Unit – II

➢ Functions :

I] Defining Functions :

1. **Function Basics:**
   - In PHP, a function is a block of code that can be defined and called for reusability and manageable units.
   - They play a crucial role in organising and simplifying code.
   - Functions are defined using the `function` keyword, followed by a function name and a pair of parentheses.

2. **Function Syntax:**
   - Function declaration: **(With Parameters , Without Parameters)**

```
function functionName($parameter1, $parameter2, ...)
{
    // Function code
}
```

   - `$parameter1`, `$parameter2`, etc., are optional and can be used to pass data to the function.

3. **Function Naming Rules:**
   - Function names are case-insensitive. (Not case sensitive)
   - They must start with a letter or underscore.
   - They can contain letters, numbers, and underscores.

4. **Function Parameters:**
   - Parameters are variables passed into a function.
   - They can be used to pass data to the function.
   - Example:

```
function greet($name) {
    echo "Hello, $name!";
}
greet("John");     // Output: Hello, John!
```

5. **Return Statement:**
   - Functions can return values using the `return` statement.
   - Example:

```
function add($a, $b) {
   return $a + $b;
}
$result = add(5, 3);            // $result will be 8
echo $result;
```

6. Example :

(i) With Parameters :

```
function addNumbers($num1, $num2)
{
   $sum = $num1 + $num2;
   return $sum;                  // Return the result
}
$result = addNumbers(5, 7);  //
echo $result;                  // Output: 12
```

(ii) Without Parameters :

```
function sayHello()
{
   echo "Hello, World!";
}
// Call the function
sayHello();
```

II] Calling / Invocation a Function in PHP : (8)

7. **Function Invocation: (Calling a Function)**

- In PHP, We can call a function by using its name followed by parentheses, and We can pass any required arguments inside the parentheses. Here's an example of calling a function in PHP:

- Let's say We have a function that calculates the square of a number:

```
function square($number) {
    return $number * $number;
}
```

- We can call this function and use its return value like this:

```
$number = 5;                  // The number we want to square
$result = square($number);    // Call the function and store the result in $result
echo "The square of $number is $result";  //
```

Output:
The square of 5 is 25

In this example:

1. We define a function named `square` that takes one parameter, `$number`.
2. We assign the value `5` to the variable `$number`.

3. We call the `square` function, passing `$number` as an argument, which calculates and returns the square of the number.
4. We store the result of the function call in the variable `$result`.
5. Finally, we use `echo` to display the result in a sentence.

The output of this code will be "The square of 5 is 25," as the `square` function calculates the square of the number 5 and returns 25.

(i) With Parameters :

functionName($arg1, $arg2);

(ii) Without Parameters :

functionName();

III] Variable Scope:

In PHP, variables used inside functions have different scopes, which determine where and how those variables can be accessed. There are primarily two types of variable scopes in PHP functions: local and global.

1. Local Scope (Function Scope):

- Variables defined within a function are said to have local scope.
- Local variables are only accessible within the function in which they are defined.
- They are not accessible outside of the function.

Example of local scope :

```
function myFunction() {
    $localVariable = "I am local!";
    echo $localVariable;
}
myFunction();                    // Output: I am local!
echo $localVariable; // This will result in an error because $localVariable is not defined in the global scope.
```

2. Global Scope:

- Variables defined outside of all functions are said to have global scope.
- Global variables can be accessed from anywhere in the script, including inside functions.
- To access a global variable within a function, We need to use the `global` keyword or the `$GLOBALS` superglobal.

Example of global scope with `global` keyword :

```
$globalVariable = "I am global!";

function myFunction() {
    global $globalVariable; // Access the global variable
    echo $globalVariable;
}
myFunction(); // Output: I am global!
```

3. Static Variables:

- PHP also allows for the use of static variables inside functions.
- Static variables retain their values between function calls but have local scope.
- They are declared using the **`static`** keyword.

Example of a static variable:

```
function countCalls() {
    static $count = 0;
    $count++;
    echo "Function has been called $count times."."\n";
}

countCalls();          // Output: Function has been called 1 times.
countCalls();          // Output: Function has been called 2 times.
countCalls();          // Output: Function has been called 3 times.
```

Understanding variable scope is important when working with functions in PHP, as it determines where a variable can be used and whether changes to a variable inside a function affect its value outside of that function.

IV] Function Parameters :

Introduction to Function Parameters :

- In PHP, function parameters are variables or values that We pass to a function when we call it.
- Parameters are used to provide input to a function so that it can perform a specific task.
- There are two different ways to pass the parameter to the function. The first and more common, Is By value and the other one is By Reference.

A] Passing Parameters by Value :

- When we pass parameters by value to a function, a copy of the parameter's value is passed to the function.
- Any changes made to the parameter within the function do not affect the original value outside the function.
- This is the default way of passing parameters in PHP.
- This way is useful when we want to work with the parameter's value without modifying the original variable.

Example :

1)

```
function incrementByValue($number) {
   $number++;
}
$value = 5;
incrementByValue($value);
echo $value; // Output: 5    (original value is unchanged)
```

(OR)

```
function squareByValue($number) {
   $number = $number * $number;
   echo "Square is $number";
}
$value = 5;
squareByValue($value);        // Output : Square is 25
```

2)

```
function squareByValue($number) {
   $number = $number * $number;
   return $number;
}
$value = 7;
$squaredValue = squareByValue($value);
echo "Original value: $value";                    // Output: Original value: 7
echo "Squared value: $squaredValue";         // Output: Squared value: 49
```

In this example, we have a function `squareByValue` that takes a parameter by value, which is `$number`. It calculates the square of the number and returns it. When we call this function with the value `7`, it doesn't modify the original variable `$value`, and we get the squared value as a result.

B] Passing Parameters by Reference :

- When We pass parameters by reference, We use the ampersand ' `&` ' symbol before the parameter name in both the function definition and the function call.
- This allows the function to work directly with the original variable, and any changes made to the parameter within the function will affect the original variable.
- Be cautious when using this method, as it can lead to unexpected behavior if not used carefully.

Example:

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

Key Points :

- Passing by value creates a copy of the variable's value, while passing by reference allows the function to work with the original variable.
- Passing by reference is denoted with the ampersand '`&`' symbol.
- Be mindful of potential side effects when passing by reference, as it can lead to unexpected behavior if not used carefully.

These two methods of passing parameters give us flexibility when working with functions in PHP, allowing us to choose whether us want to work with copies of values (pass by value) or modify the original variables (pass by reference).

C] Default Parameters :

1. **Introduction to Default Parameters**:
   - Default parameters, also known as default arguments, are values that We assign to function parameters in case the caller doesn't provide a value when calling the function.

2. **Defining Default Parameters** :
   - When defining a function in PHP, we can assign default values to one or more parameters by using the assignment operator (=) within the parameter list.

```
function greet($name, $greeting = "Hello") {
    echo $greeting . ", " . $name;
}
greet("Harry");                    //Output : Hello, Harry
```

In this example, the "greet" function expects two parameters, one is "$name", and the other is "$greeting" parameter has a default value of "Hello", so if we not provided argument to greeting when calling the function, the default value is used. Otherwise, it shows **Fatal Error**.

➢ Key Points :

1. **Enhanced Flexibility** :
   - Default parameters make function parameters optional, allowing for greater flexibility when calling functions.
2. **Optional Values** :
   - Default parameters allow We to specify default values for function arguments, making them optional when calling the function.
3. **Customization** :
   - Users can override default parameter values by providing their own values when calling the function, which allows customization.
4. **Default Parameters and Type Hinting** :
   - Default parameters can be used in conjunction with type hinting to specify both data types and default values for parameters.
5. **Middle Parameters with Defaults** :
   - We can have parameters with default values in the middle of the parameter list, but any subsequent parameters must be provided if We skip a parameter with a default value.

Default parameters are a valuable feature in PHP that makes functions more flexible, user-friendly, and versatile, and they are commonly used to create functions that can accommodate a variety of use cases.

D] Variable Parameter :

- Variable parameters, also known as variadic functions, allow us to work with a variable number of arguments in a function.
- The functions `func_get_args()`, `func_num_args()`, and `func_get_arg()` are used for working with variable parameters in PHP.

1. `func_get_args()` :

- **`func_get_args()`** is used to retrieve all arguments passed to a function as an indexed array. It captures all arguments, whether they are explicitly named in the function's parameter list or not.

Example:

```
function sum() {
   $args = func_get_args();        // Get all arguments as an array
   $total = 0;
   foreach ($args as $arg) {
      $total += $arg;
   }
   return $total;
}
echo sum(1, 2, 3, 4, 5);           // Output: 15
```

In this example, the **`sum`** function accepts a variable number of arguments, and **`func_get_args()`** collects them into an array. It then calculates the sum of all provided arguments.

2. `func_num_args()`:

- **`func_num_args()`** returns the number of arguments passed to a function.

Example:

```
function numberOfArguments() {
return func_num_args();                    // Get the number of arguments
}
echo numberOfArguments(1, 2, 3);   // Output: 3
```

This example shows a simple function that returns the number of arguments passed to it.

3. `func_get_arg()`:

- **`func_get_arg()`** is used to access a specific argument by its position (index) within the list of arguments. It takes an index as an argument, with the first argument having an index of 0.

Example :

```
function getSecondArgument() {
   return func_get_arg(1);                 // Get the second argument (index 1)
}
echo getSecondArgument(10, 20, 30);     // Output: 20
```

In this example, the **`getSecondArgument`** function retrieves the second argument (index 1) passed to it.

E] Missing Parameter :

In PHP, a "missing parameter" typically refers to a situation where a function or method is called with fewer arguments than it expects. This can lead to errors or unexpected behaviour, depending on how the function is designed. Here's an example of a missing parameter :

Example :

```
function greet($name, $greeting) {
    echo $greeting . ", " . $name;
}
greet("John");
```

In this example, the **`greet`** function expects two parameters: **`$name`** and **`$greeting`**. However, when calling the function, only one argument, **`"John"`**, is provided. This results in a missing parameter issue because the required parameter **`$greeting`** was not supplied.

- At this time, if we not supplied a value to a parameter, it causes error or warning :

1. **Fatal Error**: If error reporting is set to a high level, PHP will generate a fatal error due to the missing parameter. The error message will indicate that the function is missing an argument.

2. **Undefined Variable Warning** : If error reporting is not set to the highest level, PHP may issue an "Undefined variable" warning because the variable **`$greeting`** was not defined in the function's scope.

F] Type Hinting :

Type hinting in PHP allows you to specify the data type of a function's parameters or return value. Here's an example of type hinting for return values,

Return Type Hinting:

```
class Calculator {
    public function add(float $a, float $b): float {
        return $a + $b;
    }
}
```

```
$calculator = new Calculator();
$result = $calculator->add(2.5, 3.5);
echo "The sum is: " . $result;                    //Output : The sum is: 6
```

- In this example, the **`Calculator`** class has a method **`add`** that takes two float values as parameters and returns a float.
- The return type of the **`add`** method is specified using return type hinting (**`: float`**), ensuring that the result of the method is of the specified data type.
- When you call the **`add`** method, it ensures that the arguments are of the correct type (**`float`**) and that the result is also a **`float`**. This helps catch potential type-related errors.

Type hinting improves code quality by making it clear what data types are expected for function parameters and return values. It helps prevent type-related errors and enhances code readability and maintainability. When used consistently, it also serves as a form of documentation for your code.

---

6. Variable Function :

In PHP, a "variable function" refers to the capability of treating a function name as a variable. This means you can use a variable to call a function dynamically based on the value of that variable. Variable functions are particularly useful when you need to select a function to execute at runtime, based on certain conditions or parameters.

Here's a definition and an explanation of variable functions in PHP:

Definition: A variable function in PHP is a function whose name is determined by a variable. It allows you to dynamically invoke functions based on the value of a variable.

Explanation: Variable functions are useful in scenarios where you need to switch between multiple functions or call functions conditionally. Here's a basic example to illustrate how variable functions work:

```
function sayHello() {
   echo "Hello, World!";
}
function sayGoodbye() {
   echo "Goodbye, World!";
}
$functionName = "sayHello"; // Assign the function name to a variable
$functionName(); // Call the function based on the variable's value
$functionName = "sayGoodbye"; // Change the variable's value
$functionName(); // Call a different function based on the updated variable
```

In this example, we have two functions, **`sayHello`** and **`sayGoodbye`**. We use a variable, **`$functionName`**, to store the name of the function we want to call. By appending **`()`** to the variable containing the function name, we can dynamically invoke the function.

Variable functions are often employed in situations such as:

1. **Callback Functions**: You can use variable functions as callbacks in functions like `array_map()`, `array_filter()`, and custom event handlers.
2. **Dynamic Function Selection**: When you need to choose a function to execute based on certain conditions, such as in a menu system, a plugin system, or feature toggles.
3. **Function Wrappers**: In cases where you want to create a wrapper function around different implementations based on a configuration or runtime variable.

While variable functions can be powerful and flexible, they should be used with caution. It's important to sanitize user inputs and validate function names to prevent security vulnerabilities, such as code injection. Additionally, you should aim for clear and readable code to avoid confusion when dynamically invoking functions based on variables.

---

7. Anonymous Function :

An anonymous function in PHP, often referred to as a "lambda function" or a "closure," is a function without a specified name. Anonymous functions are particularly useful when you need to create small, one-time-use functions or when you want to pass functions as arguments to other functions. Here's an explanation and an example of an anonymous function in PHP:

Explanation:

- Anonymous functions are functions that are defined without a specific name. They are created using the `function` keyword, followed by an optional list of parameters, and the function body enclosed in curly braces.
- Anonymous functions can be assigned to variables or passed as arguments to other functions. They are typically used for short, simple tasks.

Example: Here's a basic example of an anonymous function that adds two numbers:

```
$add = function ($a, $b) {
   return $a + $b;
};
$result = $add(5, 3); // Call the anonymous function
echo "Result: $result"; // Output: Result: 8
```

In this example:

1. We define an anonymous function and assign it to the variable `$add`.
2. The anonymous function takes two parameters, `$a` and `$b`, and returns their sum.
3. We then call the anonymous function stored in the `$add` variable and pass it the arguments `5` and `3`, resulting in the sum of 8.

Anonymous functions are particularly handy when you need to create short, reusable code blocks, especially for callback functions in functions like `array_map()`, `array_filter()`, or when working with event handlers in libraries and frameworks. They offer flexibility and compactness in your code.

➢ Strings :

A string is a sequence of characters, like "Hello world!".

I] Quoting String Constants :

Quoting string constants in PHP can be done using single quotes (**'**), double quotes (**"**), and the here document (heredoc) format. Each of these methods has its own characteristics and use cases. Here are examples :

1. Variable Interpolation :

Variable interpolation is a feature in programming languages that allows you to insert the values of variables directly into strings. This makes it easy to create dynamic strings by embedding variable values within them. In PHP, variable interpolation is commonly used in strings enclosed in double quotes (" "). Here's how variable interpolation works in PHP :

- **Using Variable Interpolation in PHP :**

In PHP, you can include variable values within double-quoted strings. Variables are denoted by a dollar sign ($) followed by the variable name, and they will be replaced with their actual values when the string is evaluated.

```
$name = "Harry";
$age = 30;

// Variable interpolation in a double-quoted string
$message = "My name is $name, and I am $age years old.";

// Output : My name is Harry, and I am 30 years old.
```

In this example, the variables $name and $age are interpolated into the string, and the resulting string stored in the $message variable will be "My name is John, and I am 30 years old."

2. Single Quotes (' '):

- Enclosing a string in single quotes is the simplest way to create a string constant.
- The string will be treated literally, and variables or escape sequences within it won't be interpreted.
- Useful for simple strings without variables or special characters.

Example :

```
$variable = 'world';
echo 'Hello, $variable!';              // Output: Hello, $variable!
```

3. Double Quotes (" "):

- Enclosing a string in double quotes allows for variable interpolation, escape sequences, and special characters.
- Variables enclosed in double quotes will be replaced with their values.
- Useful when you need to include variables or special characters within the string.

Example :

```
$variable = 'world';
echo "Hello, $variable!";          // Output: Hello, world!
```

4. Here Document (Heredoc):

- Heredoc is a way to define multiline strings without the need for escaping quotes or special characters.
- It's especially useful when working with large blocks of text.
- Heredoc syntax begins with `<<<`, followed by a delimiter (often `EOT`, but you can use any identifier), and ends with the same delimiter.

Example :

```
$variable = 'world';
$message = <<< 'EOT'
Hello, $variable!
This is a multiline string.
EOT;

echo $message;                    // Output : Hello, World
                                             This is multiline string.
```

In this example, `<<<EOT` marks the start of the heredoc string, and `EOT;` marks the end. Variables are interpolated, and line breaks and special characters are preserved.

II] Printing Strings :

In PHP, you can print strings to the screen using various methods. The most common way to display a string is by using the **`echo`** and **`print`** statements. Here's how you can do it:

1. Using echo :

```
$str = "Hello, World!";
echo $str;                  //Output : Hello, World!
```

You can also echo multiple strings or variables :

```
$name = "John";
$age = 30;
echo "My name is " . $name . " and I am " . $age . " years old.";

// Output : My name is John and I am 30 years old.
```

2. Using print :

```
$str = "Hello, World!";
print($str);                // Output : Hello, World!
```

Both **echo** and **print** are used for displaying strings, but there are some differences between them:

- **echo** can take multiple arguments separated by commas and does not return a value. It's slightly faster.
- **print** can only take a single argument and returns 1 (always). It's slightly slower.

To output strings is by using the printf function for formatted output :

```
$name = "John";
$age = 30;
printf("My name is %s and I am %d years old.", $name, $age);
```

3. Using printf() :

In PHP, the **printf()** function is used for formatted output, allowing you to control the format and placement of variables within a string. To format output using **printf()**, you can use format modifiers and type specifiers. Format modifiers specify how a value should be displayed, while type specifiers indicate the type of the

variable you want to display. Here's a list of some commonly used format modifiers and type specifiers:

Format Modifiers:

1. **%** - This is the character used to indicate that a format specifier follows.
2. **- (dash)** - Left-justifies the variable within the given width.
3. **+** - Forces a sign (+ or -) to be displayed in front of a positive number.
4. **0 (zero)** - Pads the number with leading zeros instead of spaces.
5. **#** - Prefixes numbers with "0x" for hexadecimal or "0" for octal.
6. **.** - Specifies a precision for floating-point numbers. For example, %0.2f will display two decimal places.

Type Specifiers:

1. **%s - String.** Used to format strings.
2. **%d - Integer**. Used to format whole numbers (integers).
3. **%f - Floating-point number**. Used for formatting floating-point or double-precision numbers.
4. **%c - Character**. Used to format a single character.
5. **%b - Binary**. Used to format numbers in binary (e.g., %b for 9 will display "1001").
6. **%o - Octal**. Used to format numbers in octal (e.g., %o for 9 will display "11").
7. **%x - Lowercase hexadecimal**. Used to format numbers in lowercase hexadecimal (e.g., %x for 15 will display "f").
8. **%X - Uppercase hexadecimal**. Used to format numbers in uppercase hexadecimal (e.g., %X for 15 will display "F").

Examples:

Here are some examples of how you can use format modifiers and type specifiers with printf():

```
$name = "John";
$age = 30;
$height = 5.75;

printf("Name: %s, Age: %d, Height: %0.2f", $name, $age, $height);
```

In this example:

- `%s` is used for formatting the string `$name`.
- `%d` is used for formatting the integer `$age`.
- `%0.2f` is used for formatting the floating-point number `$height`, with two decimal places.

When you run this code, it will format and display the variables as specified in the `printf()` statement.

`var_dump()` and `print_r()` are two important functions in PHP used for debugging and examining the structure and values of variables, arrays, and objects. Here are some key notes on their use:

4. var_dump() :

1. **Purpose**: `var_dump()` is primarily used for detailed and comprehensive debugging. It provides a lot of information about the variable or expression, including its data type, value, and structure.
2. **Syntax**: `var_dump($variable)` where `$variable` is the variable or expression you want to inspect.
3. **Usage**:
   - It's useful for debugging complex data structures like arrays and objects.
   - It's especially helpful when you need to examine the data type and length of a variable.
   - It shows the type and size of variables, which can be essential for understanding issues with data types.
   - It's commonly used during development but should be avoided in production code due to its verbosity.
4. **Output**: The output includes the data type, value, and, for arrays and objects, the structure.
5. **Example**:
   ```
   $array = [1, 2, 3];
   var_dump($array);
   ```

5. print_r() :

1. **Purpose**: `print_r()` is used for displaying the structure of arrays or objects in a human-readable format. It's more concise and user-friendly than `var_dump()`.
2. **Syntax**: `print_r($array)` where `$array` is the variable or expression you want to display.
3. **Usage**:
   - It's helpful for quick and easy inspection of array or object structures.
   - It's commonly used when you want to see the contents and structure of an array or object in a readable format.

- Unlike `var_dump()`, `print_r()` is safe to use in production code as it produces cleaner output.

4. **Output**: The output displays the structure of arrays and objects in a readable format, showing keys and values.
5. **Example**:
   ```
   $array = [1, 2, 3];
   print_r($array);
   ```

Comparison:

- `var_dump()` provides more detailed information, making it suitable for debugging complex issues.
- `print_r()` is simpler and easier to read, making it ideal for quick inspections of array and object structures.

In summary, `var_dump()` is a powerful debugging tool for detailed analysis of variables, while `print_r()` is a more user-friendly choice for quick examination of array and object structures. You can use them according to your specific debugging needs.

III] Accessing Individual Characters :

In PHP, we can access individual characters from a string by using various methods, including array-like indexing, the substr() function, and character iteration. Here are some common ways to access individual characters from a string :

1. Using Array-Like Indexing :

We can treat a string like an array of characters and access individual characters using square brackets and the character's position (index). Keep in mind that PHP uses a 0-based index.

```
$str = "Hello, World!";
$char = $str[0];                    // Access the first character ('H')
```

2. Using the substr() Function:

The substr() function allows you to extract a substring starting from a given position (index). You can use it to access individual characters.

```
$str = "Hello, World!";

$char = substr($str, 4, 1);          // Get the character at index 4 ('o')
```

3. Using a Loop :

We can iterate over the characters of a string using a loop, such as a for or foreach loop, to access and process individual characters.

```
$str = "Hello";
for ($i = 0; $i < strlen($str); $i++) {
   $char = $str[$i];
   echo "\n";
   echo $char;
// Process or print $char
}
```

IV] Cleaning Strings :

Often, the strings we get from files or users need to be cleaned up before we can use them. Two common problems with raw data are the presence of extraneous whitespace and incorrect capitalization (uppercase versus lowercase).

1. Removing Whitespace :

Cleaning and removing whitespace from strings is a common task in programming, and it can be done in various ways in PHP. Here are some points on how to remove whitespace from strings :

Using trim(), ltrim(), and rtrim():

- The **trim()** function removes whitespace (spaces, tabs, and newline characters) from the beginning and end of a string.
- **ltrim()** removes whitespace from the beginning (left) of a string, while **rtrim()** removes whitespace from the end (right).
- These functions are helpful when you want to clean user input or remove unnecessary spaces.

Example :

```
$string = "  Hello, World!  ";
$trimmed = trim($string);
echo $trimmed;              // Output: "Hello, World!"
```

2. Changing Case :

PHP has several functions for changing the case of strings: **strtolower()** and **strtoup per()** operate on entire strings, **ucfirst()** operates only on the first character of the string, and **ucwords()** operates on the first character of each word in the string. Each function takes a string to operate on as an argument and returns a copy of that string, appropriately changed.

Example:

```
$string1 = "FRED flintstone";
$string2 = "barney rubble";
print(strtolower($string1));
print(strtoupper($string1));
print(ucfirst($string2));
print(ucwords($string2));
```

fred flintstone
FRED FLINTSTONE
Barney rubble
Barney Rubble

If we've got a mixed-case string that you want to convert to "title case," where the first letter of each word is in uppercase and the rest of the letters are in lowercase (and you are not sure what case the string is in to begin with), use a combination of strto lower() and ucwords():

```
print(ucwords(strtolower($string1)));
```

Fred Flintstone

V] Encoding and Escaping :

Encoding and escaping in PHP refer to the process of converting special characters in a string to their corresponding HTML or URL entities, so they can be safely displayed on a webpage or used in a URL without causing or used in a URL without causing any issues with the underlying code. Here are some point-wise notes on encoding and escaping in HTML, URLs, SQL, and C when working with strings:

HTML String Encoding :

1. **Purpose** : HTML encoding is used to prevent cross-site scripting (XSS) attacks by converting characters with special meaning in HTML into their corresponding HTML entities.
2. **Functions** : In PHP, you can use htmlspecialchars() to HTML-encode strings. It converts characters like <, >, &, etc., to their HTML entities.

```
$input = "<script>alert('XSS');</script>";
$encoded = htmlspecialchars($input);
echo $encoded;       // Output: &lt;script&gt;alert('XSS')&lt;/script&gt;
```

3. **Use Cases** : Use HTML encoding when displaying user-generated content in web pages to prevent XSS attacks.

URL String Encoding :

1. **Purpose**: URL encoding is used to safely include special characters in URLs.
2. **Function**: In PHP, you can use `urlencode()` to URL-encode strings. It converts characters like spaces and special characters into percent-encoded values.

```
$input = "Hello World!";
$encoded = urlencode($input);
echo $encoded;          // Output: Hello+World%21
```

3. **Use Cases :** Use URL encoding when creating or manipulating URLs in web applications to handle special characters.

SQL String Escaping :

1. **Purpose**: SQL escaping is used to prevent SQL injection by escaping special characters in SQL queries.
2. **Functions**: In PHP, you can use `mysqli_real_escape_string()` or prepared statements to escape strings for use in SQL queries.

```
$input = "John's book";
$escaped = mysqli_real_escape_string($connection, $input);
$query = "SELECT * FROM books WHERE title = '$escaped'";
```

3. **Use Cases**: Use SQL escaping when embedding user-provided data in SQL queries to prevent SQL injection.

C String Encoding :

1. **Purpose**: C string encoding is used to handle special characters and escape sequences in C/C++ strings.

2. **Escaping**: In C/C++, you can use escape sequences like `\n` (newline), `\t` (tab), `\"` (double quote), and `\\` (backslash) to represent special characters within strings.

   char* message = "This is a C string\nwith a newline.";

3. **Use Cases**: Use C string encoding when working with strings in C/C++ to represent special characters or escape sequences.

In summary, encoding and escaping are important for maintaining data integrity, security, and correct interpretation in various contexts. Be sure to use the appropriate encoding or escaping techniques depending on the environment and the specific type of data you are handling.

VI] Comparing Strings :

In PHP, we can compare strings in various ways, depending on our specific requirements. Here are some two common methods for comparing strings in PHP :

- **Exact Comparison :**

The === operator checks for both the content and the data type to be identical. This means that not only should the strings have the same characters but they should also be of the same data type (string).

Example :

```
$string1 = "Hello";
$string2 = "Hello";

if ($string1 === $string2) {
    echo "Strings are exactly equal.";
} else {
    echo "Strings are not exactly equal.";
}

// Output : Strings are exactly equal.
```

In Exact Comparison, we can compare strings in various ways, depending on our specific requirements. Here are some common methods for comparing strings :

1. **Equality Comparison (==) :** We can use the double equals (**==**) operator to compare if two strings are equal in terms of their content. This comparison is case-insensitive.

   Example :

   ```
   $string1 = "Hello";
   ```

```
$string2 = "hello";

if ($string1 == $string2) {
   echo "Strings are equal.";
} else {
   echo "Strings are not equal.";
}
```
// Output : Strings are not equal.

2. **Case-Insensitive Comparison :** To compare strings while ignoring the case (make it case-insensitive), you can use functions like **strcasecmp()** or **stricmp()** :

   Example :

```
$string1 = "Hello";
$string2 = "hello";
if (strcasecmp($string1, $string2) == 0) {
   echo "Strings are equal (case-insensitive).";
} else {
   echo "Strings are not equal (case-insensitive).";
}
```
   // Output : Strings are equal (case-insensitive).

3. **String Comparison Functions :** PHP provides several functions for comparing strings, including **strcmp()** and **strnatcmp()**. These functions return 0 if the strings are equal and a positive or negative number depending on their order:

   Example :

```
$string1 = "apple";
$string2 = "banana";
$result = strcmp($string1, $string2);
if ($result == 0) {
   echo "Strings are equal.";
} elseif ($result < 0) {
   echo "String 1 is less than String 2.";
} else {
   echo "String 1 is greater than String 2.";
}
```
   // Output : String 1 is less than String 2.

4. **String Length Comparison :** We can also compare strings based on their length using the **strlen()** function or by comparing their length directly.
   Example :

```
$string1 = "Hello";
$string2 = "Hello, world!";

if (strlen($string1) == strlen($string2)) {
    echo "Strings have the same length.";
} else {
    echo "Strings have different lengths.";
}
// Output : Strings have different lengths.
```

- ## Appropriate Equality Comparison :

PHP provides several functions that allow us to test whether two strings are approximately equal. The Functions **soundex()**, **metaphone()**, **similar_text()** and **levenshtein()** are useful for comparing strings based on their phonetic or similarity characteristics. Here's an overview of these functions:

1. soundex() :

- The soundex() function calculates a phonetic representation of a string.
- It's often used to compare words that sound similar.
- It returns a four-character code representing the string's phonetic sound.
- We can use soundex() to find words that have similar pronunciations.

Example :

```
$word1 = "hello";
$word2 = "helo";

if (soundex($word1) === soundex($word2)) {
    echo "Words sound similar.";
} else {
    echo "Words do not sound similar.";
}
//   Output : Words sound similar.
```

2. metaphone() :

- The **metaphone()** function provides another way to create a phonetic representation of a string.
- It returns a metaphone key, which is a more modern and flexible alternative to soundex.
- **metaphone()** is used to find similar-sounding words.

Example :

```
$word1 = "write";
$word2 = "right";

if (metaphone($word1) === metaphone($word2)) {
    echo "Words sound similar.";
} else {
    echo "Words do not sound similar.";
}
```
// Output : Words do not sound similar.

3. similar_text():

- The **similar_text()** function calculates the similarity between two strings as a percentage.
- It returns the number of matching characters between the two strings.
- This function is useful for finding the similarity between two strings, regardless of phonetics.

Example :

```
$string1 = "programming";
$string2 = "programmer";

similar_text($string1, $string2, $percentage);
echo "Similarity: $percentage%";
```
// Output : Similarity : 76.190476190476 %

4. levenshtein() :

- The **levenshtein()** function calculates the Levenshtein distance between two strings.
- It measures the number of single-character edits (insertions, deletions, substitutions) required to change one string into another.
- A smaller Levenshtein distance indicates greater similarity.

Example :

```
$string1 = "kitten";
$string2 = "sitting";

$distance = levenshtein($string1, $string2);
echo "Levenshtein distance: $distance";
```
// Output : Levenshtein distance : 3

VII] Manipulation and Searching :

String manipulation and searching are common operations in PHP. PHP offers a variety of functions to help we manipulate and search within strings. Here are some essential functions for string manipulation and searching :

- **Substrings :**

In PHP, there are several functions related to **substr(),** which allow you to work with substrings in different ways. Here are some functions related to **substr() :**

1. substr() :

- The **substr()** function is used to extract a portion of a string based on its starting position and length.

Example :

```
$str = "Hello, world!";
$substring = substr($str, 0, 5);
echo $substring;
// Output :  "Hello"
```

2. substr_replace():

- The **substr_replace()** function replaces a portion of a string with another string.
- **Syntax : substr_replace(original_string, replacement_string, start, length)**

Example :

```
$str = "Hello, world!";
$newStr = substr_replace($str, "Universe", 7, 5);
echo $newStr;
// Output :  "Hello, Universe!"
```

3. stristr() (Case-insensitive substring search) :

- The **stristr()** function searches for the first occurrence of a substring in a string (case-insensitive) and returns the rest of the string from that point.

Example :

```
$str = "Hello, world!";
$substring = strstr($str, "world");
echo $substring;
```

// Output : "world!"

These functions allow you to manipulate and work with substrings in various ways, whether it's extracting portions of a string, replacing parts of a string, or reversing a string. Choose the function that best suits your specific string manipulation needs in PHP.

- ## Miscellaneous String Functions :

PHP provides a variety of string functions for manipulating and working with strings. These functions help you perform common string operations like searching, replacing, formatting, and more. Here are some miscellaneous string functions in PHP :

strlen($strings) , strpos(Sstring, value), str_replace($search, Sreplace , $string) strtolower($strings) , strtoupper($strings) , trim($strings) , ucfirst($string) , ucwords($string) , strrev($string), substr($string , $start , $length).

These are just a few of the many string functions available in PHP. You can use these functions to perform a wide range of string manipulation tasks in your PHP applications.

- ## Decomposing a Strings :

Decomposing strings in PHP can be achieved using various techniques, including exploding and imploding, tokenizing, and **sscanf()** (Scanf). Here's how to decompose strings with these methods :

1. Exploding and Imploding:

- **Exploding :** This method involves breaking a string into an array of substrings based on a specified delimiter.
- **Imploding :** This method takes an array of substrings and joins them into a single string using a specified glue.
- **Example :**

```
$str = "apple,banana,cherry";
// Explode the string into an array
$fruitsArray = explode(",", $str);
// Implode the array back into a string using a different delimiter
$newStr = implode("|", $fruitsArray);
echo $newStr;
```

// Output : apple|banana|cherry

2. Tokenizing using strtok():

- Tokenizing a string means breaking it into smaller parts, often based on specific delimiters. PHP's **strtok()** function allows you to tokenize a string.
- **Example :**

```
$str = "apple,orange;banana:cherry";

$delimiter = ",;:";
$token = strtok($str, $delimiter);
while ($token !== false) {
    echo "$token\n";
    $token = strtok($delimiter);
}
```

Output : apple
banana
orange
cherry

3. Using sscanf() :

- The **sscanf()** function allows us to extract data from a string using a format specifier. It works similarly to the **scanf()** function in C.
- **Example :**

```
$str = "Name: John, Age: 30, City: New York";

$format = "Name : %s, Age: %d, City: %s";
$result = sscanf($str, $format);

// $result is an array with extracted values
$name = $result[0];   // "John"
$age = $result[1];    // 30
$city = $result[2];   // "New York"
```

VIII] Searching Strings Functions :

String searching functions are essential in many programming languages, including PHP, for finding specific patterns or substrings within a larger string. These functions help we locate, extract, or manipulate data in text-based content. Below, we will provide a theoretical overview of some common string searching techniques and functions.

Let's explore some PHP functions related to string searching and manipulating URLs, categorized by their functionalities :

1. **String Searches Returning Position :**

   strpos() and **stripos() :**

   - These functions find the position of the first occurrence of a substring in a string.
   - **`strpos()`** is case-sensitive, while **`stripos()`** is case-insensitive.

   Example:

```
$haystack = "Hello, world!";
$needle = "world";
$position = strpos($haystack, $needle);     // Returns the position of "world"
echo $position;
```

   Output :

   7

2. **String Searches Returning Rest of Strings :**

   strstr() and **stristr() :**

   - These functions find the first occurrence of a substring and return the rest of the string from that position onward.
   - **`stristr()`** is case-insensitive.

   Example:

```
$haystack = "Hello, World!";
$needle = "world";
$substring = stristr($haystack, $needle);          // Returns "World!"
```

3. String Searches Using Marks :

`strchr()`:

- This function is similar to **`strstr()`**. It finds the first occurrence of a substring and returns the rest of the string from that position onward.
- It uses a different name and syntax but serves a similar purpose.

Example:

```
$haystack = "Hello, World!";
$needle = "world";
$substring = strchr($haystack, $needle);          // Returns "World!"
```

4. Decomposing URL :

`parse_url()`:

- This function is used to decompose a URL into its various components, such as scheme, host, path, query, fragment, etc.

Example:

```
<?php
$url = "https://www.example.com/path/to/page?query=string#section";
$urlComponents = parse_url($url);

echo "Scheme      : " . $urlComponents['scheme']."\n";
echo "Host        : " . $urlComponents['host']."\n";
echo "Path        : " . $urlComponents['path']."\n";
echo "Query       : " . $urlComponents['query']."\n";
echo "Fragment    : " . $urlComponents['fragment'];
?>
```

Output :

```
Scheme      : https
Host        : www.example.com
Path        : /path/to/page
Query       : query=string
Fragment    : section
```

These functions can be very useful for searching and manipulating strings as well as working with URLs in PHP. We can choose the one that best fits your specific needs, whether it's finding positions, extracting substrings, or dissecting URLs into their components.

IX] Regular Expression :

A regular expression is a string that represents a *pattern*. The regular expression functions compare that pattern to another string and see if any of the string matches the pattern. Some functions tell us whether there was a match, while others make changes to the string.

There are three uses for regular expressions : matching, which can also be used to extract information from a string; substituting new text for matching text; and splitting a string into an array of smaller chunks. PHP has functions for all. For instance, **preg_match()** does a regular expression match. Perl has long been considered the benchmark for powerful regular expressions. PHP uses a C library called **pcre** to provide almost complete support for Perl's arsenal of regular expression features. Perl regular expressions act on arbitrary binary data, so we can safely match with patterns or strings that contain the NUL-byte.

• The Basics :

Most characters in a regular expression are literal characters, meaning that they match only themselves. For instance, if you search for the regular expression "/cow/" in the string "Dave was a cowhand", you get a match because "cow" occurs in that string.

Some characters have special meanings in regular expressions. For instance, a caret (^) at the beginning of a regular expression indicates that it must match the beginning of the string (or, more precisely, anchors the regular expression to the beginning of the string) :

```
preg_match("/^cow/", "Dave was a cowhand");      // returns false
preg_match("/^cow/", "cowabunga!");              // returns true
```

Similarly, a dollar sign ($) at the end of a regular expression means that it must match the end of the string (i.e., anchors the regular expression to the end of the string) :

```
preg_match("/cow$/", "Dave was a cowhand");      // returns false
preg_match("/cow$/", "Don't have a cow");        // returns true
```

A period (.) in a regular expression matches any single character :

```
preg_match("/c.t/", "cat");      // returns true
preg_match("/c.t/", "cut");      // returns true
preg_match("/c.t/", "c t");      // returns true
preg_match("/c.t/", "bat");      // returns false
preg_match("/c.t/", "ct");       // returns false
```

Regular expressions are case-sensitive by default, so the regular expression "/cow/" doesn't match the string "COW".

- ## Character Classes :

In regular expressions, character classes are used to define a set of characters that can match a single character at a particular position in the string you're searching. In PHP, you can use the `preg_match()` function to perform regular expression matching. Character classes are defined within square brackets `[ ]` and can be used to specify a range of characters or a list of characters that can match.

Here are some common character classes and their usage in `preg_match()`:

1. **Literal Characters** : We can specify literal characters inside the character class, and they will match exactly those characters. For example, `[abc]` will match either 'a,' 'b,' or 'c' in the input string.
   ```
   $str = "The cat is on the mat.";
   if (preg_match('/[cm]at/', $str, $matches)) {
      echo "Match found: " . $matches[0];     // Output: "Match found: cat"
   }
   ```

2. **Ranges**: We can specify a range of characters using a hyphen `-` inside the character class. For example, `[A-Z]` matches any lowercase letter from 'a' to 'z'.
   ```
   $str = "The quick brown fox jumps over 9 lazy dogs.";
   if (preg_match('/[a-z]/', $str, $matches)) {
      echo "Match found: " . $matches[0]; // Output: "Match found: T"
   }
   ```

3. **Negation**: We can use a caret `^` as the first character inside the character class to match any character that is NOT in the specified class. For example, `[^0-9]` will match any character that is not a digit.
   ```
   $str = "The quick brown fox jumps over 9 lazy dogs.";
   if (preg_match('/[^0-9]/', $str, $matches)) {
      echo "Match found: " . $matches[0]; // Output: "Match found: T"
   }

   $str = "The quick brown fox jumps over 9 lazy dogs.";
   if (preg_match('/[^0-9]/', $str, $matches)) {
      echo "Match found: " . $matches[0]; // Output: "Match found: T"
   }
   ```

Shorthand Character Classes : PHP provides shorthand character classes for common character groups, such as:

- `\d`: Matches any digit, equivalent to `[0-9]`.
- `\D`: Matches any non-digit.
- `\w`: Matches any word character (alphanumeric and underscore), equivalent to `[a-zA-Z0-9_]`.
- `\W`: Matches any non-word character.
- `\s`: Matches any whitespace character (e.g., space, tab, newline).
- `\S`: Matches any non-whitespace character.

```
$str = "The 42 dogs are barking!";
if (preg_match('/\d\s\w/', $str, $matches)) {
    echo "Match found: " . $matches[0]; // Output: "Match found: 42 "
}
```

These are some common examples of character classes in regular expressions with `preg_match()` in PHP. We can combine and customize character classes to match more complex patterns in your input strings.

- ## Alternatives :

We can use the vertical pipe (|) character to specify alternatives in a regular expression:

```
preg_match("/cat|dog/", "the cat rubbed my legs");          // returns true
preg_match("/cat|dog/", "the dog rubbed my legs");          // returns true
preg_match("/cat|dog/", "the rabbit rubbed my legs");       // returns false
```

The precedence of alternation can be a surprise: "/^cat|dog$/" selects from "^cat" and "dog$", meaning that it matches a line that either starts with "cat" or ends with "dog". If you want a line that contains just "cat" or "dog", you need to use the regular expression "/^(cat|dog)$/".

We can combine character classes and alternation to, for example, check for strings that don't start with a capital letter:

```
preg_match("/^([a-z]|[0-9])/", "The quick brown fox");      // returns false
preg_match("/^([a-z]|[0-9])/", "jumped over");              // returns true
preg_match("/^([a-z]|[0-9])/", "10 lazy dogs");             // returns true
```

- ## Repeating Sequences :

To specify a repeating pattern, you use something called a quantifier. The quantifier goes after the pattern that's repeated and says how many times to repeat that pattern.

The quantifiers that are supported by both PHP's regular expressions.

Table. Regular expression quantifiers

Quantifier	Meaning
?	0 or 1
*	0 or more
+	I or more
{ n }	Exactly n times
{ n , m }	At least n, no more than m times
{ n ,}	At least n times

To repeat a single character, simply put the quantifier after the character:

```
preg_match("/ca+t/", "caaaaaaat");      // returns true
preg_match("/ca+t/", "ct");             // returns false
preg_match("/ca?t/", "caaaaaaat");      // returns false
preg_match("/ca*t/", "ct");             // returns true
```

With quantifiers and character classes, we can actually do something useful, like matching valid U.S. telephone numbers:

```
preg_match("/[0-9]{3}-[0-9]{3}-[0-9]{4}/", "303-555-1212");      // returns true
preg_match("/[0-9]{3}-[0-9]{3}-[0-9]{4}/", "64-9-555-1234");     // returns false
```

- ## Subpatterns :

We can use parentheses to group bits of a regular expression together to be treated as a single unit called a subpattern:

```
preg_match("/a (very )+big dog/", "it was a very very big dog");   // returns true
preg_match("/^(cat|dog)$/", "cat");                                // returns true
preg_match("/^(cat|dog)$/", "dog");                                // returns true
```

The parentheses also cause the substring that matches the subpattern to be captured. If you pass an array as the third argument to a match function, the array is populated with any captured substrings:

```
preg_match("/([0-9]+)/", "You have 42 magic beans", $captured);
// returns true and populates $captured
```

The zeroth element of the array is set to the entire string being matched against. The first element is the substring that matched the first subpattern (if there is one), the second element is the substring that matched the second subpattern, and so on.

- ## Delimiters :

Delimiters are used to enclose regular expressions and any associated flags. Delimiters are important because they define the beginning and end of a regular expression pattern. The most commonly used delimiter in PHP is the forward slash /, but other characters like #, %, or ~ can also be used as delimiters, as long as they are not part of the regular expression pattern itself.

We can use different delimiters for your regex pattern. Common delimiters include /, #, and ~.

For example :

```
$pattern = '/example/';
$pattern = '#example#';
$pattern = '~example~';
```

Here's a simple PHP example using preg_match() to match a regular expression with delimiters:

```
<?php
$input = "The quick brown fox jumps over the lazy dog.";
$pattern = '#fox#';  // Using a forward slash as the delimiter
if (preg_match($pattern, $input, $matches)) {
    echo "Match found: " . $matches[0];
} else {
    echo "No match found.";
}
?>
```

In this example, we use the forward slash **#** as the delimiter to define the regular expression pattern **#fox#**. It searches for the word "fox" in the input string. If a match is found, it prints the matched substring. If no match is found, it displays "No match found."

- ## Match Behaviour :

The regular expression **"/is (.*)$/"** is used to search for a substring that starts with **"is "** (note the space after **"is"**) and then captures everything that follows until the end of the string. Let's break down the regular expression and explain how it works:

1. **/ :** This is the delimiter that marks the start and end of the regular expression.
2. **is :** This part of the regular expression matches the literal characters "is ".
3. **(.*):** This is a capturing group. The . matches any character, and the * quantifier means "zero or more times," so .* matches any sequence of characters. This capturing group captures everything that follows "is ".
4. **$:** This anchors the regular expression to the end of the string.

So, when we use **preg_match()** with this regular expression and the input string "the key is in my pants," here's what happens:

1. The regular expression searches for the first occurrence of "is " in the input string. It finds "is " at the position where "is " appears in the string.
2. Once it finds "is " in the string, the (.*) part captures the rest of the string, which is "in my pants."
3. Finally, the $ ensures that the match extends to the end of the string (or just before a newline, if the string ends with a newline).

The result is that the contents of the capturing group (.*?), which is "in my pants," are stored in the $captured array at index 1.

So, after executing the code:

preg_match("/is (.*)$/", "the key is in my pants", $captured);

The $captured array will contain:

- $captured[0] will contain the entire matched string, which is "is in my pants".
- $captured[1] will contain the content captured by the capturing group (.*?), which is "in my pants."

- ## Character Classes :

In PHP, we can use the **[: :]** construct within regular expressions to match characters from a specified character class. This feature is part of **POSIX** character classes and is supported in PHP's regular expression engine.

Here's how we can use **[: :]** in a regular expression :

- **[:alpha:]** : Matches any alphabetic character (equivalent to [A-Za-z]).
- **[:alnum:]** : Matches any alphanumeric character (equivalent to [A-Za-z0-9]).
- **[:digit:]** : Matches any digit character (equivalent to [0-9]).
- **[:blank:]** : Matches a space or tab character.
- **[:space:]** : Matches any whitespace character (including space, tab, newline, etc.).
- **[:lower:]** : Matches any lowercase alphabetic character (equivalent to [a-z]).
- **[:upper:]** : Matches any uppercase alphabetic character (equivalent to [A-Z]).
- **[:punct:]** : Matches any punctuation character.
- **[:graph:]** : Matches any printable character, excluding space.
- **[:print:]** : Matches any printable character, including space.

Here's an example of using **[: :]** in a PHP regular expression to match uppercase letters:

```
$pattern = '/[[:upper:]]/';
$input = 'Hello World';
if (preg_match($pattern, $input)) {
    echo 'Uppercase letter found.';
} else {
    echo 'No uppercase letter found.'; }
```

Output : Uppercase letter found.

In this example, the regular expression **[[:upper:]]** is used to match any uppercase letter in the input string. The **preg_match** function checks if there is a match, and it will output **"Uppercase letter found"** if an uppercase letter is present in the input string.

The reason for the double brackets is that the inner brackets **[[:upper:]]** are used to specify a predefined character class, and the outer brackets **[...]** are used to define the character class.

We can use these character classes to simplify our regular expressions when we need to match specific character types.

- ## Anchors :

Anchors in regular expressions are used to specify positions within the input text where a match should occur.

In PHP's regular expressions, you can use various anchor tags to specify positions within the input text where a match should occur. Here is a list of commonly used anchor tags :

1. **^ (Caret) :** This anchor specifies the beginning of a line or the beginning of the input string. If we place ^ at the start of your regular expression pattern, it will only match if the pattern occurs at the beginning of a line or the input string.

```
$pattern = '/^Start/';
$input = 'Start something';

if (preg_match($pattern, $input)) {
    echo 'Match found at the beginning.';
} else {
    echo 'No match at the beginning.';
}
```

Output : Match found at the beginning.

In this example, the regular expression **/^Start/** matches the word **"Start"** only if it appears at the beginning of the input string.

2. **$ (Dollar Sign) :** This anchor specifies the end of a line or the end of the input string. If we place **$** at the end of our regular expression pattern, it will only match if the pattern occurs at the end of a line or the input string.

```
$pattern = '/end$/';
$input = 'something at the end';
if (preg_match($pattern, $input)) {
    echo 'Match found at the end.';
} else {   echo 'No match at the end.';  }
```

Output : Match found at the end.

3. **\b (Word Boundary) :** The **\b** anchor matches a word boundary. It can be used to ensure that a pattern only matches whole words and not parts of words.

```
$pattern = '/\bword\b/';
$input = 'This is a word. Notaword here.';

if (preg_match($pattern, $input)) {
   echo 'Match found for the whole word "word".';
} else {
   echo 'No match for the whole word "word".';
}
```

Output : Match found for the whole word "word".

In this example, the regular expression **/\bword\b/** matches the word **"word"** as a whole word and not the partial word **"Notaword."**

4. **\B (Non-Word Boundary) :** Matches a position that is not a word boundary.
5. **\A (Start of Input) :** Matches the start of the input string (similar to ^ but does not match the start of a line within a multiline input).
6. **\z (End of Input) :** Matches the end of the input string (similar to $ but does not match the end of a line within a multiline input).
7. **\Z (End of Input or Before a Final Line Break) :** Matches the end of the input string or just before the final line break (similar to \z but can match before the last line break in multiline input).
8. **\G (Start of Match or End of Previous Match) :** Matches the position where the previous match ended or, in the absence of a previous match, the start of the input string.

These anchors are used to control where in the input text a regular expression should match. They are essential for various text processing tasks, such as validation, searching, and text extraction.

- ## Quantifiers and Greed :

Quantifiers in regular expressions are used to specify how many times a character or group of characters should be matched in the input text. They determine the repetition of the preceding element in a regular expression pattern. Greed in regular expressions refers to whether a quantifier should match as much text as possible (greedy) or as little as possible (non-greedy or lazy). In PHP, you can use various quantifiers and specify greed using modifiers. Here are some common quantifiers and their greedy/non-greedy forms :

1. *** (Asterisk) :** Matches 0 or more occurrences of the preceding element.
    - **Greedy :** .* matches as much as possible.
    - **Non-greedy :** .*? matches as little as possible.

2. **+ (Plus) :** Matches 1 or more occurrences of the preceding element.
    - **Greedy :** .+ matches as much as possible.
    - **Non-greedy :** .+? matches as little as possible.

3. **? (Question Mark) :** Matches 0 or 1 occurrence of the preceding element.
    - **Greedy :** a? matches "a" if present.
    - **Non-greedy :** a?? matches "a" if present but prefers 0 occurrences.

4. **{n} (Exact Quantifier) :** Matches exactly n occurrences of the preceding element.
    - **Greedy :** a{3} matches "aaa."
    - **Non-greedy :** a{3}? matches "aaa" but prefers fewer occurrences.

5. **{n,} (Minimum Quantifier) :** Matches n or more occurrences of the preceding element.
    - **Greedy :** a{2,} matches "aa" or more.
    - **Non-greedy :** a{2,}? matches "aa" or more but prefers fewer occurrences.

6. **{n,m} (Range Quantifier) :** Matches between n and m occurrences of the preceding element.
    - **Greedy** : a{2,4} matches "aa," "aaa," or "aaaa."
    - **Non-greedy** : a{2,4}? matches "aa," "aaa," or "aaaa" but prefers fewer occurrences.

We can make a quantifier non-greedy by adding a **?** immediately after it. This tells the regular expression engine to match the minimum number of occurrences and stop as soon as the next part of the pattern can be matched.

For example, consider the following PHP regular expression :

```
$pattern = '/<.*>/'; // Greedy
$patternNonGreedy = '/<.*?>/'; // Non-greedy

$input = '<div>example</div>';

preg_match($pattern, $input, $matches);
print_r($matches);      // Output: Array([0] => "<div>example</div>")

preg_match($patternNonGreedy, $input, $matches);
print_r($matches);      // Output: Array([0] => "<div>")
```

In the non-greedy version of the pattern, the regular expression engine matches as little as possible, resulting in a different match compared to the greedy version.

X] POSIX - Style Regular Expression :

POSIX-style regular expressions, also known as "Basic Regular Expressions" (BRE), are a type of regular expression syntax commonly used in various Unix utilities and programming languages that support POSIX standards. These regular expressions are generally simpler and less feature-rich compared to the more powerful "Extended

Regular Expressions" (ERE) found in many modern programming languages and tools.

Here's an explanation of some common elements in POSIX-style regular expressions along with examples:

1. **Literal Characters :** Most characters in a POSIX regular expression are treated as literal characters. For example, the regular expression /abc/ will match the string "abc."
2. **Anchors :** The ^ symbol is used to indicate the start of a line, and the $ symbol is used to indicate the end of a line. For example, ^abc matches lines that start with "abc," and xyz$ matches lines that end with "xyz."
   Example:
   - **^**The matches lines that start with "The."
   - **dog$** matches lines that end with "dog."
3. **Character Classes :** You can define character classes using [ ] brackets. Inside the brackets, you can specify a range of characters or individual characters.
   Example:
   - **[aeiou]** matches any single vowel.
   - **[0-9]** matches any single digit.
   - **[A-Za-z]** matches any single letter (uppercase or lowercase).
4. **Repetition:** You can specify the repetition of a character or character class using the {n} syntax, where n is a non-negative integer.
   Example:
   - a{3} matches "aaa."
   - [0-9]{2,4} matches 2 to 4 consecutive digits.
5. **Alternation :** The | symbol is used for alternation, allowing you to match one of several alternatives.
   Example:
   - cat|dog matches either "cat" or "dog."
6. **Escape Sequences:** To match special characters like *, ?, or . literally, you need to escape them using a backslash \.
   Example:
   - a*b matches "a*b."
7. **Grouping:** Parentheses ( ) are used for grouping to apply operators or alternation to a group of characters.
   Example:
   - (abc){2} matches "abcabc."
8. **Bracket Expression :** You can use [: :] notation to define certain character classes, such as [:digit:] for digits and [:alpha:] for alphabetic characters.
   Example:
   - **[[:digit:]]{3}** matches any three consecutive digits.

Here's an example of using POSIX-style regular expressions in a Unix utility like grep:

$ echo "The cat and the dog are friends." | grep 'cat\|dog'

In this example, we're using grep to search for lines that contain either "cat" or "dog." The | symbol is used for alternation, and \| is an escape sequence to match the | character literally.

Example of POSIX-style regular expressions, along with explanations for each part of the expression:

"/^([a-zA-Z0-9_.-]+)@([a-zA-Z0-9.-]+)\.([a-zA-Z]{2,5})$/"

This regular expression is designed to match email addresses. Let's break it down:

1. **^ and $:** These are anchors that indicate the start and end of the line. So, the entire regular expression will match the entire line, not just a part of it.
2. **([a-zA-Z0-9_.-]+) :** This part is a capturing group that matches the local part of the email address before the "@" symbol. It consists of:
    - **[a-zA-Z0-9_.-]+ :** A character class that matches one or more characters that are either letters (both uppercase and lowercase), digits, underscores, periods, or hyphens.
3. **@ :** This part matches the "@" symbol in the email address.
4. **([a-zA-Z0-9.-]+) :** This is another capturing group that matches the domain part of the email address. It consists of:
    - **[a-zA-Z0-9.-]+ :** A character class that matches one or more characters that are either letters (both uppercase and lowercase), digits, periods, or hyphens.
5. **\. :** This part matches the literal dot (.) that separates the domain name from the top-level domain (TLD).
6. **([a-zA-Z]{2,5}) :** This is a capturing group that matches the TLD, which consists of:
    - [a-zA-Z]{2,5}: A character class that matches between 2 and 5 letters, indicating common TLDs like "com," "org," "net," etc.

So, in summary, the regular expression matches strings that have the format of an email address, where there is a local part before the "@" symbol, a domain part after the "@" symbol, and a TLD separated by a dot.

Here's a shell command using grep to demonstrate the usage of this regular expression :

echo "user@example.com" | grep -E '^([a-zA-Z0-9_.-]+)@([a-zA-Z0-9.-]+)\.([a-zA-Z]{2,5})$'

In this command, we use echo to send the email address "user@example.com" to grep. The -E flag is used to enable extended regular expressions. The email address matches the pattern, so grep will output the email address.

POSIX-style regular expressions are more basic compared to Extended Regular Expressions (ERE) found in languages like Perl or Python, but they are still useful for

simple pattern matching tasks and are widely supported in Unix utilities and POSIX-compliant programming languages.

XI] Perl – Compatible Regular Expression :

Perl-Compatible Regular Expressions, often abbreviated as PCRE, are a powerful and widely used pattern matching and text processing tool. They are compatible with the regular expression features found in the Perl programming language, which is known for its flexible and expressive pattern matching capabilities. PCRE is not tied to Perl itself and can be used in many programming languages, including Python, PHP, and C/C++, among others.

Here are some key features and concepts of Perl-Compatible Regular Expressions (PCRE):

1. **Syntax:** PCRE patterns are written as strings that define a set of rules for matching text. These patterns can include a wide range of metacharacters, special sequences, and modifiers that enable complex and precise text matching.
2. **Metacharacters:** PCRE includes a variety of metacharacters that carry special meanings. For example:
    - `.` matches any character (except newline).
    - `*` matches zero or more of the preceding element.
    - `+` matches one or more of the preceding element.
    - `?` matches zero or one of the preceding element.
    - `|` functions as an OR operator.
    - `()` are used for grouping and capturing.
3. **Character classes:** PCRE supports character classes, which allow you to match any character from a specified set. For example, `[aeiou]` matches any vowel.
4. **Quantifiers:** PCRE supports various quantifiers to control the number of times a character or group of characters can appear. For example, `{3,5}` matches between 3 and 5 occurrences of the preceding element.
5. **Anchors:** Anchors like `^` (start of line) and `$` (end of line) are used to specify where in the text a match should occur.
6. **Backreferences:** PCRE allows you to refer to previously captured groups in your pattern, which can be useful for tasks like finding repeated words or phrases.
7. **Modifiers:** You can use modifiers to change the behavior of the regular expression. For example, `i` makes the pattern case-insensitive, and `s` allows `.` to match newline characters.
8. **Lookahead and lookbehind assertions:** PCRE supports positive and negative lookahead and lookbehind assertions, which allow you to define conditions that must be satisfied without consuming the characters in the match.

9. **Greedy and non-greedy matching:** PCRE provides the ability to specify whether a quantifier should be greedy (matches as much as possible) or non-greedy (matches as little as possible) by appending `?` to the quantifier.
10. **Escape sequences:** You can escape special characters with a backslash `\` to match them as literal characters. For example, `\.` matches a period.
11. **Subpatterns:** PCRE allows you to create subpatterns within your regular expression, which can be used for capturing specific portions of the matched text.

Here's an example of a PCRE regular expression in PHP :

```
$input = "The quick brown fox jumps over the lazy dog.";

// Match words containing 'fox' or 'dog'
$pattern = '/\b(?:fox|dog)\b/';
if (preg_match_all($pattern, $input, $matches)) {
    echo "Matches found:\n";
    foreach ($matches[0] as $match) {
        echo $match . "\n";
    }
} else {
    echo "No matches found.";
}
```

In this PHP example, we're using PCRE to match words that contain either "fox" or "dog" as whole words. Here's a breakdown of the regular expression :

- **\b:** Word boundary anchor. It ensures that we match whole words and not partial matches.
- **(?:fox|dog) :** A non-capturing group that matches either "fox" or "dog." The | symbol is used for alternation.
- **\b :** Another word boundary anchor to ensure we match whole words.

The regular expression is used with `preg_match_all`, which finds all non-overlapping matches in the input text. The matches are stored in the `$matches` array, and we iterate through them to print the found words containing "fox" or "dog."

PCRE regular expressions provide various advanced features beyond what's available in POSIX-style regular expressions, making them suitable for complex pattern matching and text processing tasks. These features include lookaheads, lookbehinds, named capture groups, and more, making them a popular choice for many programming languages and text-processing tools.

XII] Array :

In PHP, arrays are a versatile data structure used to store collections of values. There are several types of arrays, including indexed arrays, associative arrays, and multidimensional arrays. I'll provide examples for each type of array:

1. **Indexed Arrays:** Indexed arrays are the most common type of arrays in PHP, where elements are accessed by their numeric index.

   Example :

```
$fruits = array("apple", "banana", "cherry", "date");
echo $fruits[1];                    // Outputs  : "banana"
```

2. **Associative Arrays :** Associative arrays use named keys to access elements instead of numeric indices.

   Example:

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

 echo $person["name"];              // Outputs "John"
```

3. **Multidimensional Arrays :** Multidimensional arrays can store other arrays as elements. They are often used to represent more complex data structures.

   Example:

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

4. **Arrays with Mixed Data Types :** PHP allows you to store elements of different data types within the same array.

   Example:

```
$mixedArray = array("John", 30, true, 3.14);
echo $mixedArray[0];            // Outputs "John"
echo $mixedArray[1];            // Outputs 30
```

5. **Adding and Modifying Elements :** You can add or modify elements in an array using the array's index or key.

   Example :

   ```
   $fruits[] = "grape";           // Add an element to the end of the indexed array
   $person["name"] = "Jane"; // Modify the "name" key in the associative array
   ```

6. **Iterating Through Arrays :** You can use loops, such as foreach, to iterate through array elements.

   Example:

   ```
   foreach ($fruits as $fruit) {
       echo $fruit . ' ';
   }
   ```

7. **Finding the Length:** We can use the count function to determine the number of elements in an array.

   Example:

   ```
   $fruitCount = count($fruits); // Gets the number of elements in the indexed array
   ```

Arrays are fundamental data structures in PHP and are used for various purposes, such as storing data, iterating through data sets, and managing collections of items. The type of array you choose to use depends on your specific data needs and how you want to access and manipulate the elements within the array.

- ## Single dimensional Array :

In PHP, a single-dimensional array is a fundamental data structure that holds a collection of values or elements, indexed by numeric or string keys. Here's how you can create, access, and work with single-dimensional arrays in PHP:

1. **Creating Single-Dimensional Arrays:** We can create a single-dimensional array using the array() constructor or the shorthand [] syntax. For example:

   ```
   $fruits = array("apple", "banana", "cherry");
   ```

   OR

   ```
   $fruits = ["apple", "banana", "cherry"];
   ```

2. **Accessing Elements :** To access elements within a single-dimensional array, use the array's index, which starts at 0 for the first element. For example:

   ```
   echo $fruits[0]; // Outputs "apple"
   ```

3. **Iterating Over a Single-Dimensional Array:** We can use a foreach loop to iterate through the elements of a single-dimensional array:

   ```
   foreach ($fruits as $fruit) {
           echo $fruit . ' ';
   }
   ```

   This code will loop through the array and print each element.

4. **Adding and Modifying Elements:** To add or modify elements in a single-dimensional array, access the element using its index and assign a new value to it:

   ```
   $fruits[1] = "grape"; // Modify the second element
   $fruits[] = "orange"; // Add an element at the end
   ```

5. **Finding the Length:** We can find the number of elements in a single-dimensional array using the count function:

   ```
   $numFruits = count($fruits);
   ```

6. **Using Associative Single-Dimensional Arrays:** While traditional arrays are indexed with numeric keys, PHP also supports associative arrays, where you can use string keys to access elements. For example:

   ```
   $person = array(
      "name" => "John",
      "age" => 30
   );

   echo $person["name"]; // Outputs "John"
   ```

Single-dimensional arrays are one of the fundamental building blocks of data storage in PHP and are used extensively in various programming tasks, including storing lists of items, configuration settings, and more. They provide a straightforward way to manage and work with collections of data.

- ## Multidimensional Array :

In PHP, a multidimensional array is an array that contains other arrays (subarrays) as its elements. These subarrays can, in turn, contain more arrays, creating a

hierarchical structure of data. Multidimensional arrays are useful when we need to represent and work with data that has multiple levels or dimensions, such as tables, matrices, or records with multiple fields.

Here's how we can create, access, and work with multidimensional arrays in PHP :

1. **Creating Multidimensional Arrays :** We can create a multidimensional array by nesting arrays within arrays.

   For example :

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

2. **Accessing Elements :** To access elements within a multidimensional array, use multiple indices, separated by square brackets.
   For example:

```
echo $matrix[1][2];                    // Outputs 6
```

3. **Iterating Over a Multidimensional Array :** We can use nested loops to traverse through the elements of a multidimensional array:

```
foreach ($matrix as $row) {
    foreach ($row as $value) {
        echo $value . ' ';
    }
    echo "<br>";
}
```

   This code will iterate through the rows and columns of the matrix and print each element.

4. **Adding and Modifying Elements :** To add or modify elements in a multidimensional array, access the element using indices and assign a new value to it:

```
$matrix[0][1] = 99; // Modify an element
$matrix[2][] = 10;  // Add an element to the third row
```

5. **Finding the Length :** We can find the number of rows or columns in a multidimensional array using the count function:

```
$numRows = count($matrix);        // Number of rows
```

```
$numCols = count($matrix[0]);     // Number of columns in the first row
```

6. **Using Associative Multidimensional Arrays :** We can also create associative multidimensional arrays where each subarray is associated with a key.

For example :

```
$employees = array(
   array("name" => "John", "age" => 30),
   array("name" => "Jane", "age" => 25)
);

echo $employees[0]["name"];                    // Outputs "John"
```

Multidimensional arrays are especially useful for representing complex data structures like tables, matrices, hierarchical data, and records with multiple fields. They allow us to organize and work with data in a structured and flexible way in our PHP applications.

- ## Indexed Versus Associative Arrays :

In PHP, arrays can be categorized into two main types: indexed arrays and associative arrays. Each type of array has its own characteristics and use cases.

1. **Indexed Arrays :**
   - **Numeric Keys :** Indexed arrays are arrays where the keys are numeric and sequential, starting from zero. These keys are automatically assigned by PHP when you add elements to the array.
   - **Ordered:** Elements in indexed arrays are ordered and can be accessed by their numeric index.
   - **Example:**

```
$fruits = array("apple", "banana", "cherry");
echo $fruits[0];        // Outputs : "apple"
```

Indexed arrays are commonly used when you have a list of items, and the order of elements matters.

2. **Associative Arrays:**
   - **Named Keys:** Associative arrays are arrays where the keys are named, and they are associated with specific values. You define the keys, and they can be of any data type (usually strings or numbers).
   - **Unordered:** Elements in associative arrays are not necessarily in any specific order.
   - **Example:**

```
$person = array("first_name" => "John", "last_name" => "Doe", "age" => 30);
```

```
echo $person["first_name"];          // Outputs "John"
```

- Associative arrays are used when you need to store key-value pairs and want to access values by their associated keys.

Here's a comparison of indexed and associative arrays in PHP:

- **Indexed Array :**

```
$fruits = array("apple", "banana", "cherry");
```

- Indexed arrays use numeric keys, starting from 0.
- Access elements by their index, e.g., **$fruits[0]** to access "apple."
- Useful for lists of items where the order is important.

- **Associative Array:**

```
$person = array("first_name" => "John", "last_name" => "Doe", "age" => 30);
```

- Associative arrays use named keys (strings or numbers) that you define.
- Access elements by their key, e.g., **$person["first_name"]** to access "John."
- Useful for storing key-value pairs, such as data about a person.

It's worth noting that PHP allows you to mix indexed and associative keys within the same array. This type of array is referred to as a "mixed array," and it can store a combination of values with numeric and named keys. Here's an example :

```
$mixedArray = array("apple", "first_name" => "John", "banana", "last_name" => "Doe");
```

In this mixed array, "apple" and "banana" are stored with numeric keys (0 and 1), while "John" and "Doe" are stored with named keys ("first_name" and "last_name").

• Identifying Elements in an Indexed Arrays :

In PHP, we can identify elements of an array using their keys or indexes. The method we use depends on whether the array is an indexed array or an associative array.

1. Identifying Elements in an Indexed Array :

Indexed arrays have numeric keys, starting from 0, and you can access elements using their indexes.

```
$fruits = array("apple", "banana", "cherry");

echo $fruits[0];          // Outputs "apple"
echo $fruits[1];          // Outputs "banana"
echo $fruits[2];          // Outputs "cherry"
```

1. In this example, we access elements of the indexed array **$fruits** by their numerical index.

2. **Identifying Elements in an Associative Array :**
   Associative arrays have named keys, and you can access elements using those keys.

```
$person = array("first_name" => "John", "last_name" => "Doe", "age" => 30);

echo $person["first_name"];          // Outputs "John"
echo $person["last_name"];           // Outputs "Doe"
echo $person["age"];                 // Outputs 30
```

1. In this example, we access elements of the associative array **$person** by their named keys.
2. **Checking if an Element Exists :**
   To check if a specific key or index exists in an array, you can use functions like **isset()** or **array_key_exists().**

- Using **isset():**

```
$fruits = array("apple", "banana", "cherry");
if (isset($fruits[1])) {
    echo "Element at index 1 exists.";
} else {
    echo "Element at index 1 does not exist.";
}
```

- Using **array_key_exists()** for associative arrays:

```
$person = array("first_name" => "John", "last_name" => "Doe", "age" => 30);
if (array_key_exists("last_name", $person)) {
    echo "Key 'last_name' exists in the array.";
} else {
    echo "Key 'last_name' does not exist in the array.";
}
```

- **Looping Through Elements:**

To iterate through all elements in an array, you can use loops like **foreach**. This is especially useful when you don't know the keys or indexes in advance.

- **Looping through an indexed array :**

```
$fruits = array("apple", "banana", "cherry");
foreach ($fruits as $fruit) {
    echo $fruit . " ";
}
```

- **Looping through an associative array :**

```
$person = array("first_name" => "John", "last_name" => "Doe", "age" => 30);
foreach ($person as $key => $value) {
   echo $key . ": " . $value . " ";
}
```

In both cases, the loop allows you to access and process each element within the array, regardless of whether it's indexed or associative.

- ## Storing Data in Arrays :

In PHP, we can store data in arrays, which are versatile and flexible data structures for organizing and managing collections of values. There are two main types of arrays in PHP: indexed arrays and associative arrays. Here's how to store data in each type of array:

1. Indexed Arrays:

Indexed arrays use numeric indexes to store and access data. We can think of them as ordered lists. Here's how to create and store data in an indexed array:

```
// Creating an indexed array
$fruits = array("apple", "banana", "cherry", "date");

// Adding elements to the end of the array
$fruits[] = "fig";
$fruits[] = "grape";

// Modifying an element
$fruits[1] = "blueberry";

// Removing an element
unset($fruits[3]);

// Accessing elements
echo $fruits[0];          // Outputs "apple"
echo $fruits[1];          // Outputs "blueberry"
```

2. Associative Arrays:

Associative arrays use named keys to store and access data. Each key is associated with a specific value. Here's how to create and store data in an associative array:

```php
// Creating an associative array
$person = array(
    "first_name" => "John",
    "last_name" => "Doe",
    "age" => 30,
    "email" => "john@example.com"
);

// Adding elements
$person["city"] = "New York";

// Modifying an element
$person["age"] = 31;

// Removing an element
unset($person["email"]);

// Accessing elements
echo $person["first_name"];        // Outputs "John"
echo $person["age"];               // Outputs 31
```

3. Multidimensional Arrays:

We can also store arrays within arrays, creating multidimensional arrays. This is useful for organizing more complex data structures, such as tables or matrices.

```php
// Creating a multidimensional array
$products = array(
    array("product_name" => "Laptop", "price" => 1000),
    array("product_name" => "Smartphone", "price" => 500),
    array("product_name" => "Tablet", "price" => 300)
);

// Accessing elements in a multidimensional array
echo $products[0]["product_name"];      // Outputs "Laptop"
echo $products[1]["price"];             // Outputs 500
```

In PHP, we can use various array functions and control structures (e.g., loops) to work with arrays effectively. Arrays are a fundamental part of PHP and are used for a wide range of tasks, from storing and managing data to organizing information for further processing or presentation.

- ## Assigning a Range of Values :

If you want to create an array with a range of values in PHP, you can use a combination of **range()** and **array()** functions to achieve this. This approach allows you to create an array with a range of values quickly. Here's how you can do it:

```
// Create an array with a range of numeric values
$numericRange = range(1, 10);

// Create an array with a range of even numbers from 2 to 20
$evenNumbersRange = range(2, 20, 2);

// Create an array with a range of letters from 'a' to 'e'
$letterRange = range('a', 'e');

// Create an array with a range of dates from '2023-01-01' to '2023-01-05'
$dateRange = range('2023-01-01', '2023-01-05');

// Create an array with a range of uppercase letters from 'A' to 'Z'
$uppercaseLetterRange = range('A', 'Z');

// Create an array with a range of digits from '0' to '9'
$digitRange = range('0', '9');
```

In the above examples, we first use the range() function to create an array with the specified range of values. The function allows you to specify the start, end, and an optional step value (for numeric ranges). After creating the range, you can store it in a variable or directly use it in your code.

Remember that this approach is handy when you want to create an array with a sequence of values without manually specifying each element. The resulting array contains all the values within the specified range, making it suitable for various applications, such as creating numeric sequences, letters, dates, or other ordered sets of values.

- ## Padding an Arrays :

In PHP, you can pad an array with values to either increase or decrease its size. Padding an array means adding elements to it, either at the beginning, the end, or both, to reach a desired size. You might need to pad an array for various reasons, such as aligning data, ensuring a minimum size, or preparing it for specific operations. Here are some common approaches to padding an array :

1. Padding to Increase Array Size:

If we want to add elements to the beginning or end of an array to increase its size, you can use functions like **array_pad()** and **array_fill()**. Here's how to use them:

- **`array_pad()`**：Adds elements to the beginning or end of an array until it reaches the desired size.

```
$fruits = array("apple", "banana");
$paddedFruits = array_pad($fruits, 5, "cherry");
// Result : ["apple", "banana", "cherry", "cherry", "cherry"]
```

- **`array_fill()`**：Creates an array with a specified number of elements, all filled with the same value, and then you can merge it with your original array.

```
$fruits = array("apple", "banana");
$paddedFruits = array_merge(array_fill(0, 3, "cherry"), $fruits);
// Result: ["cherry", "cherry", "cherry", "apple", "banana"]
```

2. Padding to Decrease Array Size:

If we want to reduce the size of an array, you can use array_splice() to remove elements from it.

```
$fruits = array("apple", "banana", "cherry", "date");
array_splice($fruits, 2);
// Result: ["apple", "banana"]
```

In this example, we use **array_splice()** to remove elements from the array starting from index 2 (the third element) onward. You can adjust the parameters as needed to specify which elements to remove.

3. Padding to a Specific Size :

To pad an array to a specific size, you can combine the concepts of padding to increase or decrease the array's size. First, determine the difference between the desired size and the current size. Then, use **array_pad()** or **array_fill()** to add or remove elements accordingly.

```
$fruits = array("apple", "banana");
$desiredSize = 5;
if (count($fruits) < $desiredSize) {
    $paddedFruits = array_pad($fruits, $desiredSize, "cherry");
} elseif (count($fruits) > $desiredSize) {
    array_splice($fruits, $desiredSize);
}
```

In this example, if the array is smaller than the desired size, we pad it with "cherry" using **array_pad()**. If it's larger than the desired size, we trim it using **array_splice().**

These are some of the methods to pad arrays in PHP, allowing you to manipulate the array's size to match your specific needs.

- ## Slicing an Array :

Slicing an array in PHP involves extracting a portion of the original array, creating a new array containing the selected elements. You can specify the start and end indices of the slice to extract a range of elements. PHP provides several ways to slice an array:

1. Using array_slice():

The **array_slice()** function allows you to extract a portion of an array and create a new array with the selected elements. It takes the original array as its first argument, the start index as its second argument, and an optional length (number of elements to extract) as its third argument. If the third argument is omitted, it extracts elements from the start index to the end of the array.

```
$fruits = array("apple", "banana", "cherry", "date", "fig");

$slice = array_slice($fruits, 1, 3);
// Result: ["banana", "cherry", "date"]
```

2. Using array_splice() for In-place Slicing:

The **array_splice()** function allows you to extract a portion of an array in-place. It not only returns the extracted elements but also modifies the original array by removing the extracted elements.

```
$fruits = array("apple", "banana", "cherry", "date", "fig");

$slice = array_splice($fruits, 1, 3);
// Result: ["banana", "cherry", "date"]
// $fruits has been modified to ["apple", "fig"]
```

3. Slicing with the array_slice() Function with Negative Indices:

You can use negative indices with `array_slice()` to count elements from the end of the array. For example, to extract the last three elements of an array:

```
$fruits = array("apple", "banana", "cherry", "date", "fig");
$slice = array_slice($fruits, -3);
// Result: ["cherry", "date", "fig"]
```

4. Slicing Using Array Functions:

We can also use array functions like **array_slice()** in combination with others to achieve more complex slicing, such as extracting every other element or reversing the order of elements.

```
$fruits = array("apple", "banana", "cherry", "date", "fig");

// Extract every other element starting from the second element
$slice = array_slice($fruits, 1, count($fruits), true);

// Reverse the order of elements
$slice = array_reverse($fruits);
```

Slicing an array is a powerful operation that allows you to extract specific subsets of data from an array, facilitating various data manipulation tasks. Depending on your needs, you can choose the appropriate method for slicing arrays in PHP.

- ## Splitting an Array into Chunks :

In PHP, you can split an array into smaller chunks using the **array_chunk()** function. This function divides a given array into multiple arrays, each containing a specified number of elements. It's particularly useful when you want to process a large array in smaller segments or when you want to group data into manageable chunks. Here's how to use **array_chunk():**

```
$array = array("apple", "banana", "cherry", "date", "fig", "grape", "kiwi", "lemon");

// Split the array into chunks of 3 elements each
$chunks = array_chunk($array, 3);

// Print the resulting chunks
print_r($chunks);
```

The output will be an array of arrays, where each sub-array contains three elements :

```
Array
(
    [0] => Array
        (
            [0] => apple
            [1] => banana
            [2] => cherry
        )

    [1] => Array
        (
            [0] => date
```

```
            [1] => fig
            [2] => grape
        )

    [2] => Array
        (
            [0] => kiwi
            [1] => lemon
        )
)
```

In this example, the **array_chunk()** function takes the input array **$array** and splits it into chunks of three elements each. The resulting array of chunks is stored in the variable **$chunks**. We can adjust the second argument of **array_chunk()** to specify the desired chunk size.

Here's the syntax of the **array_chunk()** function:

array_chunk(array, size, preserve_keys);

- **array** : The input array to be split.
- **size** : The size of each chunk. This determines how many elements each sub-array will contain.
- **preserve_keys** (optional) : A boolean flag (default is **false**). If set to **true**, it preserves the original keys of the elements in the sub-arrays. If set to **false**, the sub-arrays are reindexed starting from 0.

For example, if you want to preserve keys, we can use:

```
$chunks = array_chunk($array, 3, true);
```

The **array_chunk()** function is a handy tool when you need to work with large datasets or when you want to process data in smaller, more manageable portions.

• Checking whether an Element Exists :

To check whether an element exists in an array in PHP, you can use a variety of methods and functions depending on your specific requirements. Here are some common approaches:

1. Using in_array() :

The **in_array()** function checks if a specific value exists within an array. It returns **true** if the value is found and **false** if it's not.

```
$fruits = array("apple", "banana", "cherry", "date");

if (in_array("banana", $fruits)) {
    echo "Found 'banana' in the array.";
} else {
    echo "'banana' not found in the array.";
}
```

2. Using array_search() :

The **array_search()** function finds the key associated with a specific value in an array. It returns the key if the value is found and **false** if it's not.

```
$fruits = array("apple", "banana", "cherry", "date");

$key = array_search("banana", $fruits);
if ($key !== false) {
    echo "Found 'banana' at index $key in the array.";
} else {
    echo "'banana' not found in the array.";
}
```

3. Using isset() or **array_key_exists() :**

If we want to check if a specific key exists in an associative array, you can use the **isset()** or **array_key_exists()** functions.

```
$person = array("first_name" => "John", "last_name" => "Doe", "age" => 30);

if (isset($person["age"])) {
    echo "Key 'age' exists in the array.";
} else {
    echo "Key 'age' does not exist in the array.";
}

// OR

if (array_key_exists("age", $person)) {
    echo "Key 'age' exists in the array.";
} else {
    echo "Key 'age' does not exist in the array.";
}
```

4. Using in_array() with Strict Comparison:

By default, **in_array()** performs a loose comparison (non-strict) when checking for values. If you want to perform a strict comparison, including data type, you can set the third parameter to **true**.

```
$numbers = array(1, 2, 3, "4");

if (in_array(4, $numbers, true)) {
    echo "Found the value 4 (strict comparison).";
} else {
    echo "Value 4 not found in the array.";
}
```

These methods allow we to check the existence of values or keys in both indexed and associative arrays. The choice of method depends on your specific use case and whether you need to check values, keys, or both in the array.

- ## Removing and Inserting Elements in an Array :

In PHP, you can remove and insert elements in an array using various functions and techniques. Here are some common methods for removing and inserting elements in an array:

Removing Elements :

1. **Using unset() for Indexed Arrays :**
   To remove a specific element from an indexed array, you can use the **unset()** function.

   ```
   $fruits = array("apple", "banana", "cherry", "date");
   unset($fruits[1]);                    // Removes "banana"
   ```

   After the above code, the array will look like **["apple", "cherry", "date"]**.

2. **Using unset() for Associative Arrays:**
   To remove a specific key-value pair from an associative array, use `unset()` with the key.

   ```
   $person = array("first_name" => "John", "last_name" => "Doe", "age" => 30);
   unset($person["age"]);  // Removes the "age" key and its associated value
   ```

   After the above code, the array will look like **["first_name" => "John", "last_name" => "Doe"].**

3. **Using array_splice() :**
   The **array_splice()** function allows you to remove elements from an array while preserving its keys. You can specify the start index and the number of elements to remove.

```
$fruits = array("apple", "banana", "cherry", "date");
array_splice($fruits, 1, 2);        // Removes "banana" and "cherry"
```

   After the above code, the array will look like **["apple", "date"].**

- ## Inserting Elements:

1. **Using Index for Indexed Arrays:**
   We can use specific indices to insert elements into an indexed array.

```
$fruits = array("apple", "cherry");
$fruits[1] = "banana";      // Inserts "banana" at index 1
```

   After the above code, the array will look like **["apple", "banana"].**

2. **Using Key-Value Pairs for Associative Arrays:**
   To insert a key-value pair into an associative array, simply assign the value to the desired key.

```
$person = array("first_name" => "John", "last_name" => "Doe");
$person["age"] = 30;        // Inserts "age" => 30
```

   After the above code, the array will look like **["first_name" => "John", "last_name" => "Doe", "age" => 30].**

3. **Using array_splice() to Insert Elements:**
   We can also use **array_splice()** to insert elements at a specific index in an array.

```
$fruits = array("apple", "date");
array_splice($fruits, 1, 0, "banana", "cherry");
// Inserts "banana" and "cherry" at index
```

   After the above code, the array will look like **["apple", "banana", "cherry", "date"].**

These methods allow we to remove and insert elements in both indexed and associative arrays, depending on your specific requirements.

- ## Converting between Arrays and Variables :

In PHP, you can convert between arrays and variables, depending on your specific needs. Here are some common methods to convert between the two:

1. Converting Variables to Arrays :

We can convert a variable or a set of variables into an array using various methods. For example:

- **Using the `array()` Function:**
  We can use the **array()** function to create an array and assign values to it.

```
$var1 = "apple";
$var2 = "banana";

$array = array($var1, $var2);
```

Using Square Brackets :
We can create an indexed array by enclosing variables in square brackets.

```
$var1 = "apple";
$var2 = "banana";

$array = [$var1, $var2];
```

Using Associative Arrays :
We can create an associative array by specifying keys.

```
$name = "John";
$age = 30;

$person = ["name" => $name, "age" => $age];
```

2. Converting Arrays to Variables :

We can convert an array to variables by extracting its values and assigning them to individual variables. Here's how you can do it:

- **Using List():**
  The **`list()`** construct allows you to assign array values to individual variables. This works for indexed arrays.

```
$fruits = ["apple", "banana", "cherry"];
list($first, $second, $third) = $fruits;
```

Using Array Destructuring (PHP 7.1+) :

Array destructuring is a shorthand way to assign array values to variables. It works for both indexed and associative arrays.

```
$fruits = ["apple", "banana", "cherry"];
[$first, $second, $third] = $fruits;
```

For associative arrays :

```
$person = ["name" => "John", "age" => 30];
["name" => $name, "age" => $age] = $person;
```

Please note that converting between arrays and variables is most useful when you have specific use cases, such as unpacking the values from an array to work with them as individual variables or combining variables into an array for structured data storage. The choice of method depends on your specific requirements and the context in which you are working.

- ## Creating Variables from Array :

In PHP, you can create variables from an array using the **extract()** function. This function allows you to turn the keys in an associative array into variable names and their corresponding values into variable values. Here's how you can use **extract()** to create variables from an array :

```
$data = array(
   "name" => "John",
   "age" => 30,
   "country" => "USA"
);

extract($data);

echo $name;          // Outputs "John"
echo $age;           // Outputs 30
echo $country;       // Outputs "USA"
```

In this example, the **extract()** function takes the associative array **$data** and creates variables with the same names as the keys in the array, assigning their respective values to those variables. As a result, you can access the array values as if they were individual variables.

It's important to be cautious when using **extract()** because it can introduce variables into your code dynamically, which may lead to unexpected behaviour and potential security risks, especially if the array contains user input. To mitigate these risks,

consider using the optional parameters of **extract()** to specify the behaviour and scope:

Use the second parameter to specify the behaviour when a variable with the same name already exists. For example:

```
extract($data, EXTR_SKIP);
```

Use the third parameter to specify the scope in which the variables will be created. For example :

```
$scope = ["my_function"];
extract($data, EXTR_PREFIX_ALL, "my_function");
```

By specifying the behaviour and scope, we can control how variables are created from the array, making it safer and more predictable in your code.

- # Traversing Arrays :

Traversing, or iterating through, arrays is a common operation in PHP, and there are several ways to do it, depending on the type of array and the specific requirements of your code. Here are some common methods for traversing arrays in PHP:

1. Using foreach Loop:

The **foreach** loop is a versatile way to iterate through both indexed and associative arrays. It automatically handles the iteration and provides access to the elements of the array.

- **Iterating Through an Indexed Array :**

```
$fruits = array("apple", "banana", "cherry");
foreach ($fruits as $fruit) {
    echo $fruit . "\n";
}
```

- **Iterating Through an Associative Array :**

```
$person = array("first_name" => "John", "last_name" => "Doe", "age" => 30);
foreach ($person as $key => $value) {
    echo $key . ": " . $value . "\n";
}
```

2. Using for Loop for Indexed Arrays:

We can use a **for** loop to iterate through indexed arrays when you need to work with the numerical indices explicitly.

```
$fruits = array("apple", "banana", "cherry")
for ($i = 0; $i < count($fruits); $i++) {
    echo $fruits[$i] . "\n";
}
```

3. Using while Loop:

A **while** loop can be used when we want more control over the iteration process. We need to manually manage the iteration variable.

```
$fruits = array("apple", "banana", "cherry");
$count = count($fruits);
$i = 0;

while ($i < $count) {
    echo $fruits[$i] . "\n";
    $i++;
}
```

4. Using Array Functions:

We can use array functions like **array_walk()** or **array_map()** for more complex operations during traversal.

- **Using array_walk() :**

array_walk() allows we to apply a user-defined function to each element of an array.

```
$fruits = array("apple", "banana", "cherry");
function printFruit($fruit) {
    echo $fruit . "\n";
}
array_walk($fruits, "printFruit");
```

- **Using array_map() :**

array_map() applies a given function to each element of one or more arrays and returns the results.

```
$fruits = array("apple", "banana", "cherry");
function addEmoji($fruit) {
    return $fruit . " ";
}
$fruitsWithEmoji = array_map("addEmoji", $fruits);
```

5. Using Recursive Traversal :

If we have nested or multidimensional arrays, we may need to use recursive functions or custom logic to traverse them effectively.

These methods allow we to traverse and iterate through arrays in PHP, and we can choose the one that best suits your specific needs and the type of array we are working with.

- ## Sorting Arrays :

In PHP, you can sort arrays using various sorting functions, depending on your requirements and the type of array you're working with. PHP provides both ascending and descending sorting options for indexed and associative arrays. Here are some common methods for sorting arrays:

1. Sorting Indexed Arrays:

- **Sorting Indexed Arrays in Ascending Order with sort()** or **asort() :**

  sort() : Sorts an indexed array in ascending order based on the values, re-indexing the array numerically.

  ```
  $fruits = array("banana", "cherry", "apple");
  sort($fruits);
  ```

  asort(): Sorts an indexed array in ascending order based on the values, preserving the original keys.

  ```
  $fruits = array("banana", "cherry", "apple");
  asort($fruits);
  ```

- **Sorting Indexed Arrays in Descending Order with rsort() or arsort() :**

  rsort() : Sorts an indexed array in descending order based on the values, re-indexing the array numerically.

```
$fruits = array("banana", "cherry", "apple");
rsort($fruits);
```

arsort(): Sorts an indexed array in descending order based on the values, preserving the original keys.

```
$fruits = array("banana", "cherry", "apple");
arsort($fruits);
```

2 Sorting Associative Arrays:

- **Sorting Associative Arrays in Ascending Order with ksort()** or **asort():**

  ksort(): Sorts an associative array in ascending order based on the keys.

  ```
  $person = array("first_name" => "John", "last_name" => "Doe", "age" => 30);
  ksort($person);
  ```

  asort(): Sorts an associative array in ascending order based on the values.

  ```
  $person = array("first_name" => "John", "last_name" => "Doe", "age" => 30);
  asort($person);
  ```

- **Sorting Associative Arrays in Descending Order with krsort()** or **arsort():**

  krsort() : Sorts an associative array in descending order based on the keys.

  ```
  $person = array("first_name" => "John", "last_name" => "Doe", "age" => 30);
  krsort($person);
  ```

  arsort() : Sorts an associative array in descending order based on the values.

  ```
  $person = array("first_name" => "John", "last_name" => "Doe", "age" => 30);
  arsort($person);
  ```

3. Custom Sorting with usort() for Indexed Arrays :

We can also sort indexed arrays using a custom comparison function with **usort()**.

```
$fruits = array("banana", "cherry", "apple");
usort($fruits, function($a, $b) {
return strcmp($a, $b);
});
```

This example sorts the array $fruits alphabetically using a custom comparison function.

These methods allow we to sort arrays in PHP based on values or keys in both ascending and descending order, and we can choose the appropriate method based on your specific needs.

- ## Acting on Entire Arrays :

In PHP, you can perform various operations on entire arrays to manipulate their elements or gather information about the array as a whole. Here are some common actions you can take on arrays:

1. Concatenating Arrays :

We can combine two or more arrays into a single array using the **array_merge()** function.

```
$fruits1 = array("apple", "banana", "cherry");
$fruits2 = array("date", "fig", "grape");

$combinedFruits = array_merge($fruits1, $fruits2);
```

2. Reversing an Array:

We can reverse the order of elements in an array using the **array_reverse()** function.

```
$fruits = array("apple", "banana", "cherry");
$reversedFruits = array_reverse($fruits);
```

3. Flipping an Associative Array :

We can flip an associative array, swapping keys and values, using the **array_flip()** function.

```
$person = array("first_name" => "John", "last_name" => "Doe", "age" => 30);
$flippedPerson = array_flip($person);
```

4. Combining Keys and Values:

We can combine two arrays into an associative array, with one array providing keys and the other providing values, using the **array_combine()** function.

```
$keys = array("apple", "banana", "cherry");
$values = array(3, 5, 2);
$combinedArray = array_combine($keys, $values);
```

5. Removing Duplicates :

We can remove duplicate values from an array using the **array_unique()** function.

```
$numbers = array(3, 5, 3, 7, 5);
$uniqueNumbers = array_unique($numbers);
```

6. Checking for Array Equality :

We can compare two arrays for equality (having the same keys and values in the same order) using the **==** operator or the **array_diff_assoc()** function.

```
$array1 = array("apple", "banana", "cherry");
$array2 = array("apple", "banana", "cherry");

$equal = ($array1 == $array2);          // true
```

7. Checking if an Array Contains a Value :

We can check if an array contains a specific value using functions like **in_array()**.

```
$fruits = array("apple", "banana", "cherry");
$containsBanana = in_array("banana", $fruits);        // true
```

8. Counting Array Elements :

We can count the number of elements in an array using the **count()** function.

```
$fruits = array("apple", "banana", "cherry");
$numberOfFruits = count($fruits);        // 3
```

9. Determining if an Array is Empty:

We can check if an array is empty using functions like **empty()** or by comparing its count to zero.

```
$fruits = array();
$isNotEmpty = !empty($fruits);           // false
```

These are some common actions we can perform on entire arrays in PHP. Depending on your specific needs, we can use these functions to manipulate and work with arrays effectively.

- ## Using Arrays :

In PHP, arrays can be used to implement both sets and stacks, which are common data structures used for different purposes. Here's how you can use arrays to create sets and stacks in PHP:

1. Sets using Arrays :

A set is a collection of unique elements, and you can use an array to implement a set in PHP because arrays naturally enforce uniqueness of keys. To create a set, you can add elements to an array, ensuring that each element is unique. Here's an example:

```
$set = array();

// Adding elements to the set
$set["apple"] = true;
$set["banana"] = true;
$set["cherry"] = true;

// Checking if an element is in the set
if (isset($set["banana"])) {
    echo "Banana is in the set.";
} else {
    echo "Banana is not in the set.";
}

// Adding another element (ensuring uniqueness)
$set["banana"] = true;

// Now, the set still contains only unique elements
```

In this example, the **$set** is implemented as an associative array where the keys represent the unique elements. When we add an element, we set its value to true, ensuring uniqueness. We can then use **isset()** to check for the presence of an element in the set.

2. Stacks using Arrays :

A stack is a data structure that follows the Last-In, First-Out (LIFO) principle. You can use an indexed array in PHP to implement a stack. Here's an example of how to create and use a stack :

```
$stack = array();

// Push elements onto the stack
array_push($stack, "apple");
array_push($stack, "banana");
array_push($stack, "cherry");

// Pop elements from the stack
$topElement = array_pop($stack);
```

In this example, the $stack is an indexed array. You can push elements onto the stack using the **array_push()** function, and pop elements from the stack using the **array_pop()** function. The last element pushed onto the stack will be the first one to be popped, following the LIFO principle.

These are some basic examples of using arrays in PHP to implement sets and stacks. Depending on your specific use case, you can build more complex data structures or implement other operations like unions, intersections, and more for sets, or additional stack operations for stacks.
