Unit II: Array and Functions

Weightage: 25%
Lectures: 7
Practicals: 14

Array:

Types of Array:

1) Numeric array,
2) Associated array,
3) Multi-dimensional Arrays.

What is an Array in PHP?

In PHP, an array is a special variable that allows you to store multiple values in a single variable. Instead of declaring multiple individual variables to hold each value, an array lets you store all values under one variable name, making it easier to manage and work with large datasets.

Arrays are essential in programming as they allow you to group and organize data efficiently. They can store data of different types and sizes, and can be accessed by using either numerical indexes or associative keys.

Characteristics of Arrays in PHP:

1. Single Variable to Store Multiple Values: With arrays, you can store several values in a single variable.
2. Indexed or Associative: The elements of an array can be indexed numerically (starting from 0) or associated with custom keys (strings or integers).
3. Can Store Different Data Types: Arrays can hold mixed data types, including strings, numbers, objects, or even other arrays.

Types of Array:

Numeric Array: Indexed with numbers automatically.
Associative Array: Indexed with custom keys (strings or integers).
Multi-dimensional Array: Contains arrays inside other arrays, useful for matrix-like data.

1) Numeric Array

A numeric array is an array where the keys are automatically assigned by PHP as integers starting from 0.

Syntax:

```
$array_name = array(value1, value2, value3, ...);
```

Example:

```
<?php
// Defining a numeric array
$colors = array("Red", "Green", "Blue", "Yellow");

// Accessing elements
echo $colors[0]; // Output: Red
echo $colors[1]; // Output: Green
?>
```

Output:

```
RedGreen
```

2) Associative Array

An associative array is an array where each element is associated with a custom key. The keys can be strings or integers.

Syntax:

```
$array_name = array("key1" => "value1", "key2" => "value2", ...);
```

Example:

```
<?php
// Defining an associative array
$person = array(
    "name" => "John",
    "age" => 25,
    "city" => "New York"
);

// Accessing elements using keys
echo "Name: " . $person["name"] . "<br>"; // Output: Name: John
echo "Age: " . $person["age"] . "<br>";   // Output: Age: 25
echo "City: " . $person["city"] . "<br>"; // Output: City: New York
?>
```

Output:

```
Name: John
```

```
Age: 25
City: New York
```

3) Multi-dimensional Arrays

A multi-dimensional array is an array that contains other arrays. It can be visualized as an array of arrays. This can be used to store data in a matrix or table format.

Syntax:

```
$array_name = array(
   array(value1, value2, ...),
   array(value3, value4, ...)
);
```

Example:

```
<?php
// Defining a multi-dimensional array
$students = array(
   array("name" => "Alice", "age" => 22, "course" => "Math"),->0
   array("name" => "Bob", "age" => 23, "course" => "Physics"),->1
   array("name" => "Charlie", "age" => 24, "course" => "Chemistry")->2
);

// Accessing elements
echo $students[0]["name"] . " is " . $students[0]["age"] . " years old and studies " .
$students[0]["course"] . ".<br>";
echo $students[1]["name"] . " is " . $students[1]["age"] . " years old and studies " .
$students[1]["course"] . ".<br>";
echo $students[2]["name"] . " is " . $students[2]["age"] . " years old and studies " .
$students[2]["course"] . ".<br>";
?>
```

Output:

```
Alice is 22 years old and studies Math.
Bob is 23 years old and studies Physics.
Charlie is 24 years old and studies Chemistry.
```

User Defined Functions:

- Types of Function,
- Return statement,
- How to call a function,
- Function without parameters,
- Function with parameters,
- default argument function,
- variable length argument function.

User-Defined Functions in PHP

- In PHP, a user-defined function (UDF) is a function that you define by yourself using the function keyword.
- Functions help in organizing code by dividing it into smaller, reusable chunks.
- Functions can accept arguments, return values, and be called multiple times throughout the code.

1) Types of Functions in PHP

PHP functions can be categorized into different types, depending on how they are defined and used. The main types of functions are:

1. **Function without Parameters**: A function that does not accept any parameters.
2. **Function with Parameters**: A function that accepts input values through parameters.
3. **Default Argument Function**: A function that has default values for its parameters if no argument is passed.
4. **Variable Length Argument Function**: A function that accepts a variable number of arguments.

---

2) Return Statement

- The **return** statement is used to return a value from a function to the caller. After the return statement is executed, the function terminates, and the returned value can be stored in a variable or used directly.

  Syntax:

  ```
  return value;
  ```

---

3) How to Call a Function

To call a function in PHP, you simply use the function name followed by parentheses ().

- If the function requires parameters, you pass them inside the parentheses.
- If the function doesn't require parameters, you call it with empty parentheses.

Syntax:

```
function_name();
```

---

4) Function without Parameters

A **function without parameters** does not take any input from the caller. It performs an operation but doesn't accept any values.

Example:

```
<?php
// Function without parameters
function greet() {
   echo "Hello, welcome to PHP!<br>";
}

// Calling the function
greet();  // Output: Hello, welcome to PHP!
?>
```

---

5) Function with Parameters

A **function with parameters** accepts input values when called. These parameters allow you to pass data to the function, which can then be used within its body.

Syntax:

```
function function_name($parameter1, $parameter2) {
   // Code
}
```

Example:

```
<?php
// Function with parameters
function addNumbers($a, $b) {
    $sum = $a + $b;
    echo "The sum is: $sum<br>";
}

// Calling the function
addNumbers(5, 10);  // Output: The sum is: 15
?>
```

---

6) Default Argument Function

A **default argument function** allows you to define default values for the parameters. If the caller does not provide a value for a parameter, the default value will be used.

Syntax:

```
function function_name($param1 = default_value) {
    // Code
}
```

Example:

```
<?php
// Default argument function
function greet($name = "Guest") {
    echo "Hello, $name!<br>";
}

// Calling the function without an argument
greet();  // Output: Hello, Guest!

// Calling the function with an argument
greet("John");  // Output: Hello, John!
?>
```

---

7) Variable Length Argument Function

A **variable length argument function** allows you to pass a variable number of arguments to a function. You can use the func_num_args() function to get the number of arguments passed, and func_get_arg() to access individual arguments.

Syntax:

```
function function_name(...$args) {
   // Code
}
```

Example:

```
<?php
// Variable length argument function
function sumNumbers(...$numbers) {
   $sum = 0;
   foreach ($numbers as $num) {
      $sum += $num;
   }
   echo "The total sum is: $sum<br>";
}

// Calling the function with different numbers of arguments
sumNumbers(1, 2, 3);        // Output: The total sum is: 6
sumNumbers(5, 10, 15, 20);  // Output: The total sum is: 50
?>
```

1. **Static Variable,**
2. **Global Variable,**
3. **Difference between Call by Value and Call by Reference.**

---

1. Static Variable in PHP

A **static variable** in PHP is a special type of local variable that keeps its value between multiple calls to the function in which it is declared. Here's a detailed explanation in simple points:

---

1. What is a Static Variable?

- A static variable is **local to a function** but unlike normal local variables, it does not lose its value after the function ends.
- It retains its value between successive function calls.

---

2. How is it Declared?

- Declared using the **static keyword** inside a function.
- It is initialized only once, the first time the function is called.

---

3. Key Characteristics

- **Persistence**: The variable persists in memory even after the function exits.
- **Scope**: It is accessible only within the function where it is declared.
- **Initialization**: The initialization happens only once, and the value is retained across calls.

---

4. Why Use Static Variables?

- **Retaining State Information**: Ideal for tasks where you need to keep track of data across multiple function calls (e.g., counters, logs).
- **Efficiency**: Reduces the need for global variables, keeping the variable scope restricted to the function.

---

5. Example Explained

Here is a step-by-step explanation of the code:

Code:

```
<?php
function counter() {
   static $count = 0; // Static variable retains its value
   $count++;
   echo "The counter value is: $count<br>";
}

counter(); // Output: The counter value is: 1
counter(); // Output: The counter value is: 2
counter(); // Output: The counter value is: 3
?>
```

How it Works:

1. **First Call**:
    - The function is called for the first time.
    - $count is initialized to 0.
    - $count++ increments the value to 1.
    - Output: The counter value is: 1.
2. **Second Call**:
    - The function is called again.
    - The static variable $count retains its value from the previous call (1).
    - $count++ increments the value to 2.
    - Output: The counter value is: 2.
3. **Third Call**:
    - The function is called for the third time.
    - The static variable $count retains its value from the previous call (2).
    - $count++ increments the value to 3.
    - Output: The counter value is: 3.

---

6. Important Notes

- Static variables are **different from global variables** because they are limited to the function where they are declared.
- They are initialized only **once** and retain their value between function calls.
- They are useful for functions that need to **remember past data** across calls.

2. Global Variable in PHP - Detailed Explanation

A **global variable** in PHP is a variable that is declared outside of any function, class, or block and can be accessed globally throughout the script. However, when used inside a function, special handling is required. Here's a detailed explanation:

---

1. What is a Global Variable?

- A global variable is declared **outside of any function or class**.
- By default, it has a global scope, meaning it can be accessed anywhere in the script **except inside functions or methods directly**.

---

2. Accessing Global Variables Inside Functions

- Global variables cannot be directly accessed inside functions due to PHP's scope rules.
- **To use a global variable inside a function, you must:**
  1. Use the **global keyword** to import the global variable into the local scope of the function.
  2. Alternatively, use the **$GLOBALS array**, a superglobal array in PHP that stores all global variables.

---

3. Declaring a Global Variable

- Declare it outside any function or class.
- Example:

```
$x = 10; // Global variable
```

---

4. Accessing a Global Variable

Using the global Keyword:

- The global keyword imports the variable into the function's local scope.

Using the $GLOBALS Array:

- Access the global variable by using $GLOBALS['variable_name'].

---

5. Example of Global Variables

Using the global Keyword:

```
<?php
$x = 10; // Global variable

function displayGlobal() {
   global $x; // Access the global variable
   echo "The global variable x is: $x<br>";
}

displayGlobal(); // Output: The global variable x is: 10
?>
```

Explanation:

1. The variable $x is declared globally.

2. Inside the function displayGlobal, the global keyword imports $x into the function's scope.
3. The function then accesses and displays the value of $x.

Using the $GLOBALS Array:

```
<?php
$x = 10; // Global variable

function displayGlobal() {
    echo "The global variable x is: " . $GLOBALS['x'] . "<br>"; // Access global variable using $GLOBALS
}

displayGlobal(); // Output: The global variable x is: 10
?>
```

Explanation:

1. The $GLOBALS array is used to access the global variable $x inside the function.
2. $GLOBALS['x'] directly refers to the global variable $x.

---

6. Key Characteristics of Global Variables

- **Global Scope**: Declared outside of functions and available globally throughout the script.
- **Restricted Inside Functions**: Can only be accessed inside a function using global or $GLOBALS.
- **Useful for Sharing Data**: Ideal for sharing data between functions.

---

7. Important Notes

- Overusing global variables can make code harder to debug and maintain. It's generally better to use function parameters and return values when possible.
- Be cautious with naming to avoid variable conflicts.

---

8. Practical Use Case

Global variables are often used to:

- Store configuration data that needs to be accessed across multiple functions.
- Share data between functions without explicitly passing it as a parameter.

Summary

1. A **global variable** is declared outside any function or class.
2. To access it inside a function:
    - Use the global keyword.
    - Or, use the $GLOBALS array.

Difference Between Call by Value and Call by Reference in PHP

When passing arguments to a function in PHP, you can pass them by **value** or by **reference**, which determines how the variable's value is treated inside the function. Here's a detailed comparison:

---

1. Comparison Table

Feature	Call by Value	Call by Reference
Definition	A copy of the variable's value is passed to the function.	A reference to the actual variable is passed to the function.
Changes to Original Variable	Changes made inside the function do not affect the original variable.	Changes made inside the function affect the original variable.
Syntax	Default behavior of PHP.	Use the & symbol before the parameter name in the function definition.
Performance	Less efficient for large data structures because it involves copying the values.	More efficient as it avoids copying data.
Use Case	When you do not want the function to modify the original variable.	When you want the function to modify the original variable.

Examples

1) **Call by Value Example**
   - **Code**:

```
<?php
function incrementValue($num) {
    $num += 10; // Changes the local copy only
    echo "Inside function: $num<br>";
}

$number = 5;
incrementValue($number); // Inside function: 15
echo "Outside function: $number<br>"; // Outside function: 5
?>
```

   - **Explanation**:
     1. The function incrementValue receives a **copy** of the value of $number.
     2. The local variable $num is modified inside the function, but this does not affect the original variable $number.
     3. Output:
        - Inside function: 15
        - Outside function: 5

---

2) Call by Reference Example

- **Code**:

```
<?php
function incrementReference(&$num) {
    $num += 10; // Changes the original variable
    echo "Inside function: $num<br>";
}

$number = 5;
incrementReference($number); // Inside function: 15
echo "Outside function: $number<br>"; // Outside function: 15
?>
```

- **Explanation**:
    1. The function incrementReference receives a **reference** to the variable $number.
    2. Changes made to $num inside the function directly affect the original variable $number.
    3. Output:
        - Inside function: 15
        - Outside function: 15

---

3. Key Points to Remember

- **Call by Value**:
    - The default behavior in PHP.
    - Use when you want the function to work on a copy and leave the original variable unchanged.
- **Call by Reference**:
    - Requires the & symbol before the parameter name in the function definition.
    - Use when you want the function to modify the original variable or save memory with large data structures.

---

4. Practical Use Cases

- **Call by Value**:
    - Suitable for calculations where the original data should remain unchanged.
    - Example: Calculating a discount without modifying the original price.
- **Call by Reference**:
    - Useful when you need the function to update the original variable.
    - Example: Updating user data or manipulating arrays within a function.

Summary

- **Call by Value** creates a new copy of the variable, keeping the original unchanged.
- **Call by Reference** passes the actual variable, allowing the function to modify its value directly.

Built-in Functions in PHP

PHP provides a rich set of built-in functions to perform various operations. Here, we'll cover **String Functions**, **Math Functions**, and **Array Functions** with syntax, examples, and outputs.

String Functions

PHP string functions help manipulate and work with strings. Below is a list of commonly used string functions.

List of String Functions:

1. strlen() - Returns the length of a string.
2. strtolower() - Converts a string to lowercase.
3. strtoupper() - Converts a string to uppercase.
4. strpos() - Finds the position of the first occurrence of a substring.
5. str_replace() - Replaces all occurrences of a search string with a replacement string.
6. substr() - Extracts a part of a string.
7. trim() - Removes whitespace or specified characters from the beginning and end of a string.
8. explode() - Splits a string into an array by a delimiter.
9. implode() - Joins array elements into a string.
10. strrev() - Reverses a string.

String Functions in PHP - Detailed Explanation

PHP provides a rich set of string functions for working with and manipulating strings. Below is a detailed explanation of commonly used string functions.

---

1. strlen()

- **Description**: Returns the length of a string.
- **Syntax**: strlen(string)
- **Use Case**: Useful for validating the length of inputs (e.g., passwords or usernames).

Example:

```
<?php
$str = "Hello, PHP!";
echo "Length of the string: " . strlen($str);
?>
```

Output:

```
Length of the string: 11
```

---

2. strtolower()

- **Description**: Converts a string to lowercase.
- **Syntax**: strtolower(string)
- **Use Case**: Useful for case-insensitive comparisons or formatting.

Example:

```
<?php
```

```
$str = "Hello, PHP!";
echo "Lowercase string: " . strtolower($str);
?>
```

Output:

```
Lowercase string: hello, php!
```

---

3. strtoupper()

- **Description**: Converts a string to uppercase.
- **Syntax**: strtoupper(string)
- **Use Case**: Useful for formatting or emphasis.

Example:

```
<?php
$str = "Hello, PHP!";
echo "Uppercase string: " . strtoupper($str);
?>
```

Output:

```
Uppercase string: HELLO, PHP!
```

---

4. strpos()

- **Description**: Finds the position of the first occurrence of a substring within a string.
- **Syntax**: strpos(haystack, needle, [offset])
- **Parameters**:
  - haystack: The main string.
  - needle: The substring to search for.
  - offset (optional): The position to start searching.
- **Use Case**: Useful for searching and validating substrings.

Example:

```
<?php
$str = "I love PHP!";
echo "Position of 'PHP': " . strpos($str, "PHP");
?>
```

Output:

```
Position of 'PHP': 7
```

---

5. str_replace()

- **Description**: Replaces all occurrences of a search string with a replacement string.
- **Syntax**: str_replace(search, replace, subject, [count])
- **Parameters**:
  - search: The string to be replaced.
  - replace: The replacement string.
  - subject: The string being searched.
  - count (optional): The number of replacements made.
- **Use Case**: Useful for modifying text dynamically.

Example:

```
<?php
$text = "I love PHP!";
echo str_replace("PHP", "coding", $text);
?>
```

Output:

```
I love coding!
```

---

6. substr()

- **Description**: Extracts a part of a string.
- **Syntax**: substr(string, start, [length])
- **Parameters**:
  - start: The starting position (0-based).
  - length (optional): The number of characters to extract.
- **Use Case**: Useful for truncating or extracting parts of strings.

Example:

```
<?php
$str = "Hello, PHP!";
echo substr($str, 7, 3); // Extracts "PHP"
?>
```

Output:

```
PHP
```

---

7. trim()

- **Description**: Removes whitespace or specified characters from the beginning and end of a string.
- **Syntax**: trim(string, [characters])
- **Use Case**: Useful for cleaning user input.

Example:

```
<?php
$str = "   Hello, PHP!   ";
echo "Trimmed string: '" . trim($str) . "'";
?>
```

Output:

```
Trimmed string: 'Hello, PHP!'
```

---

8. explode()

- **Description**: Splits a string into an array using a delimiter.
- **Syntax**: explode(delimiter, string, [limit])
- **Parameters**:
  - delimiter: The string used to split.
  - limit (optional): Maximum number of elements in the output array.
- **Use Case**: Useful for breaking down data into manageable pieces.

Example:

```
<?php
$data = "red,green,blue";
$colors = explode(",", $data);
```

```
print_r($colors);
?>
```

Output:

css

Copy code

```
Array ( [0] => red [1] => green [2] => blue )
```

---

9. implode()

- **Description**: Joins array elements into a single string using a glue string.
- **Syntax**: implode(glue, array)
- **Use Case**: Useful for creating strings from array data.

Example:

```
<?php
$colors = ["red", "green", "blue"];
echo implode(", ", $colors);
?>
```

Output:

```
red, green, blue
```

---

10. strrev()

- **Description**: Reverses a string.
- **Syntax**: strrev(string)
- **Use Case**: Useful for algorithms, such as palindrome checks.

Example:

```
<?php
$str = "Hello, PHP!";
echo "Reversed string: " . strrev($str);
?>
```

Output:

```
Reversed string: !PHP ,olleH
```

Math Functions in PHP

PHP provides various built-in math functions for performing arithmetic operations and computations. Below is a detailed explanation of commonly used math functions, along with syntax and examples.

PHP math functions perform mathematical operations.

List of Math Functions:

1. `abs()` - Returns the absolute value of a number.
2. `ceil()` - Rounds a number up to the nearest integer.
3. `floor()` - Rounds a number down to the nearest integer.
4. `round()` - Rounds a number to the nearest integer.
5. `pow()` - Raises a number to the power of another.
6. `sqrt()` - Returns the square root of a number.
7. `rand()` - Generates a random number.
8. `max()` - Finds the largest number in a set.
9. `min()` - Finds the smallest number in a set.

---

1. abs()

- **Description**: Returns the absolute (non-negative) value of a number.
- **Syntax**: abs(number)
- **Use Case**: Useful for ensuring positive values, such as distances or magnitudes.

Example:

```
<?php
echo "Absolute value of -10: " . abs(-10);
?>
```

Output:

```
Absolute value of -10: 10
```

---

2. ceil()

- **Description**: Rounds a number up to the nearest integer.
- **Syntax**: ceil(number)
- **Use Case**: Useful for calculations where fractional parts should be rounded up.

Example:

```
<?php
echo "Ceiling of 4.2: " . ceil(4.2);
?>
```

Output:

```
Ceiling of 4.2: 5
```

---

3. floor()

- **Description**: Rounds a number down to the nearest integer.

- **Syntax**: floor(number)
- **Use Case**: Useful for calculations where fractional parts should be rounded down.

Example:

```
<?php
echo "Floor of 4.8: " . floor(4.8);
?>
```

Output:

```
Floor of 4.8: 4
```

---

4. round()

- **Description**: Rounds a number to the nearest integer. By default, it rounds half up.
- **Syntax**: round(number, [precision], [mode])
- **Parameters**:
  - precision (optional): Number of decimal digits to round to.
  - mode (optional): Specifies rounding behavior.
- **Use Case**: Useful for formatting numbers to a specified precision.

Example:

```
<?php
echo "Round 4.5: " . round(4.5);
?>
```

Output:

```
Round 4.5: 5
```

---

5. pow()

- **Description**: Raises a number to the power of another number.
- **Syntax**: pow(base, exp)
- **Use Case**: Useful for exponential calculations.

Example:

```
<?php
echo "2 to the power of 3: " . pow(2, 3);
?>
```

Output:

```
2 to the power of 3: 8
```

---

6. sqrt()

- **Description**: Returns the square root of a number.
- **Syntax**: sqrt(number)
- **Use Case**: Useful for geometric and scientific calculations.

Example:

```
<?php
echo "Square root of 16: " . sqrt(16);
?>
```

Output:

```
Square root of 16: 4
```

---

7. rand()

- **Description**: Generates a random integer.
- **Syntax**: rand([min], [max])
- **Parameters**:
  - min (optional): The minimum value.
  - max (optional): The maximum value.
- **Use Case**: Useful for generating random numbers for games or simulations.

Example:

```
<?php
echo "Random number between 1 and 100: " . rand(1, 100);
?>
```

Output:

Random number between 1 and 100: 42 (Output may vary)

---

8. max()

- **Description**: Finds the largest number in a set of values.
- **Syntax**: max(value1, value2, ..., valueN)
- **Use Case**: Useful for finding the maximum value in a dataset.

Example:

```
<?php
echo "Maximum value: " . max(3, 7, 9, 2);
?>
```

Output:

Maximum value: 9

---

9. min()

- **Description**: Finds the smallest number in a set of values.
- **Syntax**: min(value1, value2, ..., valueN)
- **Use Case**: Useful for finding the minimum value in a dataset.

Example:

```
<?php
echo "Minimum value: " . min(3, 7, 9, 2);
?>
```

Output:

Minimum value: 2

Array Functions in PHP

PHP provides a variety of built-in functions to work efficiently with arrays. Below is a detailed explanation of commonly used array functions with syntax, examples, and outputs.

PHP array functions allow working with arrays, such as sorting, merging, and finding elements.

List of Array Functions:

1. count() - Returns the number of elements in an array.
2. array_merge() - Merges two or more arrays.
3. in_array() - Checks if a value exists in an array.
4. array_push() - Adds one or more elements to the end of an array.
5. array_pop() - Removes the last element of an array.
6. sort() - Sorts an array in ascending order.
7. rsort() - Sorts an array in descending order.
8. array_slice() - Extracts a portion of an array.
9. array_keys() - Returns all keys of an array.
10. array_values() - Returns all values of an array.

---

1. count()

- **Description**: Returns the number of elements in an array.
- **Syntax**: count(array)
- **Use Case**: Useful for checking the size of an array.

Example:

```
<?php
$arr = [1, 2, 3, 4, 5];
echo "Number of elements in the array: " . count($arr);
?>
```

Output:

```
Number of elements in the array: 5
```

---

2. array_merge()

- **Description**: Merges two or more arrays into one.
- **Syntax**: array_merge(array1, array2, ...)
- **Use Case**: Useful for combining data.

Example:

```
<?php
$arr1 = ["red", "green"];
$arr2 = ["blue", "yellow"];
$result = array_merge($arr1, $arr2);
print_r($result);
?>
```

Output:

```
Array ( [0] => red [1] => green [2] => blue [3] => yellow )
```

---

3. in_array()

- **Description**: Checks if a value exists in an array.

- **Syntax**: in_array(value, array, [strict])
- **Parameters**:
  - value: The value to search for.
  - strict (optional): If true, checks for type as well.
- **Use Case**: Useful for searching specific elements.

Example:

```
<?php
$arr = ["apple", "banana", "cherry"];
echo in_array("banana", $arr) ? "Found" : "Not Found";
?>
```

Output:

```
Found
```

---

4. array_push()

- **Description**: Adds one or more elements to the end of an array.
- **Syntax**: array_push(array, value1, value2, ...)
- **Use Case**: Useful for dynamically adding elements.

Example:

```
<?php
$arr = ["red", "green"];
array_push($arr, "blue", "yellow");
print_r($arr);
?>
```

Output:

```
Array ( [0] => red [1] => green [2] => blue [3] => yellow )
```

---

5. array_pop()

- **Description**: Removes and returns the last element of an array.
- **Syntax**: array_pop(array)
- **Use Case**: Useful for stack-like operations (LIFO).

Example:

```
<?php
$arr = ["red", "green", "blue"];
$last = array_pop($arr);
echo "Removed element: $last\n";
print_r($arr);
?>
```

Output:

```
mathematica

Removed element: blue
Array ( [0] => red [1] => green )
```

---

6. sort()

- **Description**: Sorts an array in ascending order.
- **Syntax**: sort(array)
- **Use Case**: Useful for arranging data in increasing order.

Example:

```
<?php
$arr = [4, 2, 8, 1];
sort($arr);
print_r($arr);
?>
```

Output:

```
Array ( [0] => 1 [1] => 2 [2] => 4 [3] => 8 )
```

---

7. rsort()

- **Description**: Sorts an array in descending order.
- **Syntax**: rsort(array)
- **Use Case**: Useful for arranging data in decreasing order.

Example:

```
<?php
$arr = [4, 2, 8, 1];
rsort($arr);
print_r($arr);
?>
```

Output:

```
Array ( [0] => 8 [1] => 4 [2] => 2 [3] => 1 )
```

---

8. array_slice()

- **Description**: Extracts a portion of an array.
- **Syntax**: array_slice(array, offset, [length], [preserve_keys])
- **Parameters**:
  - offset: Start index.
  - length (optional): Number of elements to return.
  - preserve_keys (optional): If true, retains original keys.
- **Use Case**: Useful for pagination or extracting subsets.

Example:

```
<?php
$arr = ["a", "b", "c", "d", "e"];
print_r(array_slice($arr, 2, 2));
?>
```

Output:

```
Array ( [0] => c [1] => d )
```

---

9. array_keys()

- **Description**: Returns all keys of an array.

- **Syntax**: array_keys(array)
- **Use Case**: Useful for fetching all keys for associative arrays.

Example:

```
<?php
$arr = ["name" => "John", "age" => 25, "city" => "New York"];
print_r(array_keys($arr));
?>
```

Output:

```
Array ( [0] => name [1] => age [2] => city )
```

---

10. array_values()

- **Description**: Returns all values of an array.
- **Syntax**: array_values(array)
- **Use Case**: Useful for fetching only values from an array.

Example:

```
<?php
$arr = ["name" => "John", "age" => 25, "city" => "New York"];
print_r(array_values($arr));
?>
```

Output:

```
Array ( [0] => John [1] => 25 [2] => New York )
```