# Web Application Development

Using

# PHP – MYSQL

**EVS Professional Training Institute**
**http://www.evslearning.com/**
**+92-42-111-685-822**
**+92-51-111-685-822**

**Compiled By :**
**Kashif Umar**
**kashif_umar@ymail.com**

# Week 2 – Lecture 1

## ❖ Strings

- ✦ As PHP is a hypertext preprocessor, and most textual data is represented as strings.
- ✦ Strings are an essential part of the language and probably more used than any other data type.
- ✦ For example, data read in from a file, database, email, or Web page is represented as string data.
- ✦ By definition, a PHP string is a piece of text, a series of characters (called bytes) enclosed in quotes.
- ✦ Because PHP puts no boundaries on the length of a string, it can consist of one character, a word, or an entire novel.
- ✦ PHP, in addition to basic operators, provides a huge collection of useful string functions to help you manipulate actions such as comparing strings, searching for strings, extracting substrings, copying strings, trimming strings, and translating characters in strings to uppercase or lowercase

### ➢ Quotes

- ✦ There are two types of quotes and thus two types of strings: singly quoted strings and doubly quoted strings.

- ✦ **Single Quotes**
  - ⅄ All characters enclosed within single quotes are treated as literals, so what you see is what you get, with the exception of a single quote embedded within a set of single quotes, and the backslash character. The quotes must be matched, a single quote to start the string and a single quote to terminate it.

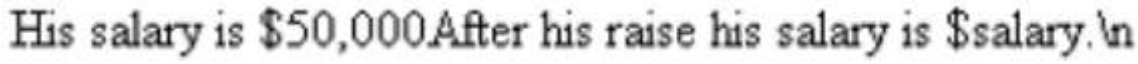
**Example**

```
<?php
print 'His salary is $50,000';
$salary=50000 * 1.1;
print 'After his raise his salary is $salary\n';
?>
```

All characters are treated literally within single quotes, the dollar sign in salary is not interpreted

as a variable. The \n to represent the newline is also treated literally. When these characters are inserted between double quotes, they will be interpreted; that is, the value of the variable, $salary, will be extracted, and the backslash sequence \n will be converted into a newline.

**Output**




- **Quoting Errors**
  - Because quotes are matched from left to right, embedding a single quote in a string, such as 'I don't care', would produce an error because PHP would treat the quote in the contraction don't as the terminating single quote for the string.
  - The solution is to either place the whole string in double quotes or precede the apostrophe in don't with a backslash (e.g., don\'t).

**Example**

```
<?php
$business = 'Joe's Pizza';
print $business;
?>
```

The first single quote opens the string and is matched by the next single quote. The problem: PHP sees the apostrophe in Joe's as the string's closing quote because it is the next quote it encounters after the initial single quote. The rest of the string is syntactically invalid and PHP reports an error

**Output**


```
$business = 'Joe\'s Pizza';
```

  - The backslash character takes away the special meaning of the inner quote. PHP will treat it as any other character in the string and continue looking for the closing single quote to terminate the string.

- **Double Quotes**
  - Another way to denote a string is to enclose it within double quotes:
    $business = "Joe's Pizza";
  - When PHP encounters the first double quote in the string "Joe's Pizza" it considers all the enclosed text as part of a string until it reaches the closing double quote.
  - The inner quote is ignored and treated as just another character.
  - Double quotes are like single quotes, but with three important exceptions:
    - They interpret escape sequences, which consist of a backslash followed by a single character. When enclosed in double quotes, the backslash causes the interpretation of the next character to "escape" from its normal ASCII code and to represent something else. For example, "\t" is interpreted as a tab character and "\n" as a newline character. (If the ouput from PHP will be displayed in a browser, the HTML <pre> tag should be used or the escape sequences will not be interpreted.) Escape sequences will not be displayed in your browser unless you use the HTML <pre> tag.
    - Single quotes are ignored within double quotes, such as in "Joe's Pizza".
    - Variables are replaced with their values when placed between double quotes.

## ❖ String Operators

### ➢ Concatenation

- String concatenation is the merging of one or more strings into one string. You might recall the string concatenation operator is a dot (.). It concatenates its left and right operand.

**Example**

```
<html>
<head><title>String Concatenation</title></head>
<body bgcolor="silver">
<h3>String Concatenation</h3>
<pre>
<?php
$name = 'Ellie Quigley';
$street = '123 Main Street';
```

```
$city = 'San Francisco';
$state = 'CA';
$zip = '94107';
$address="Name:  "  .  "$name\n"  .  "Address:  "  .
"$street\n" .
"Zip: ". $zip ."\n";
print $address;
print ".............................\n";
?>
</pre>
</body>
</html>
```

**Explanation**

- String values are assigned to a set of variables.
- In this example we use the dot (.) operator, the PHP concatenation operator, to "glue" together multiple strings into one long string. You use the concatenation operator to merge any two strings, whether they are single--quoted, double--quoted, or assigned to variables.
- The variable, $address, contains the concatenated string values.
- If you put the dot within a string, it is just the literal dot character with no special meaning.

## ➢ Equal and Identical

- The equality operator, ==, can be used to see if two strings are equal, and the === operator can be used to check that the strings are identical.
- If you are using these operators for string comparison, make sure that both of the operands are strings, because if you are comparing a string to a number, PHP will first cast the string to a number.
- This means that all strings that don't begin with a numeric value will be cast to zero.
- For instance, if ("total" > 5) will actually be compared as if (0 > 5).

**Example**

```
<html><head><title>Equal and Identical Strings</title>
</head>
<body bgcolor="lavender">
<font size="+1">
<h3>The == and === Operator</h3>
<?php
$str1="hello";
$str2="hello";
$str3=0;
if ( $str1 == $str2 ){
// They are equal
print "\"$str1\" and \"$str2\" are equal.<br />";
}
else{
print "\"$str1\" and \"$str2\" are not equal.<br />";
}
if ( $str2 == $str3 ){
print "\"$str2\" and $str3 are equal.<br />";
}
else{
print "\"$str2\" and $str3 are not equal.<br />";
}
if ($str2 === $str3){
print "\"$str2\" and $str3 are identical.<br />";
}
else{
print "\"$str2\" and $str3 are not identical.<br />";
}
?>
</body>
</html>
```

**Explanation**

- Three variables are defined. The first two are assigned string values, and the third is assigned a number.
- Because both strings contain the same value, "hello", they are considered equal; that is, all the characters are the same.
- Here a string, "hello", is being compared to a number, 0. PHP will convert the string to 0 and compare. They are now equal numeric values.
- This time, the identity operator compares the string by both data type and value. One is a string and the other a number, so they are not identical.

## ❖ Examples of String Functions

### ➢ trim() and strtolower()/strtoupper()

- ✦ This example uses trim() and strtolower() to improve the form validation script.
- ✦ **Code Sample: Strings/Demos/Greeting.php**

```
<html>
<head>
 <title>Greeting Page</title>
</head>
<body>
<?php
 $LastName = trim($_GET['LastName']);
 $Gender = strtolower(trim($_GET['Gender']));

 if ($LastName == '' || $Gender == '')
 {
  echo 'Error: You must fill out the form.
    Please <a href="Greeting.html">try again</a>.';
 }
 else
 {
  switch($Gender)
  {
   case 'male' :
    echo "Hello Mr. $LastName!";
    break;
   case 'female' :
    echo "Hello Ms. $LastName!";
    break;
   default :
    echo "<b>$Gender</b> is not a gender!";
  }
 }
?>
</body>
</html>
```

- **explode()**

The explode() function is used to convert a string to an array. The following form submits to Explode.php, the code of which is shown below.


**Code Sample: Strings/Demos/Explode.php**

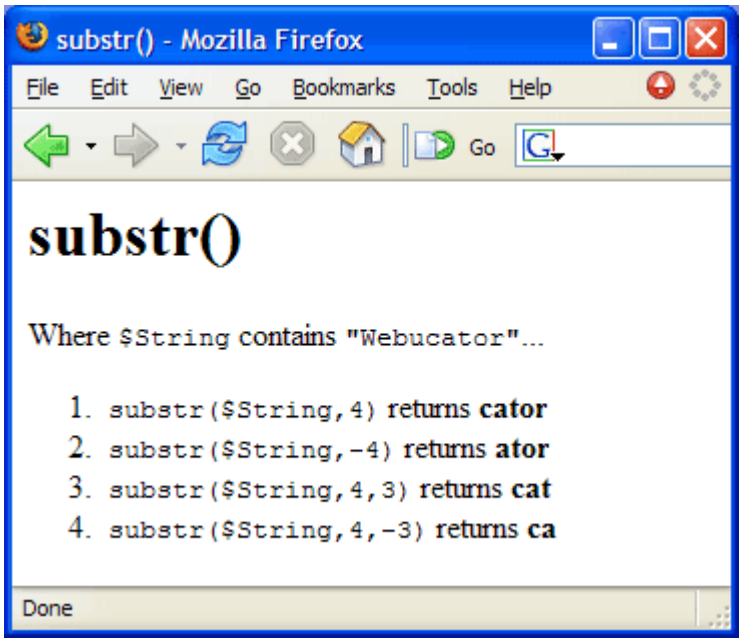
```
<html>
<head>
<title>Exploding Emails</title>
</head>
<body>
<?php
 $Emails = explode(';',$_POST['Emails']);
 echo '<ol>';
 foreach ($Emails as $Email)
 {
  echo '<li>' . trim($Email) . '</li>';
 }
 echo '</ol>';
?>
</body>
</html>
```

Notice that the trim() function is used to trim the resulting elements of the array. This is because the string is exploded on the semi-colon only. If the user

adds additional whitespace around the semi-colon, that whitespace will be part of the array element.

- **substr()**

  As shown earlier, the substr() function behaves differently depending on the values passed to it. The following screenshot shows the effects of using substr().




## ❖ String Manipulation Functions

| Trimming Strings | |
|---|---|
| **Function** | **Description** |
| trim() | Removes whitespace at beginning and end of a string. |
| ltrim() | Removes whitespace at the beginning of a string. |
| rtrim() | Removes whitespace at the end of a string. |

| **Presentation** | |
|---|---|
| **Function** | **Description** |
| htmlentities() | Escapes all HTML entities. |
| nl2br() | Inserts a <br /> tag before each newline character in a string. |
| strtoupper() | Converts a string to uppercase. |
| strtolower() | Converts a string to lowercase. |
| ucfirst() | Converts the first character of a string to uppercase. |
| ucwords() | Converts the first character of each word in a string to uppercase. |

| **Converting Strings and Arrays** | |
|---|---|
| **Function** | **Description** |
| explode() | Splits a string into an array on a specified character or group of characters. |
| implode() | Converts an array into a string, placing a specified character or group of characters between each array element. |
| join() | Same as implode(). |

| **Substrings** | |
|---|---|
| **Function** | **Description** |
| substr(str,pos) | Returns the substring from the character in position pos to the end of the string. |
| substr(str,-len) | Returns the substring from len characters from the end of the string to the end of the string. |
| substr(str,pos,len) | Returns a len length substring beginning with the character in position pos. |
| substr(str,pos,-len) | Returns a substring beginning with the character in position pos and chopping off the last len characters of the string. |

| **Substrings** | |
|---|---|
| strstr(haystack,needle,before_needle) | If the third argument (before_needle) is false (default), then it returns the part of the haystack from the needle on.<br><br>If the third argument (before_needle) is true, then it returns the part of the haystack before the needle.<br><br>The needle can be a string or an integer (or a number that can be converted to an integer). |
| stristr(haystack,needle,before_needle) | Same as strstr(), but *case insensitive*. |
| strpos(haystack,needle) | Finds the position of the first occurrence of a specified needle in a haystack (string).<br><br>The needle can be a string or an integer (or a number that can be converted to an integer). |
| strrpos(haystack,needle) | Finds the position of the last occurrence of a specified needle in a haystack (string).<br><br>The needle can be a string or an integer (or a number that can be converted to an integer). |
| str_replace() | Replaces all occurrences of one string with another string. |

| **Comparing Strings** | |
|---|---|
| **Function** | **Description** |
| strcmp() | Compares two strings. Returns < 0 if str1 is less than str2, > 0 if str1 is greater than str2, and 0 if they are equal. |
| strcasecmp() | Like strcmp() but case *ins*ensitive. |
| strlen() | Returns the length of a string. |

# ❖ Date and Time

## ➢ date() function

- ✦ The date() function is used to format a local time, date, or both.

### ✦ Date and Time Formatting Options

| Option | Description |
|---|---|
| a | am or pm |
| A | AM or PM |
| B | Swatch Internet time |
| d | Day of the month, two digits with leading zeros (i.e., 01 to 31) |
| D | Day of the week in text, three letters (e.g., Mon for Monday) |
| F | Month in text (e.g., January) |
| g | Hour, 12-hour format without leading zeros (i.e., 1 to 12) |
| G | Hour, 24-hour format without leading zeros (i.e., 0 to 23) |
| h | Hour, 12-hour format with leading zeros (i.e., 01 to 12) |
| H | Hour, 24-hour format (i.e., 00 to 23) |
| i | Minutes with leading zeros (e.g., 00 to 59) |
| I | 1 if Daylight Savings Time, 0 if not |
| j | Day of the month without leading zeros (i.e., 1 to 31) |
| l | Day of the week in text (e.g., Friday) |
| L | 1 if it is a leap year, 0 if not |
| m | Month with leading zeros (i.e., 04 to 11) |
| M | Month in text, three letters (e.g., Jul) |
| n | Month without leading zeros (e.g., 1 to 12) |
| r | RFC 822 formatted date (i.e., Thu, 21 Dec 2000 16:01:07 +0200) |

| Option | Description |
|---|---|
| s | Seconds with leading zeros (e.g., 00 to 59) |
| S | English ordinal suffix, textual, two characters (e.g., th, nd) |
| t | Number of days in the given month (i.e., 28 to 31) |
| T | Time zone setting of this machine (i.e., MDT) |
| U | Seconds since the epoch, January 1 1970 00:00:00 GMT |
| w | Day of the week, numeric (i.e., 0 [Sunday] to 6 [Saturday]) |
| Y | Year, four digits (i.e., 2007) |
| y | Year, two digits (i.e., 07) |
| z | Day of the year (i.e., 0 to 365) |
| Z | Time zone offset in seconds (i.e., -43200 to 43200). The offset for time zones west of UTC is always negative, and for those east of UTC is always positive. |

### ✦ Examples

```
date("d-M-Y")
date("M/d/Y")
date("D dS M, Y h:i a")
date("\T\h\e \\t\i\m\e \i\s: h:m a")
date("\T\o\d\a\y \i\s: l")
```

## ➢ strftime() function

- ✦ The strftime() function is also used to format the date and time (based on the locale settings), similar to the date() function but with different options.

## ✦ Arguments for strftime() function

| Format Character | What It Means |
|---|---|
| %a | Abbreviated weekday name |
| %A | Full weekday name |
| %b | Abbreviated month name |
| %B | Full month name |
| %c | Preferred date and time representation |
| %C | Century number |
| %d | Day of the month as a decimal number (range 01 to 31) |
| %D | Same as %m/%d/%y |
| %e | Day of the month as a decimal number, a single digit is preceded by a space (range "1" to "31") |
| %g | Like %G, but without the century |

| Format Character | What It Means |
|---|---|
| %G | The four-digit year corresponding to the ISO week number (see %V) |
| %h | Same as %b |
| %H | Hour as a decimal number using a 24-hour clock (range 00 to 23) |
| %I | Hour as a decimal number using a 12-hour clock (range 01 to 12) |
| %j | Day of the year as a decimal number (range 001 to 366) |
| %m | Month as a decimal number (range 01 to 12) |
| %M | Minute as a decimal number |
| %n | Newline character |
| %p | Either am or pm according to the current locale |
| %r | Time in a.m. and p.m. notation |
| %R | Time in 24-hour notation |
| %S | Second as a decimal number |
| %t | Tab character |
| %T | Current time, equal to %H:%M:%S |
| %u | Weekday as a decimal number ( 1 to 7), with 1 representing Monday |
| %U | Week number of the current year as a decimal number, starting with the first Sunday as the first day of the first week |
| %V | The ISO 8601:1988 week number of the current year as a decimal number (range 01 to 53), where week 1 is the first week that has at least four days in the current year, and with Monday as the first day of the week |
| %W | Week number of the current year as a decimal number, starting with the first Monday as the first day of the first week |
| %w | Day of the week as a decimal, Sunday being 0 |
| %x | Preferred date representation for the current locale without the time |
| %X | Preferred time representation for the current locale without the date |
| %y | Year as a decimal number without a century (range 00 to 99) |
| %Y | Year as a decimal number including the century |

| Format Character | What It Means |
|---|---|
| %Z or %z | Time zone or name or abbreviation |
| %% | A literal % character |

## ➢ time() function

- ✦ The time() function returns the current timestamp in seconds.
- ✦ For a more accurate time you can use the microtime() function.
- ✦ By adding or subtracting the number of seconds from the output of the time() function, you find the amount of time from now until some future or past time.

| Unit of Time | Seconds |
|---|---|
| Hour | 60 * 60 |
| Day | 60 * 60 * 24 |
| Week | 60 * 60 * 24 * 7 |
| Month | 60 * 60 * 24 * 30 |
| Year | 60 * 60 * 24 * 365 |

### ✦ Examples

```
date("M-d-Y", time()),"\n"; // Today
date("M-d-Y", time() + (7 * 24 * 60 * 60)); // Next week
date("M-d-Y", time() - (7 * 24 * 60 * 60)); // Last week
```

## ➢ mktime() function

- ✦ This function returns the timestamp for a particular date.
- ✦ This timestamp is an integer containing the number of seconds between the UNIX epoch (January 1 1970 00:00:00 GMT) and the time specified in the argument list.

| Argument | Meaning |
|---|---|
| Hour | Number of Hours |
| Minute | Number of minutes |
| Second | Number of seconds |
| Month | Number of the month |
| Day | Number of the day |
| Year | Number of the year, can be a two or four year digit value |

## ✦ Example - 1

```
$seconds=mktime(0, 0, 0, 7, 19, 2007);
print "$seconds\n"; // 1184828400
echo date("M-d-Y", $seconds), "\n"; // Jul-19-2007
date("M-d-Y", mktime(0, 0, 0, 7, 18, 2007));
//Jul-18-2007
date("M-d-Y", mktime(0, 0, 0, 1, 1, 07));
//Jan-01-2007
date("M-d-Y", mktime(0, 0, 0, 7, 32, 7));
//Aug-01-2007
```

## ✦ Example - 2

```
$today=mktime(0,0,0, date("m"),date("d"),date("Y"));
echo strftime("Today is %m/%d/%Y",$ today),".<br>";
$tomorrow = mktime(0, 0, 0, date("m"),date("d")+1,
date("Y"));

echo strftime("Tomorrow is %A", $tomorrow),".<br>";
$yesterday = mktime(0, 0, 0, date("m") , date("d")-1,
date("Y"));

echo strftime("Yesterday was %A", $yesterday),".<br>";
$nextmonth = mktime(0, 0, 0, date("m")+1, date("d"),
date("Y"));

echo strftime("Next month is %B", $nextmonth),".<br>";
$lastmonth = mktime(0, 0, 0, date("m")-1, date("d"),
date("Y"));

echo strftime("Last month was %B",$lastmonth),".<br>";
$nextyear = mktime(0, 0, 0, date("m"), date("d"),
date("Y")+1);

echo strftime("Next year is %Y", $nextyear),".<br>";
$lastyear = mktime(0, 0, 0, date("m"), date("d"),
date("Y")-1);

echo strftime("Last year was %Y", $lastyear),".<br>";
```

## ➢ getdate() function

- ✦ The getdate() function returns an associative array containing date and time values for the local time

| Key | Value |
|---|---|
| Seconds | Seconds, range 0 to 59 |
| minutes | Minutes, range 0 to 59 |
| hours | Hours, range 0 to 23 |
| mday | Day of the month, range 1 to 31 |
| wday | Day of the the week, range 0 (Sunday), through 6(Saturday) |
| mon | A month, range 1 through 12 |
| year | A year in four--‐digit format, such as 1999 or 2006 |
| yday | Day of the year, range 0 through 365 |
| weekday | The day of the week, range Sunday through Saturday |
| month | The month, such as January or December |

### ✦ Example - 1

```
extract(getdate());

echo("Seconds = $seconds<br/>");
echo("Minutes = $minutes<br/>");
echo("Hours = $hours<br/>");
echo("Day of The Month = $mday<br/>");
echo("day Of The Week = $wday<br/>");
echo("Month = $mon<br/>");
echo("Year = $year<br/>");
echo("day of the year = $yday<br/>");
echo("day of the week = $weekday<br/>");
echo("Month = $month<br/>");
```

### ✦ Example – 2

```
$date=mktime(0, 0, 0, 12, 25, 2006);
echo("date = $date<br/>");
$holiday=getdate($date);
print_r($holiday);
echo "$holiday[month] $holiday[mday], $holiday[year]
is on a ", $holiday["weekday"], ".\n<br>";
```

## ➢ Validating Dates

- The checkdate() function checks whether the arguments it receives are valid date values. It returns TRUE if the date is valid, and FALSE if not. Valid date values are:
  - The month must be between 1 and 12 inclusive.
  - The day must be within the allowed number of days for the given month, and leap years are allowed.
  - The year must be between 1 and 32767 inclusive.

### ✦ Example

```
echo "\nIs 12/31/2006 a valid date? ";
var_dump(checkdate(12, 31, 2006));
echo "\nIs 12/32/2006 a valid date? ";
var_dump(checkdate(12,32, 2006));
echo "Is this better? ";
// Let mktime() adjust the date to the correct date
echo date('m/d/Y', mktime(0,0,0, 12,32,2006)),"\n";
echo "\nIs 2/29/2008 a leap year? ";
var_dump(checkdate(2, 29, 2008));
echo "\nIs 2/29/2006 a leap year? ";
var_dump(checkdate(2, 29, 2006));
```

## ❖ Regular Expressions

- When a user fills out a form, you might want to verify that the format was correct before sending the data to a database. For example, did the user enter a valid birthdate, e-mail address, or credit card number? This is where regular expressions enter the picture. Their power is great and they are used by many other programming languages for handling text, for performing refined searches and replacements, capturing subpatterns in strings, testing input data for certain characters, and more.

- So, what is a regular expression? A regular expression is really just a sequence or pattern of characters that is matched against a string of text when performing searches. When you create a regular expression, you test the regular expression against a string. The regular expression is enclosed in forward slashes. For example, the regular expression /green/ might be matched against the string "The green grass grows". If green is contained in the string, there is a successful match. Like Perl, PHP also provides a large variety of regular expression metacharacters to control the way a pattern is found; for example, the regular expression /^[Gg]reen/ consists of a caret and a set of square brackets. These metacharacters control the search so that the regular expression matches only strings starting with an upper- or lowercase letter g. The possibilities of fine-tuning your search with regular expressions and their metacharacters are endless.

- PHP regular expressions are used primarily to verify data on the server side. When a user fills out a form and presses the submit button, the form is sent to a server, and then to a PHP script for further processing. Although it is more efficient to handle form validation on the client side with programs like Javascript or JScript, these programs might be disabled, or might not be programmed to verify form data. Checking the form on the client side allows for instant feedback, and less travelling back and forth between the browser and server, but to ensure that the data has been verified, PHP can recheck it. Once the user has filled out a form and submitted it, PHP can check to see if all the boxes have been filled out

correctly, and if not, the user is told to reenter the data before the form data is processed. With the power provided by regular expressions, the ability to check for any type of input, such as e-mail addresses, passwords, social security numbers, birthdates, and so on, is greatly simplified. You can also use regular expressions to complete complex search and replace operations in text files, processes that would be difficult, if not impossible, with PHP's standard string functions.

## Types of Regular Expression in PHP

- POSIX
- Perl Style

## POSIX

- The first set of functions (POSIX style) are those prefixed with ereg_. They behave much like the traditional UNIX egrep command. The advantage of the ereg functions is that they are supported by the oldest versions of PHP.

## Perl Style

- The second set of regular expression functions (Perl style) start with preg_. These functions mimic Perl regular expressions and support the newer features, such as backreferences, capturing, lookahead, and lookbehind. These functions are only available if your version of PHP was compiled with support for the PCRE (Perl Compatible Regular Expression) library, and the PCRE library is installed on your Web server. Check the phpinfo() output from your first test scripts to see if PCRE is enabled

**pcre**

| | |
|---|---|
| PCRE (Perl Compatible Regular Expressions) Support | enabled |
| PCRE Library Version | 4.5 01-December-2003 |

## ➢ Why Perl style regular expressions?

Perl is a popular powerful scripting language known for its ability to manipulate and extract text. It supports regular expressions and regular expression metacharacters to make pattern matching relatively easy and quick. PHP has mimicked Perl by providing special functions to handle pattern matching and included Perl's metacharacters for pattern matching. We discuss each of the pattern-matching functions before delving into regular expression metacharacters.

| Function | What It Does |
|---|---|
| `preg_grep()` (PHP 4, PHP 5) | Returns an array of patterns that were matched. |
| `preg_match()` | Performs a regular expression pattern match. |
| `preg_match_all()` | Performs a global regular expression match. |
| `preg_quote()` | Puts a backslash in front of regular expression characters found within a string. |
| `preg_replace()` | Searches for a pattern and replaces it with another. |
| `preg_replace_callback()` | Like `preg_replace()`, but uses a function for the replacement argument. |
| `preg_split()` | Splits up a string into substrings using a regular expression as the delimiter. |

## ➢ Pattern-Matching Functions

Following Table lists the PHP built-in functions that will be used for performing searches with regular expressions, performing searches and replacements, splitting up strings based on a regular expression delimiter, and so on. Both the Perl style and POSIX style functions are listed in the following two tables, but we would focuses on the Perl style functions

## Regular Expression Functions—POSIX Style

| Function | What It Does |
|---|---|
| ereg() | Performs a regular expression pattern match. |
| eregi() | Performs a case-insensitive regular expression pattern match. |
| ereg_replace() | Searches for a pattern and replaces it with another. |
| eregi_replace() | Searches for a pattern and replaces it with another, case insensitive. |
| split() | Splits a string into an array by using a regular expression as the delimiter. |
| spliti() | Splits a string into an array by a regular expression and is case insensitive. |

## Examples

### ➢ Finding a Pattern preg_match() / preg_match_all()

- ✦ Both used to find a pattern in a string
- ✦ preg_match() only finds the first occurrence
- ✦ preg_match_all() finds all the occurrences

### ➢ Example

- ✦ **Ex - 1**

```
$string = "My gloves are worse love for wear";
$reg = "/love/";
$result = preg_match($reg, $string, $matches);

echo("result = $result<br/>");
Contents of matches
echo("<pre>");
print_r($matches);
echo("</pre>");
```

**Output**

```
result = 1
Contents of matches
Array
(
    [0] => love
)
```

✦ **Ex - 2**

```
$string = "My gloves are worse love for wear";
$reg = "/(love) for (wear)/";
$result = preg_match($reg, $string, $matches);

echo("result = $result<br/>");
Contents of matches
echo("<pre>");
print_r($matches);
echo("</pre>");
```

**Output**

```
result =
Contents of matches
Array
(
    [0] => love for wear
    [1] => love
    [2] => wear
)
```

✦ **Ex - 3**

```
   $string = "My gloves are worse love for wear";
   $reg = "/love/";
   $result = preg_match($reg, $string, $matches,
   PREG_OFFSET_CAPTURE);

echo("result = $result<br/>");
Contents of matches
echo("<pre>");
print_r($matches);
echo("</pre>");
```

**Output**

```
result = 1
Contents of matches
   Array
   (
       [0] => Array
           (
               [0] => love
               [1] => 4
           )

   )
```

✦ **Ex - 4**

```
$string = "My gloves are worse love for wear";
$reg = "/(love) for (wear)/";
$result = preg_match($reg, $string, $matches,
PREG_OFFSET_CAPTURE);

echo("result = $result<br/>");
Contents of matches
echo("<pre>");
print_r($matches);
echo("</pre>");
```

**Output**

```
result =
Contents of matches
Array
(
    [0] => Array
        (
            [0] => love for wear
            [1] => 20
        )

    [1] => Array
        (
            [0] => love
            [1] => 20
        )

    [2] => Array
        (
            [0] => wear
            [1] => 29
        )

)
```

✦ **Ex - 5**

```
$string = "My gloves are worse love for wear";
$reg = "/love/";
$result = preg_match_all($reg, $string, $matches);

echo("result = $result<br/>");
Contents of matches
echo("<pre>");
```

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

**Output**

```
result = 2
Contents of matches
Array
(
    [0] => Array
        (
            [0] => love
            [1] => love
        )

)
```

✦ **Ex - 6**

```
$string = "My gloves are worse love for wear";
$reg = "/(love) for (wear)/";
$result = preg_match_all($reg, $string, $matches);

echo("result = $result<br/>");
Contents of matches
echo("<pre>");
print_r($matches);
echo("</pre>");
```

**Output**

```
result =
Contents of matches
Array
(
    [0] => Array
        (
            [0] => love for wear
        )
    [1] => Array
        (
            [0] => love
        )
    [2] => Array
        (
            [0] => wear
        )

)
```

✦ **Ex - 7**

```
$string = "My gloves are worse love for wear";
$reg = "/love/";
$result = preg_match_all($reg, $string, $matches,
PREG_OFFSET_CAPTURE);

echo("result = $result<br/>");
Contents of matches
echo("<pre>");
print_r($matches);
echo("</pre>");
```

**Output**

```
result = 2
Contents of matches
Array
(
    [0] => Array
        (
            [0] => Array
                (
                    [0] => love
                    [1] => 4
                )

            [1] => Array
                (
                    [0] => love
                    [1] => 20
                )

        )
)
```

✦ **Ex - 8**

```
$string = "My gloves are worse love for wear";
$reg = "/(love) for (wear)/";
$result = preg_match_all($reg, $string, $matches,
PREG_OFFSET_CAPTURE);

echo("result = $result<br/>");
Contents of matches
echo("<pre>");
print_r($matches);
echo("</pre>");
```

**Output**

```
result =
Contents of matches
Array
(
    [0] => Array
        (
            [0] => Array
                (
                    [0] => love for wear
                    [1] => 20
                )

        )

    [1] => Array
        (
            [0] => Array
                (
                    [0] => love
                    [1] => 20
                )

        )

    [2] => Array
        (
            [0] => Array
                (
                    [0] => wear
                    [1] => 29
                )

        )

 )
```

## ➢ Searching and Replacing preg_replace()

✦ Searches for a pattern in the subject and replaces the subject with something else.

✦ **Example – 1**

```
$old_string = "I live in New Orleans.";
print ("Original string: <em>$old_string</em><br />");
$reg = "/New Orleans/";
$rep_string = "Philadelphia";

$new_string    =    preg_replace($reg,    $rep_string,
$old_string);

print("After replacement<br/>");
print ("Original string: <em>$old_string</em><br />");
print ("New string: <em>$new_string</em><br />");
```

**Output**

Original string: *I live in New Orleans.*
After replacement
Original string: *I live in New Orleans.*
New string: *I live in Philadelphia.*

✦ **Example – 2**

```
$subject = "The flag was <em>red, white, </em>and
<em>blue</em>.";

$search_reg = array('/red/','/white/','/blue/');
$replace = array('yellow','orange','green');

echo ("Before replacement: $subject<br />");
$result = preg_replace($search_reg,$replace,$subject);
echo ("After replacement: $result");
```

**Output**

Before replacement: The flag was red, white, and blue.
After replacement: The flag was yellow, orange, and green.

✦ **Example – 3**

```
echo(preg_replace("/5 pies/", "(5*3) . 'cupcakes'",
"$subject_string"));
```

**Output**

```
He ate (5*3) . 'cupcakes'.
```

✦ **Example – 4**

```
echo(preg_replace("/5 pies/e", "(5*3) . 'cupcakes'",
"$subject_string"));
```

**Output**

He ate 15 'cupcakes'.

## ➢ Splitting preg_split()

- ✦ array preg_split ( string pattern, string subject [, int limit [,int flags]] )
  - ⅄ pattern – to find
  - ⅄ subject – to search
  - ⅄ limit – how many splits, -1 is no limit

- ✦ Splits up a string by some delimiter that marks the separation between the words in the string, such as a space or a colon or a combination of such characters.
- ✦ The function returns an array of substrings.
- ✦ If a limit is specified, then only that many substrings are returned.
- ✦ If you are using a single character or simple string as the delimiter, the explode() function is faster

✦ **Example – 1**

```
$string="apples#oranges#peaches";
$array=preg_split("/#/", $string); // Split by #
echo("<pre>");
print_r($array);
echo("</pre>");
```

**Output**

```
Array
(
    [0] => apples
    [1] => oranges
    [2] => peaches
)
```

✦ **Example – 2 – Multiple Delimiters**

```
$colors="Primary:red,yellow,blue;Secondary:violet,oran
ge,green";
$array=preg_split("/[:,;]/", $colors);
echo "<h2>Splitting Colors</h2>";
echo("<pre>");
print_r($array);
echo("</pre>");
foreach ($array as $key=>$value)
{
     if ($value == "Primary" || $value == "Secondary")
     {
          print "$value<br />";
     }
     else
     {
          print "\t$key: $value<br />";
     }
}
```

**Output**

```
Array
(
    [0] => Primary
    [1] => red
    [2] => yellow
    [3] => blue
    [4] => Secondary
    [5] => violet
    [6] => orange
    [7] => green
)
Primary
1: red
2: yellow
3: blue
Secondary
5: violet
6: orange
7: green
```

✦ **Example – 3 – PREG_SPLIT_NO_EMPTY**

```
$alpha="SAN FRANCISCO";
$array =  preg_split("//", $alpha, 3,
PREG_SPLIT_NO_EMPTY);
$array=preg_split("//", $alpha, -1,
PREG_SPLIT_NO_EMPTY); // -1 is the number of splits, -
1 means no limit, 2 means only two splits
echo "<h2>Splitting A Word into Letters</h2><pre>";
print_r($array);
echo("</pre>");
```

**Output**

```
Array
(
    [0] => S
    [1] => A
    [2] => N
    [3] =>
    [4] => F
    [5] => R
    [6] => A
    [7] => N
    [8] => C
    [9] => I
    [10] => S
    [11] => C
    [12] => O
)
```

✦ **Example – 4 – PREG_SPLIT_OFFSET_CAPTURE**

```
$alpha="PORT OF SAN FRANCISCO";
$array=preg_split("/\s/", $alpha, -1,
PREG_SPLIT_OFFSET_CAPTURE);
echo("Original : $alpha<br/>");
echo "<h2>Splitting A Word into Letters</h2>";
echo("<pre>");
print_r($array);
echo("</pre>");
```

**Output**

```
Original : PORT OF SAN FRANCISCO
Splitting A Word into Letters

Array
(
    [0] => Array
        (
            [0] => PORT
            [1] => 0
        )

    [1] => Array
        (
            [0] => OF
            [1] => 5
        )

    [2] => Array
        (
            [0] => SAN
            [1] => 8
        )

    [3] => Array
        (
            [0] => FRANCISCO
            [1] => 12
        )

)
```

✦ **Related PHP Functions -** spliti(), split(), implode(), and explode()

| **Flag** | **What It Does** |
|---|---|
| PREG_SPLIT_DELIM_CAPTURE | The Captured pattern in the delimiter pattern will be saved and returned as well. |
| PREG_SPLIT_NO_EMPTY | Returns only nonempty pieces. |
| PREG_SPLIT_OFFSET_CAPTURE | For Every occurring match, an offset will be returned where the match occurred within the string. |

## ➢ Meta-Characters

| | |
|---|---|
| /love/ | Find "love" anywhere in the string |
| /fun and games/ | Finds "fun and games" |
| /(fun) and (games)/ | Finds "fun and games", "fun", "games". "fun" and "games" are sub-patterns within the original pattern |
| /(fun)/ | First find fun as a pattern and then fun as a sub-pattern, fun would be search twice |
| /Love/i | Case insensitive search |
| /love/A | Must be at the start of the string |
| /love/D | Must be at the end of the string |
| /5 pies/e | When using preg_replace, if the replacement string has an expression, it would be first evaluated then used |
| /^B/ | Words that start with B |
| /0$/ | Words ending with 0 |
| /\bbear\b/i | Case insensitive search of the exact word "bear" |
| /a.b/ | Starts with "a" then has any character (except newline) then "b" |
| /.../ | Any string of three characters |
| /^... /", | Start of string has 3 characters |
| /J../ | Starts with J and has two characters |
| /[abc]/ | a, or b or c |
| /[a-z0-9_]/ | a to z or 0 to 9 |
| /[^a-z0-9_]/ | Not a to z and not 0 to 9 |
| /[0-9][0-9][0-9]/ | 3 digits |
| /\d\d\d/ | 3 digits |
| /\d{3}/ | 3 digits |
| /^\w\w\w\W/ | The start of the string has 3 alpha-numeric and one non-alpha-numeric |
| /\W\D/ | A non-alpha-numeric and a non-digit |
| /e\s?[A-Z]/ | Letter "e" followed by space or no-space (\s?) and a Capital alphabet. ? means 0 or 1 |
| /B[a-z]*/ | Uppercase B with 0 or more occurrences of small alphabets |
| /^[A-Z][a-z]*\s[A-Z][a-z]*\s/ | The start of string is capital alphabet, then 0 or more small alphabets, then space, then a capital alphabet, then 0 or more small alphabets, then space |
| match("/^[A-Z][a-z]*\s[A-Z][a-zAZ]*\ s/ | The start of string is capital alphabet, then 0 or more small alphabets, then space, then a capital alphabet, then 0 or more capital or small alphabets, then space |
| /B[a-z]+/ | Starting with capital B and then 1 or more small alphabet |
| /\s\d{3}$/ | At the end of the string – One Space, 3 digits |
| /B.* / | Starting with capital B, then 0 or more of any character, up until the last space |

| | |
|---|---|
| /B.*? / | Starting with capital B, then 0 or more of any character, up until the first space |
| /Steve\|Betty\|Jon/ | Steve or Betty or John |
| /(Steve\|Alexander) Blenheim/ | Steve Blengeim or Alexander Blenheim |
| /(ma)+/ | Atleast one occurance of "ma" as a group |
| preg_replace("$regex",'$2, $1 $3', $text); | There are 3 sub-patterns in the regular expression, $1 represents the first one, $2 second and $3 third one |
| Social Security number (US) | /^\d{3}-?\d\d-?\d{4}$/ |
| U.S. Phone number | /^\(?\d{3}\)?-?\s*\d{3}\s*-?\d{4}$/ |
| Zip code | /^\d{5}((-\|\s)?\d{4})?$/ |
| E---mail | /^([0-9a-zA-Z]([-.\w]*[0-9a-zA-Z])*@([0-9a-zA-Z][-\w]*[0-9a-zAZ]\.)+[a-zA-Z]{2,4})$/ |
| Credit Card number | /^((4\d{3})\|(5[1-5]\d{2})\|(6011))-?\d{4}-?\d{4}-?\d{4}\|3[4,7]\d{13}$/ |
| URL | /^((http\|https\|ftp)://)?([\w-])+(\.)(\w){2,4}([\w/+=%&_.~?-]*)$/ |

```
/^[a-zA-Z][\w\.\-]+[a-zA-Z0-9]@([a-zA-Z0-9][a-zA-Z0-9\-]*\.)+[a-zA-Z]{2,4}$/
❶  ❷      ❸      ❹  ❺          ❻                    ❼  ❽
```

| What It Matches | Metacharacter |
|---|---|
| Matches any character except a newline. | `.` |
| Matches any single character in a set. | `[a-z0-9]` |
| Matches any single character not in a set. | `[^a-z0-9]` |
| Matches one digit. | `\d` |
| Matches a nondigit, same as `[^0-9]`. | `\D` |
| Matches an alphanumeric (word) character. | `\w` |
| Matches a nonalphanumeric (nonword) character. | `\W` |
| Matches whitespace character, spaces, tabs, and newlines. | `\s` |
| Matches a nonwhitespace character. | `\S` |
| Matches a newline. | `\n` |
| Matches a return. | `\r` |
| Matches a tab. | `\t` |
| Matches a form feed. | `\f` |
| Matches a null character | `\0` |
| Matches a word boundary. | `\b` |
| Matches a nonword boundary. | `\B` |
| Matches to beginning of line. | `^` |
| Matches to end of line. | `$` |
| Matches the beginning of the string only. | `\A` |
| Matches the end of the string or line. | `\D` |
| Matches 0 or 1 occurrences of the letter x. | `x?` |
| Matches 0 or more occurrences of the letter x. | `x*` |

| What It Matches | Metacharacter |
|---|---|
| Matches 1 or more occurrences of the letter x. | `x+` |
| Matches one or more patterns of `xyz` (e.g., `xyxxyzxyz`). | `(xyz)+` |
| Matches at least `m` occurrences of the letter `x`, and no more than `n` occurrences of the letter `x`. | `x{m,n}` |
| Matches one of `was`, `were`, or `will`. | `was\|were\|will` |
| Used for backreferencing.<br>Matches first set of parentheses.<br>Matches second set of parentheses.<br>Matches third set of parentheses. | `(string)`<br>`\1` or `$1`<br>`\2` or `$2`<br>`\3` or `$3` |
| Matches `x` but does not remember the match. These are called noncapturing parentheses. | `(?:x)` |
| Matches `x` only if `x` is followed by `y`. For example, `/Jack(?=Sprat)/` matches `Jack` only if it is followed by `Sprat`. `/Jack(?=Sprat\|Frost)/` matches `Jack` only if it is followed by `Sprat` or `Frost`. Neither `Sprat` nor `Frost` is kept as part of what was matched. | `x(?=y)` |
| Matches `x` only if `x` is not followed by `y`. For example, `/\d+(?!\.)/` matches one or more numbers only if they are not followed by a decimal point. | `x(?!y)` |