keyword.

Reading a Cookie

- Before reading a cookie, you should check if the cookie is set using the `isset()` function to avoid undefined index errors.

```
if(isset($_COOKIE['cookie_name'])) {
    // process the cookie
}
```

- To check if a cookie equal a value, you use the following syntax

```
if(isset($_COOKIE['cookie_name']) && $_COOKIE['cookie_name'] === 'value') {
    // cookie is set and equals 'value'
} else {
    // cookie is not set or does not equal 'value'
}
```

Deleting a Cookie

- If you don't want to use a cookie, you can force the browser to delete it. PHP doesn't provide a function that directly deletes a cookie. However, you can delete a cookie using the setcookie() function by setting the expiration date to the past
- The following code deletes a cookie with the cookie_name in the subsequent page request:

```
unset($_COOKIE['cookie_name']);
setcookie('cookie_name', null, time()-3600);
```

How it works:

- The `unset()` function removes the cookie from the `$_COOKIE` superglobal variable in the current script execution.
- The `setcookie()` function is called with the cookie name, a null value, and an expiration time set to one hour in the past (`time() - 3600`). This instructs the browser to delete the cookie.

- Note that deleting a cookie using the `setcookie()` function will only take effect in the next page request. The cookie will still be available in the current script execution until the page is reloaded or a new request is made.
- Example of setting and reading a cookie

```
const TIME = 60;

$returning_user = false;

if (isset($_COOKIE['return'])) {
    $returning_user = true;
} else {
```

```
    setcookie('return', '1', time() + TIME);
}

echo $returning_user ? 'Welcome back!' : 'Welcome to my website';
```

How it works:

- The code first defines a constant `TIME` with a value of `60` seconds.
- It initializes a variable `$returning_user` to `false`.
- It checks if a cookie named `return` is set using the `isset()` function.
    - If the cookie is set, it means the user has visited the website before, so it sets `$returning_user` to `true`. Thus, for any request made by the browser (such as refreshing the page) within the next 60 seconds, the user will be considered a returning user.
    - If the cookie is not set, it means the user is visiting for the first time. The code then sets a cookie named `return` with a value of `1` and an expiration time of `60` seconds from the current time using the `setcookie()` function. If the page is the requested by the client again, the PHP script runs and `returning_user` will be set to `true`.
- Finally, it uses a ternary operator to display a welcome message based on whether the user is a returning user or not. If `$returning_user` is `true`, it displays "Welcome back!"; otherwise, it displays "Welcome to my website".

- Consider the following example that demonstrates how to set, read, and delete a cookie in PHP:

```
// Set a cookie
setcookie('user', 'John Doe', time() + 3600); // Expires in 1 hour
echo 'Cookie has been set.<br>';
// Read the cookie
if (isset($_COOKIE['user'])) {
    echo 'Hello, ' . $_COOKIE['user'] . '!<br>';

} else {
    echo 'Cookie is not set.<br>';
}
// Delete the cookie
unset($_COOKIE['user']);
setcookie('user', '', time() - 3600);
echo 'Cookie has been deleted.<br>';
```

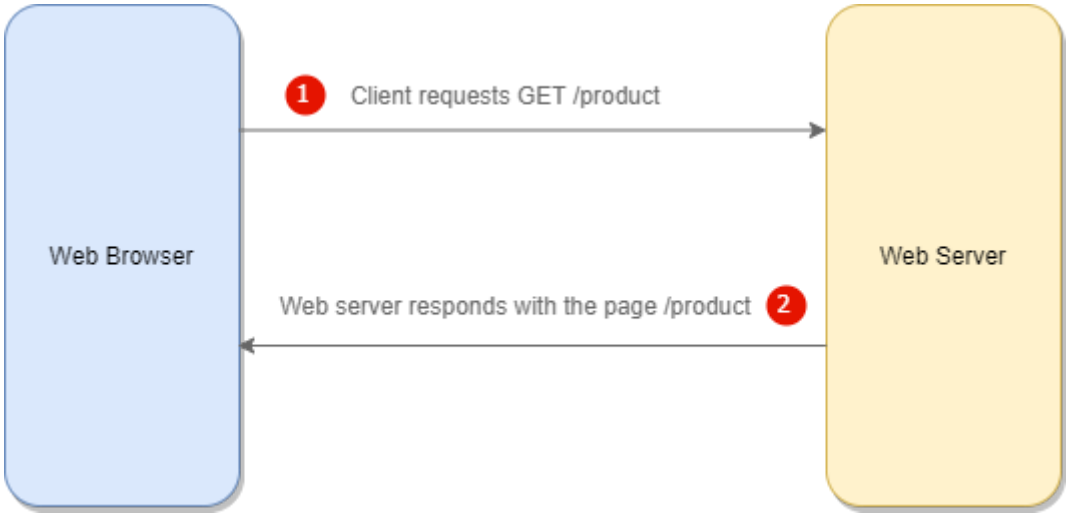
How it works:

- The code first sets a cookie named `user` with the value `John Doe` that expires in one hour using the `setcookie()` function.
- It then checks if the cookie is set using the `isset()` function. If the cookie is set, it greets the user by displaying "Hello, John Doe!". If the cookie is not set, it displays "Cookie is not set.".
- Finally, it deletes the cookie by setting its expiration time to one hour in the past using the `setcookie()` function and displays "Cookie has been deleted.".

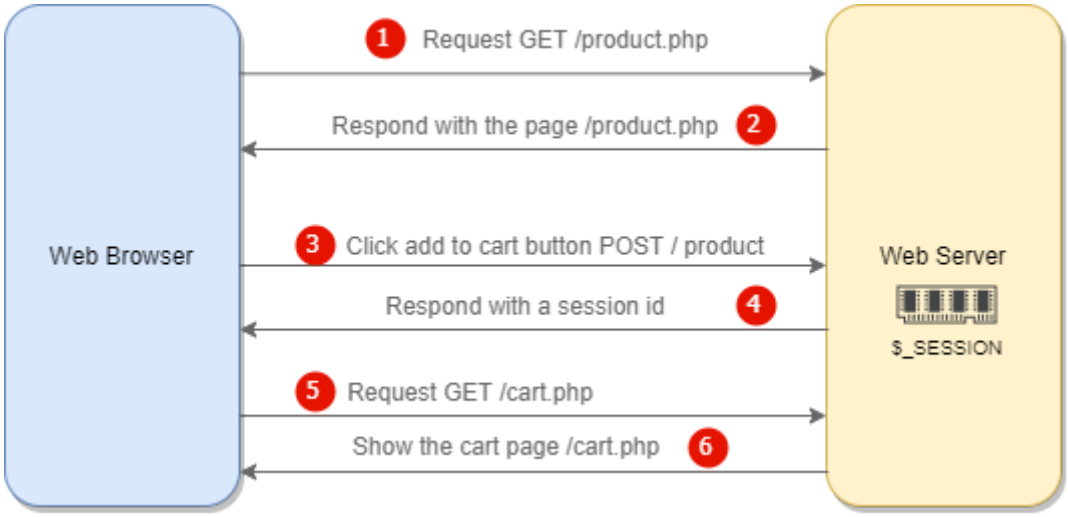
- Note that cookies are sent to the server with each HTTP request, so they can be used to maintain state and track user sessions across multiple requests.

PHP Sessions

- Suppose, you click the add to cart button on a `product.php` page and navigate to the `cart.php` page, the web server won't know that you have added the product to the cart and it renders as if nothing has been done. This is because HTTP is a stateless protocol that doesn't remember anything about the previous requests.



- To persist data across multiple requests, you can use sessions. Therefore, whenever you click the add to cart button, the web server will store the product on the server
- When you view the `cart.php` page, the web server gets the products from the session and displays them on the cart.php page:



How it works:

  - When you click the add to cart button on the `product.php` page, the web server creates a unique session ID for your session and stores the product information in a session variable on the server.
  - The web server sends the session ID to your web browser as a cookie.
  - When you navigate to the `cart.php` page, your web browser sends the session ID cookie back to the web server.
  - The web server retrieves the product information from the session variable using the session ID and displays it on the `cart.php` page. Technically speaking:
  - First, the web browser requests for the product.php page.

    - Second, the web server responds with the product.php page's content.
    - Third, you click the Add To Cart button on the product.php page. The page will send an HTTP request (either POST or GET) to the web server. The web server validates the product and generates a session id. It also creates a new text file on the server to store the information related to the selected product.
    - Fourth, the web server responds to the web browser with the `PHPSESSID` cookie in the response header. If the web browser allows cookies, it will save the `PHPSESSID` cookie, which stores the session id passed by the web server.
    - Fifth, in the subsequent request, for example, when you view the cart.php page, the web browser passes the `PHPSESSID` back to the web server. When the web server sees the `PHPSESSID` cookie, it will resume the session with the session id stored in it.
    - Finally, the web server returns the cart page with your selected products.
- Sessions allows you to store data in a web server associated with a session id. The session id is usually stored on the client-side as a cookie named `PHPSESSID`. The session id is a unique identifier that is used to identify the session on the server.
- A session typically lasts until the user closes the web browser or the session expires on the server. The expiration time can be configured on the server.

Creating a new session

- To create a new session in PHP, you use the `session_start()` function. This function initializes a new session or resumes an existing session based on the session ID passed by the client (usually via a cookie).
- It's syntax is as follows:

```
session_start ( void ) : bool
```

- When the `session_start()` runs at the first time, PHP generates a unique session id and passes it to the web browser in the form of a cookie named `PHPSESSID`.
- If a session already exists, PHP checks the `PHPSESSID` cookie sent by the browser, the session_start() function will resume the existing session instead of creating a new one.
- Since PHP sends the `PHPSESSID` cookie in the header of the HTTP response, you need to call the session_start() function before any statement that outputs the content to the web browser. (This is because PHP needs to send the session ID cookie in the HTTP response heade). Otherwise, you will get a warning message saying the header cannot be modified because it is already sent. This is a well-known error message in PHP

Where PHP stores session data

- PHP sessions are stored in a temporary file on the web server
- You can find the location of the temporary files using the directive `session.save_path` in the PHP configuration file
- The `ini_get()` function returns the value of the `session.save_path` directive

```
echo ini_get('session.save_path');
```

Or you can call the session_save_path() function:

```
echo session_save_path();
```

- Typically, the session data is stored in the `/tmp` folder of the web server e.g, `/xampp/tmp` .

Accessing session data

Unlike cookies, you can store any data in the session. To store data in the session, you set the key and value in the `$_SESSION` superglobal array.

- For example, in the index.php file, you store the user string and roles array in the session as follows:

```
<?php
    session_start();
    $_SESSION['user'] = 'admin';
    $_SESSION['roles'] = ['administrator', 'approver', 'editor'];
?>

<html>
<head>
    <title>PHP Session Demo</title>
</head>
<body>
    <a href="profile.php">Go to profile page</a>
</body>
</html>
```

How it works:

- The `session_start()` function initializes a new session or resumes an existing session.
- The `$_SESSION['user']` variable is set to the string value `'admin'`.
- The `$_SESSION['roles']` variable is set to an array containing three roles: `'administrator'`, `'approver'`, and `'editor'`. Therefore, the `$_SESSION` superglobal array contains an array and string as data-types in this case
- The HTML part of the code contains a link to the `profile.php` page.

- In the profile.php file, you can access the session data as follows:

```
<?php session_start() ?>
```

```php
<?php if (isset($_SESSION['user'])): ?>
    <p> Welcome <?php echo $_SESSION['user'] ?> </p>
<?php endif; ?>

<?php if(isset($_SESSION['roles'])): ?>
    <p> Current roles: </p>
        <ul>
            <?php foreach ($_SESSION['roles'] as $role): ?>
                <li><?php echo $role ?> </li>
            <?php endforeach; ?>
        </ul>
<?php endif; ?>
```

How it works:

- The `session_start()` function initializes a new session or resumes an existing session.
- The first `if` statement checks if the `$_SESSION['user']` variable is set using the `isset()` function. If it is set, it displays a welcome message with the value of the `$_SESSION['user']` variable.
- The second `if` statement checks if the `$_SESSION['roles']` variable is set. If it is set, it displays a list of roles by iterating over the array using a `foreach` loop and displaying each role in a list item (`<li>`).

Deleting the session data

- Whenever you close the web browser, PHP automatically deletes the session.
- Sometimes, you may want to explictly delete the session data before the browser is closed. You can do this using the `session_destroy()` function. This function destroys all data associated with the current session. However, it does not unset data in the `$_SESSION` array and cookie
- It's syntax is as follows:

```
session_destroy ( void ) : bool
```

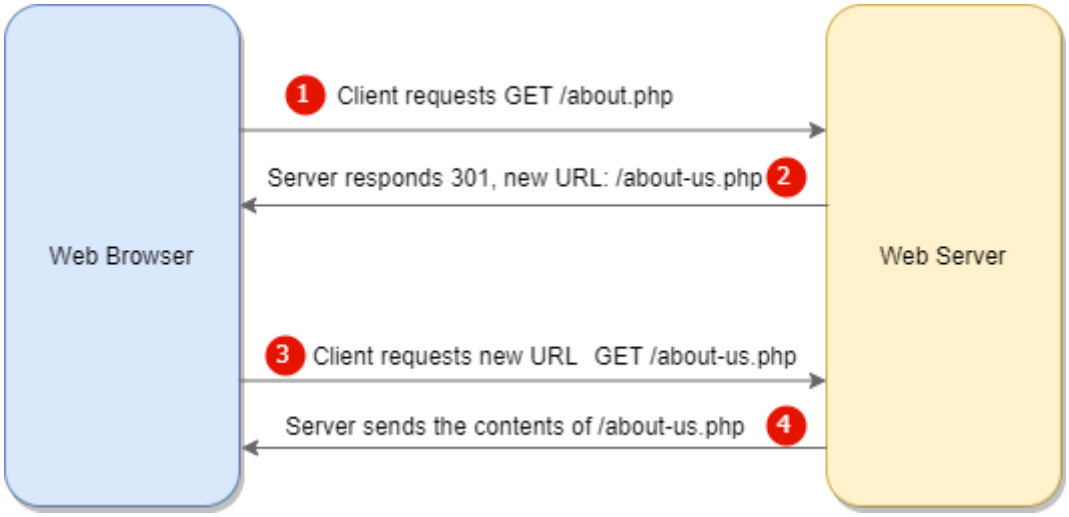
- To completely destroy the session data, you need to unset the variable in `$_SESSION` array and remove the `PHPSESSID` cookie like this:

```php
<?php
session_start();

// remove cookie
if(isset($_COOKIE[session_name()])){
    setcookie(session_name(),'',time() - 3600, '/');
}

// unset data in $_SESSION
$_SESSION[] = array();

// destroy the session
session_destroy();
```

- Notice that we used the `session_name()` function to get the cookie name instead of using the `PHPSESSID`. PHP allows you to work with multiple sessions with different names on the same script.
- The `session_name()` function returns the name of the current session. By default, it returns `PHPSESSID`, but you can change the session name using the `session_name()` function before calling `session_start()`.
- Use `sessions` to store sensitive data on the server-side instead of cookies. This is because cookies are stored on the client-side (browser) and can be easily manipulated by users. Sessions, on the other hand, store data on the server-side, making them more secure for sensitive information.
- Sessions are more suitable for storing larger amounts of data compared to cookies, which have size limitations.

PHP Redirection

- Redirection is the process of sending a user from one URL to another URL via the browser. In PHP, you can perform redirection using the `header()` function to send a `Location` header to the browser.
- Typically, we use redirection in the following scenarios:
  - Change the domain name of a website to another.
  - Replace the URL of a page with another.
  - Upgrade the HTTP to HTTPS.
  - Redirect users after form submission to prevent duplicate submissions.
- Suppose you navigate to the URL https://phptutorial.net/about-us.php, PHP will redirect you to the new URL https://phptutorial.net/about-us.php. The following shows how redirection works:



How it works:

  - When you enter the URL https://phptutorial.net/about-us.php in your web browser, the browser sends an HTTP request to the web server.
  - The web server processes (receives) the request and sends an HTTP response with a `Location` header that contains the new URL `https://phptutorial.net/about-us.php.`
  - The web browser receives the response and automatically navigates to the new URL by sending another HTTP request to the web server.
  - Finally, the web server responds with the content of the new page.

PHP `header()` function

- The `header()` function in PHP is used to send raw HTTP headers to the client (usually a web browser) before any output is sent. It allows you to modify the HTTP response headers, which can control various aspects of how the browser handles the response.
- It's syntax is as follows:

```
header ( string $header , bool $replace = true , int $response_code = 0 ) : void
```

where,

- `header` is the header string to be sent. For example, to redirect the browser to another URL, you can use the `Location` header like this: `Location: https://phptutorial.net/new-page.php`.
- `replace` indicates whether to replace a previous similar header or add a second header of the same type. By default, it is set to `true`, which means that it will replace any previous header with the same name.
- `response_code` allows you to specify the HTTP response code to be sent with the header. For example, you can set it to `301` for a permanent redirect or `302` for a temporary redirect. If not specified, the default response code is `200 OK`.
- Use the `exit()` construct after using the `header()` function for redirection to ensure that no further code is executed after the redirection. This prevents any unintended output or processing that could interfere with the redirection process.
- Since the `header()` function sends HTTP headers, you must call it before any output is sent to the browser. Otherwise, you will get a warning message saying the header cannot be modified because it is already sent. This is a well-known error message in PHP.

PHP Forms

- To create a web form, you basically use the `<form>` element as follows:

```
<form action="form.php" method="post"></form>
```

This `<form>` element has two important attributes:

- `action`: specifies the URL of the server-side script that will process the form data when the form is submitted. In this case, the form data will be sent to `form.php` for processing.
- `method`: specifies the HTTP method to be used when submitting the form. It can be either `GET` or `POST`. In this case, the form data will be sent using the `POST` method, which means that the data will be included in the body of the HTTP request. The form methods are case-insensitive (e.g., `POST`, `post`, `PoSt` are all valid). Also, if the method attribute isn't specified, the element uses `GET` by default

- Input elements used in forms are often called form fields
- These input elements has important attributes - `name`, `type`, `value`. The `name` attribute will be used to access the `value` in PHP

```
<input name="value" type="" />
```

HTTP POST method

- The POST method is used to send data to the server as part of the body of the HTTP request. It is commonly used when submitting form data that may include sensitive information or large amounts of data.
- After submitting the form, you can access the form data in PHP using the $_POST superglobal array. The $_POST is an associative array that contains key-value pairs, where the keys are the names of the form fields, and the values are the corresponding values entered by the user. For example, if your form has an input field with the name attribute set to username, you can access its value in PHP like this:

```
$_POST['username'];
```

- Before accessing the form data, you should check if the form has been submitted using the isset() function to avoid undefined index errors. For example if you want to check if the form field with the name username is set, you can do it like this:

```
if(isset($_POST['username'])) {
  // process the form data
}
```

- The following shows how to create a simple form (form.php) with an email input field

```
<?php
/*
if(isset($_POST['email'])) {
    echo "Thank you for subscribing to our newsletter";
} */

if($_SERVER['REQUEST_METHOD'] === 'POST') {
    $email = $_POST['email'];
    echo $email;
}

?>

<form action="form.php" method="POST">
    <label for="email"> Email </label>
    <input type="email" id="email" name="email">
    <button type="submit">Submit</button>
</form>
```

How it works:

- The PHP code at the top checks if the form has been submitted by verifying if the request method is `POST` using the `$_SERVER['REQUEST_METHOD']` variable.
- If the form has been submitted, it retrieves the value of the `email` input field from the `$_POST` superglobal array and assigns it to the `$email` variable.
- It then echoes the value of the `$email` variable, which displays the email address entered by the user.
- The HTML part of the code creates a form with an email input field and a submit button. When the user submits the form, the data is sent to `form.php` using the `POST` method for processing. By this, the `form.php` script runs again and processes the submitted form data.
- The form includes a `<label>` element for the email input field, which improves accessibility by associating the label with the input field using the `for` and `id` attributes.
- The `<input>` element has the `type` attribute set to `email`, which ensures that the input is validated as an email address by the browser.

PHP `GET` Method

- The `GET` method is used to send data to the server as part of the URL query string. It is commonly used when submitting form data that is not sensitive and can be included in the URL.
- After submitting the form, you can access the form data in PHP using the `$_GET` superglobal array. The `$_GET` is an associative array that contains key-value pairs, where the keys are the names of the form fields, and the values are the corresponding values entered by the user. For example, if your form has an input field with the `name` attribute set to `username`, you can access its value in PHP like this:

```
$_GET['username'];
```

- The follwing creates a search form with an input field

```
<?php
$languages = [
    'python' => 'Python is a high-level, general-purpose programming language
known for its clear, readable syntax that emphasizes the use of significant
indentation rather than curly brackets',
    'php' => 'PHP (a recursive acronym for PHP: Hypertext Preprocessor) is a
widely-used, open-source, server-side scripting language designed primarily for
web development. It is embedded within HTML to create dynamic and interactive web
pages. ',
    'javascript' => 'JavaScript is a versatile, high-level programming language
that is a core technology of the World Wide Web, alongside HTML and CSS'
];

if($_SERVER['REQUEST_METHOD'] === 'GET') {
    $search_value = $_GET['search'];

    if(!empty($languages['python'])) {
        echo "<p>The result of the search for <b>$search_value</b></p>
              <p>$languages[$search_value]</p>";
    };
}
?>
```

```
<form action="form.php">
    <label for='search'>Search</label>
    <input type='text' id='search' name='search'>
    <button type='submit'>Search</button>
</form>
```

How it works:

- The PHP code at the top defines an associative array `$languages` that contains descriptions of three programming languages: Python, PHP, and JavaScript.
- It then checks if the request method is `GET` using the `$_SERVER['REQUEST_METHOD']` variable.
- If the request method is `GET`, it retrieves the value of the `search` input field from the `$_GET` superglobal array and assigns it to the `$search_value` variable.
- It checks if the `$languages` array contains a description for the searched language using the `!empty()` function.
- If a description exists, it echoes a message displaying the search term and the corresponding description from the `$languages` array.
- The HTML part of the code creates a form with a text input field for searching and a submit button. When the user submits the form, the data is sent to `form.php` using the `GET` method for processing. By this, the `form.php` script runs again and processes the submitted form data.
- The form includes a `<label>` element for the search input field, which improves accessibility by associating the label with the input field using the `for` and `id` attributes.

- If the form has multiple input elements, the web browser will append the input fields to the URL in the following format `/search.php?name1=value1&name2=value2&name3=value3`

HTTP `GET` vs `POST` Methods

- Generally, use the `GET` method when you want to retrieve data from the server without making any changes to the server's state. For example a search form that allows users to search for information should use the `GET` method
- Use the `POST` method when you want to send data to the server that will result in a change in the server's state, such as creating or updating a resource. For example, a form that allows users to subscribe to a newsletter should use the `POST` method

Escaping the output

- In the examples above, both forms display the form data directly. However, the page is not secure if malicious users intentionally inject JavaScript code into the data.
- For example, if the following JavaScript code is entered in the term field and the form is submitted

```
<script>
  alert("Hello");
</script>
```

- Imagine that the script doesn't just show an alert but redirect users to a malicious page that mimic the legitimate page, users may enter credential information like username/password and lose it. This type of attack is called cross-site scripting (XSS) attack.
- To prevent XSS attacks, before displaying user input on a webpage, you should always escape the data using the htmlspecialchars() function. This function converts special characters to HTML entities, preventing the browser from interpreting them as HTML or JavaScript code. It's syntax is as follows:

```
htmlspecialchars ( string $string , int $flags = ENT_QUOTES | ENT_SUBSTITUTE |
ENT_HTML401 , ?string $encoding = null , bool $double_encode = true ) : string
```

where,

- `string` is the input string to be escaped.
- `flags` is a bitmask of one or more of the following flags, which specify how to handle quotes and other special characters. The default value is `ENT_QUOTES | ENT_SUBSTITUTE | ENT_HTML401`.
- `encoding` is the character encoding of the input string. If not specified, it defaults to the value of the `default_charset` configuration option.
- `double_encode` indicates whether to encode existing HTML entities in the input string. If set to `true`, existing entities will be encoded again. The default value is `true`.

- For example, the following form shows how to use the `htmlspecialchars` function to display the search term on the page

```
<?php

if($_SERVER['REQUEST_METHOD'] === 'GET') {
    if(isset($_GET['term'])) {
        // get the search term from the URL
        $term = $_GET['term'];

        if($term) {
            $clean_term = htmlspecialchars($term, ENT_QUOTES, 'UTF-8');
            // perform search and show the result
            echo "<p>The result of the search for <b>$clean_term</b>:</p>";
        }
    }
}
?>


<form action="search.php" method="get">
    <div>
        <label for="term">Search:</label>
        <input type="search" name="term" placeholder="Enter search term">
        <button type="submit">Search</button>
    </div>
</form>
```

By this, the `HTML` code will be considered as a normal text instead of being executed as a script. How it works:

- The PHP code at the top checks if the request method is `GET` using the `$_SERVER['REQUEST_METHOD']` variable.
- If the request method is `GET`, it checks if the `term` input field is set in the `$_GET` superglobal array.
- If the `term` input field is set, it retrieves its value and assigns it to the `$term` variable.
- It then checks if the `$term` variable is not empty.
- If the `$term` variable is not empty, it uses the `htmlspecialchars()` function to escape any special characters in the `$term` variable and assigns the result to the `$clean_term` variable.
- Finally, it echoes a message displaying the escaped search term.
- The HTML part of the code creates a form with a search input field and a submit button. When the user submits the form, the data is sent to `search.php` using the `GET` method for processing. By this, the `search.php` script runs again and processes the submitted form data.
- The form includes a `<label>` element for the search input field, which improves accessibility by associating the label with the input field using the `for` and `id` attributes.

PHP self-processing form

- Suppose you have a form that submits data to the same page for processing. This is called a self-processing form.
- To create a self-processing form, you set the `action` attribute of the `<form>` element to the current page using the `$_SERVER['PHP_SELF']` variable. This variable contains the filename of the currently executing script relative to the document root. By this, even though the file name is later changed, there will be no error
- The following shows how to create a self-processing form in PHP:

```
<?php

if($_SERVER['REQUEST_METHOD'] === 'POST') {
    $email = $_POST['email'];
    echo htmlspecialchars($email);
}
?>


<form action="<?php echo htmlspecialchars($_SERVER['PHP_SELF']); ?>"
method="post">
    <div>
        <label for="email">Email:</label>
        <input type="email" name="email">
    </div>
</form>
```

How it works:

- The PHP code at the top checks if the request method is `POST` using the `$_SERVER['REQUEST_METHOD']` variable.

- If the request method is `POST`, it retrieves the value of the `email` input field from the `$_POST` superglobal array and assigns it to the `$email` variable.
- It then echoes the escaped value of the `$email` variable using the `htmlspecialchars()` function to prevent XSS attacks.
- The HTML part of the code creates a form with an email input field and a submit button. The `action` attribute of the `<form>` element is set to the current page using the `$_SERVER['PHP_SELF']` variable. Note that since the variable is a php code, it must be within the `<?php ?>` tags. To prevent XSS attacks, the `htmlspecialchars()` function is used to escape the value of `$_SERVER['PHP_SELF']`.
- When the user then submits the form, the data is sent to the same page for processing. By this, the script runs again and processes the submitted form data.

Organizing Code

- To create a more organized code, you can separate the PHP code and HTML code into different sections as follows:

```
.
├── css
│   └── style.css
├── inc
│   ├── header.php
│   ├── footer.php
│   ├── get.php
│   ├── post.php
│   └── .htaccess
└── index.php
```

- The `index.php` file in the root directory will include the `header.php` and `footer.php` files for the HTML header and footer sections respectively.
- If the request method is GET, the index.php file loads the form in the get.php file. Otherwise, it loads the code from the post.php file for processing the POST request.

PHP `filter_has_var()` function

- The `filter_has_var()` function in PHP is used to check if a variable of a specified input type exists in the input data. It is commonly used to validate user input from forms or query strings.
- It's syntax is as follows:

```
filter_has_var ( int $type , string $variable_name ) : bool
```

where,

- `type` is the type of input data to check. It can be one of the following constants:
  - `INPUT_GET`: to check for variables in the `$_GET` superglobal array.

    - `INPUT_POST`: to check for variables in the `$_POST` superglobal array.
    - `INPUT_COOKIE`: to check for variables in the `$_COOKIE` superglobal array.
    - `INPUT_SERVER`: to check for variables in the `$_SERVER` superglobal array.
    - `INPUT_ENV`: to check for variables in the `$_ENV` superglobal array.
- `variable_name` is the name of the variable to check for. This is usually the name attribute of the form field.
- The function returns `true` if the variable exists in the specified input type, and `false` otherwise.
- Typically, you use the `filter_has_var()` function alongside `filter_input()` and `filter_var()` to validate input data.

Checking a GET variable

- The following shows how to use the `filter_has_var()` function to check if a GET variable named `search` exists:

```
if(filter_has_var(INPUT_GET, 'search')) {
    echo "The search variable exists in the GET data.";
} else {
    echo "The search variable does not exist in the GET data.";
}
```

How it works:

- The `filter_has_var()` function checks if the `search` variable exists in the `$_GET` superglobal array. The variable will only exist if the form is submitted using the GET method with a field named `search`.
- If the variable exists, it echoes a message indicating that the variable exists in the GET data. Otherwise, it echoes a message indicating that the variable does not exist.

Checking a POST variable

- The following shows how to use the `filter_has_var()` function to check if a POST variable named `email` exists:

```
if(filter_has_var(INPUT_POST, 'email')) {
    echo "The email variable exists in the POST data.";
} else {
    echo "The email variable does not exist in the POST data.";
}
```

- The `filter_has_var()` function checks variables that are define in the request body when the form is submitted using the POST method not the variables in the URL query string. By this, the following code returns false:

```
<?php

$_POST['email'] = 'example@phptutorial.net';
```

```
if(filter_has_var(INPUT_POST, 'email')) { // return false
    // ...
}
```

PHP `filter_var()` function

- The `filter_var()` function in PHP is used to filter and validate a variable using a specified filter. It is commonly used to sanitize user input and ensure that the data meets certain criteria before processing it.
- It's syntax is as follows:

```
filter_var ( mixed $variable , int $filter = FILTER_DEFAULT , array|int $options =
0 ) : mixed
```

where,

- `variable` is the variable to be filtered. It can be of any data type.
- `filter` is the filter to be applied to the variable. It can be one of the predefined filter constants in PHP, such as `FILTER_SANITIZE_STRING`, `FILTER_VALIDATE_EMAIL`, `FILTER_VALIDATE_INT`, etc. The default filter is `FILTER_DEFAULT`, which does not perform any filtering.
- `$options` is an associative array of filter options or a list of flags separated by the pipe character (`|`). The options vary depending on the filter being used.
- The function returns the filtered variable if the filtering is successful, or `false` if the filtering fails.

Validating Data

- The following shows how to use the `filter_var()` function to validate an email address:

```
$email = "example@example.com";
if(filter_var($email, FILTER_VALIDATE_EMAIL)) {
    echo "The email address is valid.";
} else {
    echo "The email address is not valid.";
}
```

How it works:

- The code defines a variable `$email` with the value `"example@example.com"`.
- It then uses the `filter_var()` function to validate the email address by passing the `$email` variable and the `FILTER_VALIDATE_EMAIL` filter as arguments.
- If the email address is valid, it echoes a message indicating that the email address is valid. Otherwise, it echoes a message indicating that the email address is not valid.

- The following shows how to use the `filter_var()` function to validate an integer:

```
if(isset($_GET['age'])) {
    if(filter_var($_GET['age'], FILTER_VALIDATE_INT, [
    'options' => [
        'min_range' => 1,
        'max_range' => 45
        ]
    ])){
        echo 'Valid age';
    } else {
        echo 'Invalid age';
    }
}
```

How it works:

- The code first checks if the `age` variable is set in the `$_GET` superglobal array using the `isset()` function.
- If the `age` variable is set, it uses the `filter_var()` function to validate the age by passing the `$_GET['age']` variable, the `FILTER_VALIDATE_INT` filter, and an array of options as arguments.
- The options specify a minimum range of `1` and a maximum range of `45` for the age. This variables `min_range` and `max_range` aren't reserved words in PHP but are specific to the `FILTER_VALIDATE_INT` filter.
- If the age is valid (i.e., it is an integer within the specified range), it echoes a message indicating that the age is valid. Otherwise, it echoes a message indicating that the age is invalid.

Sanitizing Data

- The following shows how to use the `filter_var()` function to sanitize a string by removing HTML tags:

```
$string = "<h1>Hello, World!</h1>";
$sanitized_string = filter_var($string, FILTER_SANITIZE_STRING);
echo $sanitized_string; // Output: Hello, World!
```

- This example uses the `filter_var()` function to sanitize numeric by removing strings from it:

```
$age = '120abg';
$sanitized_age = filter_var($age, FILTER_SANITIZE_NUMBER_INT);

echo $santized_age === false ? echo 'Invalid age' : echo $sanitized_age; //
Output: 120
```

How it works:

- The code defines a variable `$age` with the value `'120abg'`, which contains both numeric and non-numeric characters.
- It then uses the `filter_var()` function to sanitize the age by passing the `$age` variable and the `FILTER_SANITIZE_NUMBER_INT` filter as arguments.
- The `FILTER_SANITIZE_NUMBER_INT` filter removes all non-numeric characters from the string, leaving only the numeric characters.
- Finally, it echoes the sanitized age, which is `120`. If the result is `false`, it echoes 'Invalid age'.

PHP `filter_input()` function

- The `filter_input()` function in PHP is used to retrieve a specific input variable from a specified input type and optionally filter it using a specified or more than one filter. It is commonly used to validate and sanitize user input from forms or query strings.
- A good rule of thumb is that you should never trust external data and always:
  - Sanitize and validate data before storing it in the database.
  - Escape data before displaying it on a web page.
- *Sanitization* disables potential malicious code from data before processing it.
- *Validation* ensures the data is in the correct format regarding data type, range, and value.

- The syntax of the `filter_input()` function is as follows:

```
filter_input ( int $type , string $variable_name , int $filter = FILTER_DEFAULT ,
array|int $options = 0 ) : mixed
```

where,

- `type` is the type of input data to retrieve. It can be one of the following constants:
  - `INPUT_GET`: to retrieve variables from the `$_GET` superglobal array.
  - `INPUT_POST`: to retrieve variables from the `$_POST` superglobal array.
  - `INPUT_COOKIE`: to retrieve variables from the `$_COOKIE` superglobal array.
  - `INPUT_SERVER`: to retrieve variables from the `$_SERVER` superglobal array.
  - `INPUT_ENV`: to retrieve variables from the `$_ENV` superglobal array.
- `variable_name` is the name of the variable to retrieve. This is usually the name attribute of the form field.
- `filter` is the filter to be applied to the variable. It can be one of the predefined filter constants in PHP, such as `FILTER_SANITIZE_STRING`, `FILTER_VALIDATE_EMAIL`, `FILTER_VALIDATE_INT`, etc. The default filter is `FILTER_DEFAULT`, which does not perform any filtering.
- `$options` is an associative array of filter options or a list of flags separated by the pipe character (`|`). The options vary depending on the filter being used.
- The function returns the filtered variable if the filtering is successful, or `false` if the filtering fails. If `$var_name` does not exist, it returns `null`.

- For example, the following shows how to use the `filter_input()` function to validate an email address from a POST request:

```
$email = filter_input(INPUT_POST, 'email', FILTER_VALIDATE_EMAIL);
if($email) {
    echo "The email address is valid.";
} else {
    echo "The email address is not valid.";
}
```

- Consider this example

```
<?php

$term_html = filter_input(INPUT_GET, 'term', FILTER_SANITIZE_SPECIAL_CHARS);
$term_url = filter_input(INPUT_GET, 'term', FILTER_SANITIZE_ENCODED);

?>
<form action="search.php" method="get">
    <label for="term"> Search </label>
    <input type="search" name="term" id="term" value="<?php echo $term_html ?>">
    <input type="submit" value="Search">
</form>

<?php

if (null !== $term_html) {
    echo "The search result for <mark> $term_html </mark>.";
}
```

How it works:

- The code first uses the `filter_input()` function to retrieve and sanitize the `term` variable from the `$_GET` superglobal array using two different filters: `FILTER_SANITIZE_SPECIAL_CHARS` and `FILTER_SANITIZE_ENCODED`. The sanitized values are assigned to the `$term_html` and `$term_url` variables, respectively. The `FILTER_SANITIZE_SPECIAL_CHARS` filter returns a value for showing on the search field and the `FILTER_SANITIZE_ENCODED` filter returns a value for displaying on the page.
- The HTML part of the code creates a form with a search input field and a submit button. The `value` attribute of the input field is set to the sanitized value of `$term_html`, which ensures that any special characters in the search term are properly escaped when displayed in the input field.
- When the user submits the form, the data is sent to `search.php` using the `GET` method for processing. By this, the `search.php` script runs again and processes the submitted form data.
- Finally, the code checks if the `$term_html` variable is not `null`. If it is not `null`, it echoes a message displaying the sanitized search term.
- `type` is the type of input data to check. It can be one of the following constants:
    - `INPUT_GET`: to check for variables in the `$_GET` superglobal array.
    - `INPUT_POST`: to check for variables in the `$_POST` superglobal array.
- Use the `filter_input()` function when you need to validate and sanitize data coming directly from user and `filter_var()` when the input is already in a variable

- `filter_input()` - Validate / sanitize form input
- `filter_var()` - Validate / sanitize a variable

PHP Form Validation

- To validate data in PHP, you use filters with
  - `filter_has_var()` - check if a variable exist in the `GET` and `POST` requests
  - `filter_input()` - validate data

Validating Emails

- The following code shows how emails are validated

```
<form action="<?= htmlspecialchars($_SERVER['PHP_SELF']) ?>" method="post">
    <div>
        <label for="email">Email:</label>
        <input type="text" name="email">
        <button type="submit">Submit</button>
    </div>
</form>

<?php

if($_SERVER['REQUEST_METHOD'] === 'POST') {
    // Check if the email field is set and not empty
    if(filter_has_var(INPUT_POST, 'email')) {
        // validate email
        $email = filter_input(INPUT_POST, 'email', FILTER_VALIDATE_EMAIL);
        if($email !== false) {
            echo "Email is valid: " . htmlspecialchars($email);
        } else {
            echo "Invalid email format." . $_POST['email'];
        }
    }
}
```

- How it works:
-

Validating Integers

- The following code demonstrates how integers can be validated

```
<?php

if($_SERVER['REQUEST_METHOD'] === 'POST') {
    // Check if the age field is set and not empty
    if(filter_has_var(INPUT_POST, 'age')) {
        // validate age between 0 and 150
```

```
        $age = filter_input(INPUT_POST, 'age', FILTER_VALIDATE_INT, [
            'options' => [
                'min_range' => 0,
                'max_range' => 150
            ]
        ]);

        if($age !== false) {
            echo "Age is valid: " . htmlspecialchars($age);
        } else {
            echo "Age is not valid:" . $_POST['age'];
        }
    }
}
?>

<form action="<?= htmlspecialchars($_SERVER['PHP_SELF']) ?>" method="post">
    <div>
        <label for="age">Age:</label>
        <input type="text" name="age" placeholder="Enter your age">
        <button type="submit">Submit</button>
    </div>
</form>
```

How it works:

Validating Floats

```
<?php

if($_SERVER['REQUEST_METHOD'] === 'POST') {
    // Check if the weight field is set and not empty
    if(filter_has_var(INPUT_POST, 'weight')) {
        // validate weight between 0 and 150
        $weight = filter_input(INPUT_POST, 'weight', FILTER_VALIDATE_FLOAT, [
            'options' => [
                'min_range' => 0,
                'max_range' => 300

            ]
        ]);

        if($weight !== false) {
            echo "Weight is valid: " . htmlspecialchars($weight);
        } else {
            echo "Weight is not valid:" . $_POST['weight'];
        }
    }
}
?>
```

```
<form action="<?= htmlspecialchars($_SERVER['PHP_SELF']) ?>" method="post">
    <div>
        <label for="weight">Weight:</label>
        <input type="text" name="weight" placeholder="Enter your weight in lbs">
        <button type="submit">Submit</button>
    </div>
</form>
```

How it works:

- The PHP code at the top checks if the request method is `POST` using the `$_SERVER['REQUEST_METHOD']` variable.
- If the request method is `POST`, it checks if the `weight` variable is set in the `$_POST` superglobal array using the `filter_has_var()` function.
- If the `weight` variable is set, it uses the `filter_input()` function to validate the weight by passing the `INPUT_POST` type, the `weight` variable name, the `FILTER_VALIDATE_FLOAT` filter, and an array of options as arguments.
- The options specify a minimum range of `0` and a maximum range of `300` for the weight.
- If the weight is valid (i.e., it is a float within the specified range), it echoes a message indicating that the weight is valid. Otherwise, it echoes a message indicating that the weight is not valid.
- The HTML part of the code creates a form with an email input field and a submit button. The `action` attribute of the `<form>` element is set to the current page using the `$_SERVER['PHP_SELF']` variable. To prevent XSS attacks, the `htmlspecialchars()` function is used to escape the value of `$_SERVER['PHP_SELF']`.
- When the user submits the form, the data is sent to the same page for processing.
- The form includes a `<label>` element for the email input field, which improves accessibility by associating the label with the input field using the `for` and `id` attributes.
- The PHP code at the top checks if the request method is `POST` using the `$_SERVER['REQUEST_METHOD']` variable.
- If the request method is `POST`, it checks if the `email` variable is set in the `$_POST` superglobal array using the `filter_has_var()` function.
- If the `email` variable is set, it uses the `filter_input()` function to validate the email address by passing the `INPUT_POST` type, the `email` variable name, and the `FILTER_VALIDATE_EMAIL` filter as arguments.
- If the email address is valid, it echoes a message indicating that the email address is valid. Otherwise, it echoes a message indicating that the email address is not valid.

PHP Date and Time

PHP `time()` function

- The `time()` function in PHP is used to get the **current Unix timestamp**, which is the number of seconds that have elapsed since January 1, 1970 (also known as the Unix epoch). The Unix timestamp is a widely used format for representing dates and times in computing.
- The syntax of the `time()` function is as follows:

```
time ( ) : int
```

- The function does not take any parameters and returns the current Unix timestamp as an integer value.
- The following code shows how to use the `time()` function to get the current Unix timestamp:

```
$timestamp = time();
echo "Current Unix timestamp: " . $timestamp;
```

- The code calls the `time()` function and assigns the returned Unix timestamp to the `$timestamp` variable. It then echoes the current Unix timestamp. This timestamp is neccesarily relative to UTC time zone and not the server time zone.

PHP `date()` function

- The `date()` function in PHP is used to format a date and time based on a specified format string. It allows you to display dates and times in various formats according to your requirements.
- The syntax of the `date()` function is as follows:

```
date ( string $format , ?int $timestamp = null ) : string
```

where,

- `format` is a string that specifies the desired format for the date and time. It can include various format characters that represent different components of the date and time, such as year, month, day, hour, minute, second, etc. It can be used as follows:
    - `Y`: A four-digit representation of the year (e.g., 2024)
    - `m`: A two-digit representation of the month (01 to 12)
    - `d`: A two-digit representation of the day of the month (01 to 31)
    - `H`: A two-digit representation of the hour in 24-hour format (00 to 23)
    - `i`: A two-digit representation of the minutes (00 to 59)
    - `s`: A two-digit representation of the seconds (00 to 59)

  For example, the format string `"Y-m-d H:i:s"` would produce a date and time in the format "2024-06-15 14:30:45". A complete list of format characters can be found in the PHP documentation.
- `timestamp` is an optional parameter that specifies the Unix timestamp to be formatted. If not provided, the current date and time will be used.
- The function returns a formatted date and time string based on the specified format.
- For example, the following code shows how to use the `date()` function to format the current date and time:

```
echo date("Y-m-d H:i:s");
```

- This example shows the date and time for boxing-day

```
$timestamp = time() + (365 - 22) * 24 * 60 * 60;

echo date('y / M / jS H : iA', $timestamp);
```

- The code calls the `date()` function with the format string `"Y-m-d H:i:s"` to format the current date and time. It echoes the formatted date and time string.

PHP `date_default_timezone_set()` function

- The `date_default_timezone_set()` function in PHP is used to set the default timezone for all date and time functions in a script. This function allows you to specify the timezone that should be used when working with dates and times, ensuring that the output is consistent with the desired timezone.
- The syntax of the `date_default_timezone_set()` function is as follows:

```
date_default_timezone_set ( string $timezone_identifier ) : bool
```

where,

- `timezone_identifier` is a string that specifies the timezone to be set as the default. It should be a valid timezone identifier recognized by PHP, such as "America/New_York", "Europe/London", "Asia/Tokyo", etc. A complete list of supported timezone identifiers can be found in the PHP documentation.
- The function returns `true` if the timezone was successfully set, or `false` if the timezone identifier is invalid.
- For example, the following code shows how to use the `date_default_timezone_set()` function to set the default timezone to "America/New_York":

```
date_default_timezone_set("America/New_York");
echo date("Y-m-d H:i:s");
```

- The code calls the `date_default_timezone_set()` function with the timezone identifier "America/New_York" to set the default timezone. It then calls the `date()` function to format and echo the current date and time in the specified timezone.
- Consider another example:

```
$timestamp = time() - (22 * 24 * 60 * 60);
```

```
date_default_timezone_set('America/Toronto');
echo date('jS / M / o H : iA', $timestamp);
```

This will print the date of new year eve with respect to January 22, 2026

- Know that this function should be called before any date and time functions are used in the script to ensure that the correct timezone is applied.

PHP `date_default_timezone_get()` function

- The `date_default_timezone_get()` function in PHP is used to retrieve the current default timezone set for all date and time functions in a script. This function allows you to check which timezone is currently being used when working with dates and times.
- The syntax of the `date_default_timezone_get()` function is as follows:

```
date_default_timezone_get ( ) : string
```

Practically, it is advisable to use the `UTC` timezone in your application and convert it to the desired timezone when displaying dates and times to users. This approach helps avoid issues related to daylight saving time changes and ensures consistency across different timezones.

PHP `mktime()` function

- The `mktime()` function in PHP is used to create a Unix timestamp from a specified date and time. It allows you to generate a timestamp based on individual components such as year, month, day, hour, minute, and second.
- The syntax of the `mktime()` function is as follows:

```
mktime ( int $hour , int $minute , int $second , int $month , int $day , int $year
, int $is_dst = -1 ) : int|false
```

where,

- `hour` is an integer representing the hour of the day (0 to 23).
- `minute` is an integer representing the minutes (0 to 59).
- `second` is an integer representing the seconds (0 to 59).
- `month` is an integer representing the month of the year (1 to 12).
- `day` is an integer representing the day of the month (1 to 31).
- `year` is an integer representing the year (e.g., 2024).
- `is_dst` is an optional parameter that indicates whether daylight saving time (DST) is in effect. It can take the following values:
  - `1`: DST is in effect.
  - `0`: DST is not in effect.
  - `-1`: PHP will attempt to determine whether DST is in effect based on the provided date and time.

- The function returns the Unix timestamp corresponding to the specified date and time, or `false` if the provided date and time is invalid.
- For example, the following code shows how to use the `mktime()` function to create a Unix timestamp for a specific date and time:

```
$timestamp = mktime(14, 30, 0, 6, 15, 2024);
echo "Unix timestamp for 2024-06-15 14:30:00: " . $timestamp;
```

- The code calls the `mktime()` function with the specified hour, minute, second, month, day, and year to create a Unix timestamp for June 15, 2024, at 14:30:00. It then echoes the generated Unix timestamp.

PHP `strtotime()` function

- The `strtotime()` function in PHP is used to convert a human-readable date and time string into a Unix timestamp. It allows you to parse various date and time formats and generate a corresponding timestamp.
- The syntax of the `strtotime()` function is as follows:

```
strtotime ( string $datetime , ?int $base_timestamp = null ) : int|false
```

where,

- `datetime` is a string that represents the date and time to be converted. It can be in various formats, such as "YYYY-MM-DD", "DD/MM/YYYY", "Month Day, Year", "next Monday", "last Friday", etc.
- `base_timestamp` is an optional parameter that specifies a base Unix timestamp to use as a reference point for relative date and time calculations. If not provided, the current time will be used as the base.
- The function returns the Unix timestamp corresponding to the specified date and time string, or `false` if the provided string cannot be parsed.
- For example, the following code shows how to use the `strtotime()` function to convert a human-readable date and time string into a Unix timestamp:

```
$timestamp = strtotime("2024-06-15 14:30:00");
echo "Unix timestamp for 2024-06-15 14:30:00: " . $timestamp;
```

- The code calls the `strtotime()` function with the date and time string "2024-06-15 14:30:00" to convert it into a Unix timestamp. It then echoes the generated Unix timestamp.
- Consider another example that shows how to use relative date formats with the `strtotime()` function:

```
$timestamp = strtotime("next Monday");
echo "Unix timestamp for next Monday: " . $timestamp;
```

- The code calls the `strtotime()` function with the relative date string "next Monday" to convert it into a Unix timestamp. It then echoes the generated Unix timestamp for the next Monday from the current date.
- The `strtotime()` function is particularly useful for parsing user input or working with dynamic date and time values in PHP applications.

PHP `getdate()` function

- The `getdate()` function in PHP is used to retrieve an associative array containing information about a specified Unix timestamp or the current date and time if no timestamp is provided. The array includes various components of the date and time, such as year, month, day, hour, minute, second, etc.
- The syntax of the `getdate()` function is as follows:

```
getdate ( ?int $timestamp = null ) : array
```

where,

- `timestamp` is an optional parameter that specifies the Unix timestamp for which to retrieve the date and time information. If not provided, the current date and time will be used.
- The function returns an associative array containing the following keys:
  - `seconds`: The seconds (0 to 59).
  - `minutes`: The minutes (0 to 59).
  - `hours`: The hours (0 to 23).
  - `mday`: The day of the month (1 to 31).
  - `wday`: The day of the week (0 for Sunday, 6 for Saturday).
  - `mon`: The month of the year (1 to 12).
  - `year`: The year (e.g., 2024).
  - `yday`: The day of the year (0 to 365).
  - `weekday`: The full name of the day of the week (e.g., "Monday").
  - `month`: The full name of the month (e.g., "January").
  - `0`: The Unix timestamp.

- For example, the following code shows how to use the `getdate()` function to retrieve date and time information for the current date and time:

```
$date_info = getdate();
print_r($date_info);
```

- The code calls the `getdate()` function without any parameters to retrieve date and time information for the current date and time. It then uses the `print_r()` function to display the contents of the returned associative array.

PHP `date_parse()` function

- The `date_parse()` function in PHP is used to parse a date and time string into its individual components, such as year, month, day, hour, minute, second, etc. It returns an associative array

containing the parsed components of the date and time.
- The syntax of the `date_parse()` function is as follows:

```
date_parse ( string $date ) : array
```

where,

- `date` is a string that represents the date and time to be parsed. It can be in various formats, such as "YYYY-MM-DD", "DD/MM/YYYY", "Month Day, Year", etc.
- The function returns an associative array containing the following keys:
  - `year`: The year (e.g., 2024).
  - `month`: The month (1 to 12).
  - `day`: The day of the month (1 to 31).
  - `hour`: The hour (0 to 23).
  - `minute`: The minutes (0 to 59).
  - `second`: The seconds (0 to 59).
  - `fraction`: The fraction of a second.
  - `warning_count`: The number of warnings encountered during parsing.
  - `warnings`: An array of warning messages.
  - `error_count`: The number of errors encountered during parsing.
  - `errors`: An array of error messages.

`date_parse()` vs `getdate()`

- The `date_parse()` function is used to parse a date and time string into its individual components, while the `getdate()` function retrieves date and time information for a specified Unix timestamp or the current date and time.
- The `date_parse()` function takes a date and time string as input and returns an associative array containing the parsed components, whereas the `getdate()` function takes an optional Unix timestamp as input and returns an associative array containing date and time information for that timestamp.
- The `date_parse()` function is useful when you have a date and time string that you want to break down into its components, while the `getdate()` function is useful when you want to retrieve date and time information for a specific timestamp or the current date and time.

Working with PHP Configuration file - `PHP.INI`

- The location of this file is dependent on server, OS and so on.
- Suppose you are working with XAMPP, your PHP configuration file can be located thus

```

```

Or through the XAMPP control panel by clicking the `Config` button and selecting `PHP (php.ini)` from the dropdown menu. !['Step 1'](PHP Config.png)

- You can also create a PHP file with the following code to find the location of your `php.ini` file:

```
<?php
phpinfo();
?>
```

- In the `php.ini` file, semi-colons indicates comments and text enclosed in `[]` are ignored (they are used as heading). You can search for specific settings using the search functionality of your text editor.
- Also, in that file you can find individual settings (or directives). You can check out the PHP documentation for a complete list of PHP directives. The `Name` column is the name of the direcitve, the `Default` column is the default value of the directive, and the `Changeable` column indicates whether the directive can be changed at runtime using functions like `ini_set()` or `ini_get()`. If you have `PHP_INI_PERDIR` or `PHP_INI_SYSTEM` in the `Changeable` column, you cannot change the directive.
- The `ini_set()` function is used to set the value of a configuration option at runtime. The syntax of the `ini_set()` function is as follows:

```
ini_set ( string $option , string $value ) : string|false
```

where,

- `option` is the name of the configuration option to be set.
- `value` is the new value to be assigned to the configuration option.

Some PHP Directives

- `error_reporting`: This directive controls which types of errors are reported by PHP. It can be set to different levels, such as `E_ALL`, `E_ERROR`, `E_WARNING`, etc. For example, to report all errors (which is the default), you can set it as follows:

```
error_reporting = E_ALL
```

Or in PHP code:

```
ini_set('error_reporting', E_ALL);
```

- Know that it is advisable to stick to `E_ALL` during development and change it to a less verbose level in production. But still, it is advisable to log errors in production (i.e setting it to `E_ALL`).

- `display_errors`: This directive controls whether errors should be displayed to the user or not. It can be set to `On` or `Off`. For example, to display errors, you can set it as follows:

```
display_errors = On
```

Or in PHP code:

```
ini_set('display_errors', '1');
```

- It is advisable to set this directive to `On` during development and `Off` (the integer `0`) in production to avoid exposing sensitive information to users.
- `post_max_size`: This directive sets the maximum size of POST data that PHP will accept. It is specified in bytes. For example, to set the maximum POST size to 8 megabytes, you can set it as follows:

```
post_max_size = 8M
```

Or in PHP code:

```
ini_set('post_max_size', '8M');
```

- `max_execution_time`: This directive sets the maximum time in seconds that a PHP script is allowed to run before it is terminated by the parser. For example, to set the maximum execution time to 30 seconds (which is the default), you can set it as follows:

```
max_execution_time = 30
```

Or in PHP code:

```
ini_set('max_execution_time', '30');
```

Know that if the script runs beyong the `max_execution_time`, a fatal error will be thrown.

- `memory_limit`: This directive sets the maximum amount of memory that a PHP script is allowed to allocate / consume. It is specified in bytes. For example, to set the memory limit to 128 megabytes (default), you can set it as follows:

```
memory_limit = 128M
```

Or in PHP code:

```
ini_set('memory_limit', '128M');
```

- `file_uploads`: This directive controls whether file uploads are allowed in PHP. It can be set to `On` or `Off`. For example, to enable file uploads, you can set it as follows:

```
file_uploads = On
```

Or in PHP code:

```
ini_set('file_uploads', '1');
```

`Off` (the integer `0`) is used to disable file uploads.

- `upload_tmp_dir`: This directive sets the temporary directory where uploaded files are stored before they are moved to their final destination. For example, to set the upload temporary directory to `/tmp/uploads`, you can set it as follows:

```
upload_tmp_dir = /tmp/uploads
```

Or in PHP code:

```
ini_set('upload_tmp_dir', '/tmp/uploads');
```

`upload_max_filesize`: This directive sets the maximum size of an uploaded file that PHP will accept. It is specified in bytes. For example, to set the maximum upload file size to 2 megabytes, you can set it as follows:

```
upload_max_filesize = 2M
```

Or in PHP code:

```
ini_set('upload_max_filesize', '2M');
```

The `ini_get()` function

- This function is used to retrieve the value of a configuration option at runtime. The syntax of the `ini_get()` function is as follows:

```
ini_get ( string $option ) : string|false
```

where,

- `option` is the name of the configuration option to be retrieved.
- The function returns the current value of the specified configuration option as a string, or `false` if the option does not exist.

- For example suppose you want to check the current value of the `memory_limit` directive, you can use the `ini_get()` function as follows:

```
$memory_limit = ini_get('memory_limit');
echo "Current memory limit: " . $memory_limit;
```

PHP Error Handling

- There are different types of errors in PHP:
  - **Parse errors**: These occur when there is a syntax error in the code, such as a missing semicolon or an unmatched parenthesis. Parse errors are detected by the PHP parser before the script is executed.
  - **Fatal errors**: These occur when the script encounters a critical error that prevents it from continuing execution, such as calling a non-existent function or class. Fatal errors result in the termination of the script.
  - **Warning errors**: These occur when the script encounters a non-critical error, such as including a non-existent file or using an undefined variable. Warning errors do not terminate the script, but they may affect its behavior.
  - **Notice errors**: These occur when the script encounters a minor issue, such as accessing an undefined variable or using a deprecated function. Notice errors do not terminate the script and are often used for debugging purposes.
- PHP determines what error to report based on the `error_reporting` directive in the `php.ini` configuration file or using the `error_reporting()` function in the code. The `error_reporting` directive specifies the types of errors that should be reported by PHP. For example, setting `error_reporting` to `E_ALL` will report all types of errors, while setting it to `E_ERROR | E_WARNING` will only report fatal errors and warning errors. Setting it to `0` will disable error reporting altogether (`error_reporting(0)`).
- Make sure all error levels are stored in a constant. You can also combine multiple error levels using the bitwise operators. Suppose you want to combine all errors except warnings, you can do that as follows

```
error_reporting(E_ALL & ~E_WARNING);
```

- A list of the predefined error constants can be found in the PHP documentation. In this list, the ones with the `_USER_` tag are generated by using the `trigger_error()` function

PHP `trigger_error()` function

- This function is used to generate a user-level error message in PHP. It allows you to create custom error messages that can be handled by the PHP error handling mechanism.
- The syntax of the `trigger_error()` function is as follows:

```
trigger_error ( string $message , int $error_type = E_USER_NOTICE ) : bool
```

where,

- `message` is a string that represents the error message to be generated.
- `error_type` is an optional parameter that specifies the type of error to be generated. It can take one of the following predefined error constants:
  - `E_USER_NOTICE`: Generates a user-level notice message.
  - `E_USER_WARNING`: Generates a user-level warning message.
  - `E_USER_ERROR`: Generates a user-level fatal error message.
- The function returns `true` if the error was successfully triggered, or `false` if there was an error triggering the error.

- For example, the following code shows how to use the `trigger_error()` function to generate a user-level warning message:

```
$age = -5;
if($age < 0) {
    trigger_error("Age cannot be negative.", E_USER_WARNING);
}
```

- The code checks if the `$age` variable is less than `0`. If it is, it calls the `trigger_error()` function to generate a user-level warning message indicating that age cannot be negative. Thus the output becomes

```
Warning: Age cannot be negative. in /path/to/script.php on line X
```

- To access the PHP error log file, you can check the `error_log` directive in your `php.ini` configuration file. This directive specifies the path to the error log file where PHP will write error messages. Or you can check through XAMPP as follow !['PHP Error Log'](PHP Error Handling.png)

PHP `error_log()` function

- The `error_log()` function in PHP is used to send an error message to the web server's error log or to a specified file. This function allows you to log custom error messages for debugging and troubleshooting purposes.
- The syntax of the `error_log()` function is as follows:

```

```

```
error_log ( string $message , int $message_type = 0 , ?string $destination = null
, ?string $extra_headers = null ) : bool
```

where,

- `message` is a string that represents the error message to be logged.
- `message_type` is an optional parameter that specifies the type of message to be logged. It can take one of the following values:
    - `0`: Logs the message to the web server's error log (default).
    - `1`: Sends the message as an email to the address specified in the `destination` parameter.
    - `3`: Appends the message to the file specified in the `destination` parameter.
    - `4`: Sends the message directly to the SAPI logging handler.