ISIT307 - WEB SERVER PROGRAMMING

LECTURE 1.2 – FUNCTIONS AND CONTROL STRUCTURES

LECTURE PLAN

- Learn how to use functions to organize the PHP code
- Learn about variable scope
- Learn `if` statements, `if...else` statements, and `switch` statements
- Learn `while` statements, `do...while` statements, `for`, and `foreach` statements
- Learn about `include` and `require` statements

DEFINING FUNCTIONS

- **Functions** are groups of statements that can be executed as a single unit
- **Function definitions** are the lines of code that make up a function
- The syntax for defining a function is:

```
<?php
function name_of_function(parameters) {
        statements;
}
?>
```

DEFINING FUNCTIONS

- Functions, like all PHP code, must be contained within `<?php . . . ?>` tags
- A **parameter** is a variable declared within the function definition
- Parameters are placed within the parentheses that follow the function name
- Parameters can be assigned a default values
- Functions do not have to contain parameters
- The set of curly braces (called **function braces**) contain the function statements

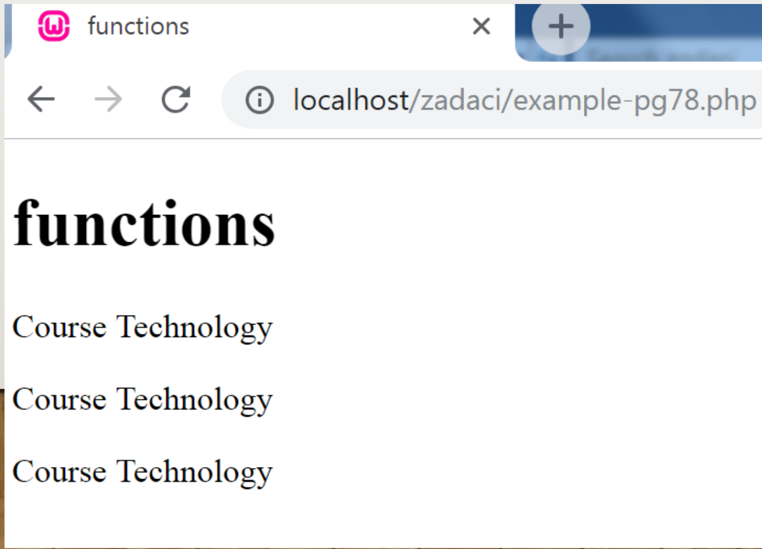
DEFINING FUNCTIONS

- **Function statements** do the actual work of the function and must be contained within the function braces

```
function displayCompanyName($Company1,
                      $Company2, $Company3) {
  echo "<p>$Company1</p>";
  echo "<p>$Company2</p>";
  echo "<p>$Company3</p>";
}
```

CALLING FUNCTIONS

```
<?php
function displayCompanyName($CompanyName) {
       echo "<p>$CompanyName</p>";
}
displayCompanyName("Course Technology");
DisplayCompanyName("Course Technology");
DISPLAYCOMPANYNAME("Course Technology");
?>
```



functions

Course Technology

Course Technology

Course Technology

RETURNING VALUES

- A **return statement** returns a value to the statement that called the function
- Not all functions return values

```
function average_numbers($a, $b, $c) {
   $SumOfNumbers = $a + $b + $c;
   $Result = $SumOfNumbers / 3;
   return $Result;
 }
 echo average_numbers (5,6,7);
 echo average_numbers (5,5,7); //what is the output?
```

RETURNING VALUES

- The function can **return multiple values** within an array

```
function multi_calc($n1, $n2, $n3)
{
   $sum = $n1+$n2+$n3;  //$res[] = $n1+$n2+$n3;
   $prod = $n1*$n2*$n3;  //$res[] = $n1*$n2*$n3;
   return array($sum, $prod); //return $res;
}
$result = multi_calc(5, 6, 7);
echo "Results are: ", $result[0], " and ", $result[1];
```

PASSING PARAMETERS TO A FUNCTION

- You can pass a function parameter by **value** or by **reference**
  - A function parameter that is passed by value is a local copy of the variable
  - A function parameter that is passed by reference is a reference to the original variable

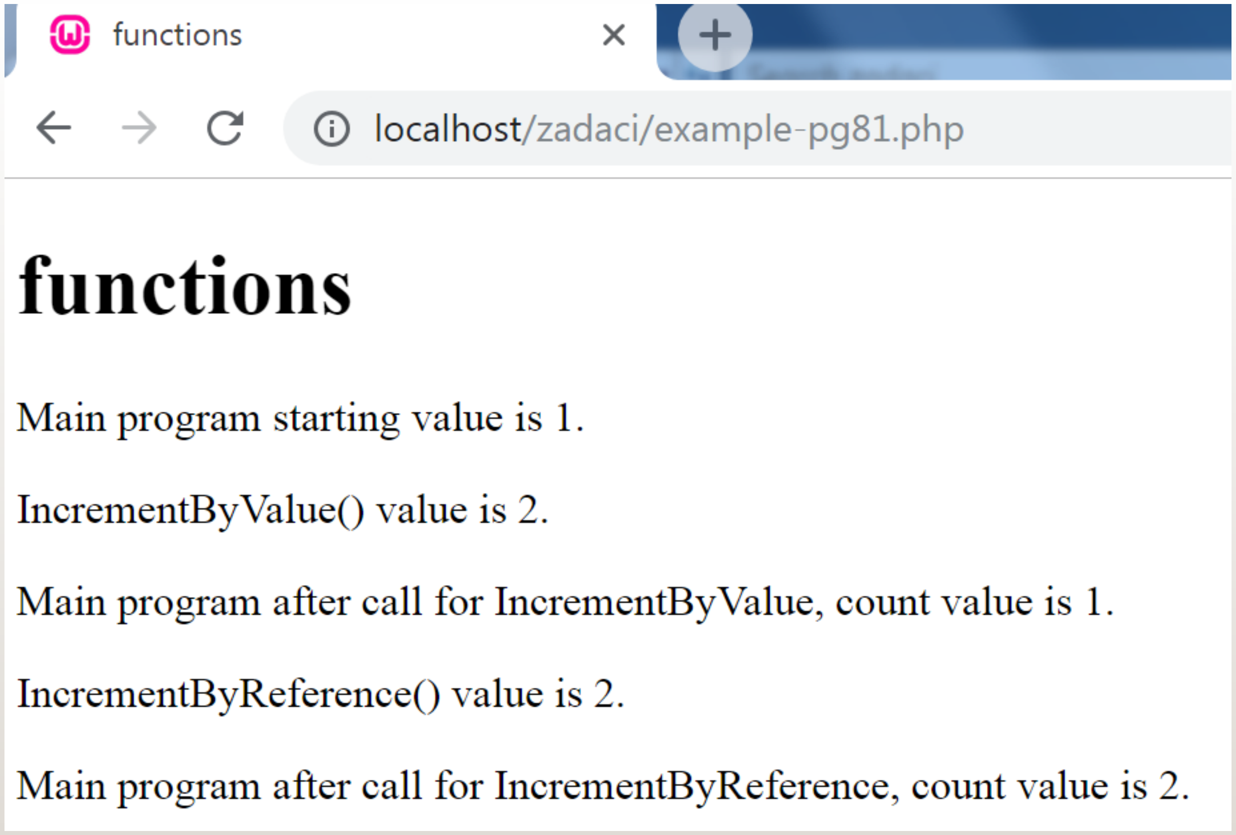
PASSING PARAMETERS TO A FUNCTION - EXAMPLE

```
<?php
function IncrementByValue($CountByValue) {
   ++$CountByValue;
   echo "<p>IncrementByValue() value is $CountByValue.</p>"; };

function IncrementByReference(&$CountByReference) {
   ++$CountByReference;
   echo "<p>IncrementByReference() value is $CountByReference. </p>";};

$Count = 1;
echo "<p>Main program starting value is $Count.</p>";
IncrementByValue($Count);
echo "<p>Main program after call for IncrementByValue, count value is$Count. </p>";
IncrementByReference($Count);
echo "<p>Main program after call for IncrementByReference, count value is $Count.
</p>";
?>
```

PASSING PARAMETERS TO A FUNCTION – EXAMPLE OUTPUT



```
functions

Main program starting value is 1.

IncrementByValue() value is 2.

Main program after call for IncrementByValue, count value is 1.

IncrementByReference() value is 2.

Main program after call for IncrementByReference, count value is 2.
```

FUNCTION – EXAMPLE
WHAT IS THE OUTPUT?

```
<?php
function add_some(&$text) {
   $text = $text."problem?";
}

$my_text = "Is there ";
echo "<p>",$my_text, "</p>";
add_some($my_text);
echo "<p>",$my_text, "</p>";

?>
```

FUNCTION PARAMETERS

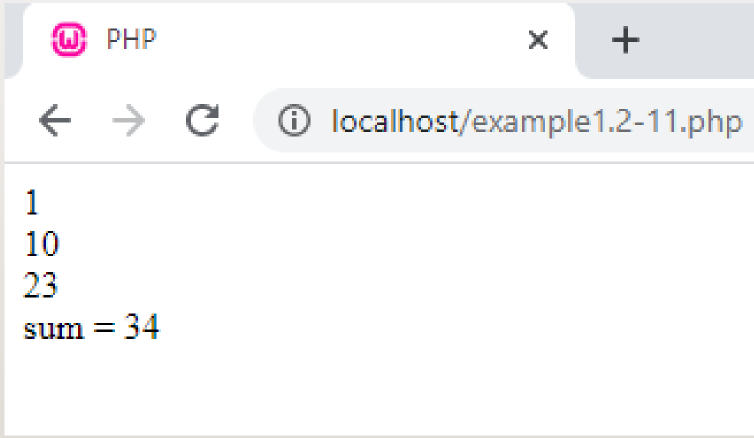
- PHP supports variable-length parameter lists - to define a function (variadic function) that accepts an arbitrary number of arguments
- Meaning you can pass 0, 1 or n number of arguments in function
- Parameter lists include ... token preceding the last (or only) parameter in the function definition
  - passed arguments will be converted into an array

FUNCTION PARAMETERS

```
<?php
function sum(...$numbers) {
    $acc = 0;
    foreach ($numbers as $n) {
        echo $n, "<br \>";
        $acc += $n;
    }
    return $acc;
}

$sum = sum(1, 10, 23);
echo "sum = $sum";

?>
```



```
1
10
23
sum = 34
```

FUNCTION PARAMETERS - OPTIONAL PARAMETERS/DEFAULT VALUES

- PHP supports optional parameters
- A function may define default values for parameters
- The default value is used only when the parameter is not specified
  - If null value is passed the default value is omitted
- Optional parameters should be specified after any required parameters

FUNCTION PARAMETERS - OPTIONAL PARAMETERS/DEFAULT VALUES

```
<?php
function makecoffee($type = "cappuccino") {
  return "<p>Making a cup of $type. </p>";
};

echo makecoffee();
echo makecoffee(null);
echo makecoffee("espresso");

?>
```

FUNCTION PARAMETERS – OPTIONAL TYPE DECLARATIONS

- Optional type declarations (data type hints) were introduced in PHP7+
- With this we can restrict the type of information passed into and out of a function
  - the specific data type can precede the parameter in the function definition
  - can specify the data type that a function returns

FUNCTION PARAMETERS – OPTIONAL TYPE DECLARATIONS

```
<?php
function double_num(int $number) : int {
      return $number *= 2;
}
$num = 5;
echo '$num =', $num, '<br>';
echo 'double_num returns ', double_num($num), '<br>';

$num = 4.8;
echo '$num =', $num, '<br>'; //Deprecated: Implicit conversion
                             //from float 4.8 to int loses precision –
                             //warning will be displayed, but will return 8
echo 'double_num returns ', double_num($num), '<br>';
?>
```

If declare(strict_types=1) is used the error will be -> Fatal error: Uncaught TypeError: double_num(): Argument #1 ($number) must be of type int, float given

strict_types declaration must be the very first statement in the script

FUNCTION PARAMETERS - SPECIFYING MULTIPLE DATA TYPES

- PHP 8 now permits union types
- Union types allow to combine two or more data types for function parameters or function return value

```
<?php
declare(strict_types=1);
function double_num(int|float $number) : int|float {
    return $number *= 2;
}

$num = 4.8;
echo '$num =', $num, '<br>';
echo 'double_num returns ', double_num($num), '<br>';
?>
```

FUNCTION PARAMETERS – NAMED ARGUMENTS

- Named Arguments (introduced in PHP 8) support function call by setting the parameters by their name
- With Named Arguments the order of the arguments is not important, as long as all the required parameters are passed
  - useful to skip over multiple optional parameters

FUNCTION PARAMETERS – NAMED ARGUMENTS

```
<?php
function make_sentence($name, $activity="no activity", $hours="") {
    return "Hi $name, you have $activity for $hours hrs";
}

echo make_sentence("John"), '<br>';
echo make_sentence("John", "swimming"), '<br>';
echo make_sentence("John", "swimming", "1"), '<br>';

echo make_sentence("John", activity: "hiking"), '<br>';

echo make_sentence(activity: "hiking", name: "John",  hours: "8"), '<br>';
echo make_sentence("John", hours: "8", activity: "hiking"), '<br>';

echo make_sentence("John", hours: "5"), '<br>';
echo make_sentence(hours: "5", name: "John"), '<br>';

?>
```

UNDERSTANDING VARIABLE SCOPE

- **Variable scope** is where in your program a declared variable can be used
- A variable's scope can be either global or local
- A **global variable** is one that is declared outside a function and is available to all parts of your program
- A **local variable** is declared inside a function and is only available within the function in which it is declared

THE `GLOBAL` KEYWORD

- You must declare a global variable with the `global` keyword inside a function definition to make the variable available within the scope of that function

```
<?php
    $GlobalVariable = "this is my value";

    function scopeExample() {
        global $GlobalVariable;
        echo "<p>$GlobalVariable</p>";
        $GlobalVariable = "if I change it";
    }
    scopeExample();
    echo "<p>$GlobalVariable</p>";
?>
```

MAKING DECISIONS

- **Decision making** or **flow control** is the process of determining the order in which statements execute in a program
- The special types of PHP statements used for making decisions are called **decision-making statements** or **decision-making structures**

`IF` STATEMENTS

- Used to execute specific programming code if the evaluation of a conditional expression returns a value of `TRUE`
- The syntax for a simple `if` statement is:

```
if (conditional expression)
  statement;
```

`IF` STATEMENTS (CONTINUED)

- Contains three parts:
  - the keyword `if`
  - a conditional expression enclosed within parentheses
  - the executable statements
- A **command block** is a group of statements contained within a set of braces
- Each command block must have an opening brace ({) and a closing brace (})

`IF` STATEMENTS (CONTINUED)

```
$ExampleVar = 5;
if ($ExampleVar == 5) {   // condition evaluates to 'TRUE'
     echo " <p>The condition evaluates to true.</p> ";
     echo '<p>$ExampleVar is equal to ', " $ExampleVar.</p> ";
     echo " <p>Each of these lines will be printed.</p> ";
}
echo " <p>This statement always executes after the if
                                            statement.</p> ";
```

`IF...ELSE` STATEMENTS

- An `if` statement that includes an `else` clause is called an **`if...else` statement**
- An `else` clause executes when the condition in an `if...else` statement evaluates to `FALSE`
- The syntax for an `if...else` statement is:

```
if (conditional expression)
        statement;
else
        statement;
```

`IF...ELSE` STATEMENTS (CONTINUED)

- An `if` statement can be constructed without the `else` clause
- The `else` clause can only be used with an `if` statement

```
$Today = "Tuesday";
 if ($Today == "Monday")
     echo "<p>Today is Monday</p>";
 else
     echo "<p>Today is not Monday</p>";
```

NESTED `IF` AND `IF...ELSE` STATEMENTS

- When one decision-making statement is contained within another decision-making statement, they are referred to as nested **decision-making structures**

```
if ($SalesTotal >= 50)
    if ($SalesTotal <= 100)
        echo "<p>The sales total is between 50 and
                                 100, inclusive.</p>";
```

- *Check the* ***<=> rocket ship operator*** *(available in PHP7+)*

`SWITCH` STATEMENTS

- Control program flow by executing a specific set of statements depending on the value of an expression
- Compare the value of an expression to a value contained within a special statement called a **case label**
- A **case label** is a specific value that contains one or more statements that execute if the value of the case label matches the value of the switch statement's expression

`SWITCH` STATEMENTS (CONTINUED)

- Consist of the following components:
  - The `switch` keyword
  - An expression
  - An opening brace
  - One or more `case` labels (witch can be different data types)
  - The executable statements
  - The `break` keyword
  - A `default` label
  - A closing brace

`SWITCH` STATEMENTS (CONTINUED)

- The syntax for the `switch` statement is:

```
switch (expression) {
        case label:
              statement(s);
              break;
        case label:
              statement(s);
              break;
        ...
        default:
              statement(s);
              break;
}
```

`SWITCH` STATEMENTS (CONTINUED)

- A `case` label consists of:
  - The keyword **case**
  - A literal value or variable name
  - A colon (:)
- A `case` label can be followed by a single statement or multiple statements
- Multiple statements for a `case` label do not need to be enclosed within a command block

`SWITCH` STATEMENTS (CONTINUED)

- The **`default` label** contains statements that execute when the value returned by the `switch` statement expression does not match a `case` label
- A `default` label consists of the keyword `default` followed by a colon (:)

REPEATING CODE

- A **loop statement** is a control structure that repeatedly executes a statement or a series of statements while a specific condition is `TRUE` or until a specific condition becomes `TRUE`
- There are four types of loop statements:
  - `while` statements
  - `do...while` statements
  - `for` statements
  - `foreach` statements

`WHILE` STATEMENTS

- Tests the condition prior to executing the series of statements at each iteration of the loop
- The syntax for the `while` statement is:

```
while (conditional expression) {
        statement(s);
}
```

- As long as the conditional expression evaluates to `TRUE`, the statement or command block that follows executes repeatedly

`WHILE` STATEMENTS (CONTINUED)

- Each repetition of a looping statement is called an **iteration**
- A `while` statement keeps repeating until its conditional expression evaluates to `FALSE`
- A **counter** is a variable that increments or decrements with each iteration of a loop statement

`WHILE` STATEMENTS – WHAT IS THE OUTPUT?

```
(ex. 1)
$Count = 1;
while ($Count <= 5) {
        echo "$Count<br />";
        ++$Count;
}
echo "<p>You have printed 5 numbers.</p>";
```

WHILE STATEMENTS - WHAT IS THE OUTPUT?

```
(ex. 2)
$Count = 10;
while ($Count > 0) {
    echo "$Count<br />";
    --$Count;
}
echo "<p>We have liftoff. </p>";
------
```

```
(ex. 3)
$Count = 1;
while ($Count <= 100) {
        echo "$Count<br />";
        $Count *= 2;
}
```

`WHILE` STATEMENTS – INFINITE LOOP

- In an **infinite loop**, a loop statement never ends because its conditional expression is never `FALSE`

```
$Count = 1;
while ($Count <= 10) {
     echo " The number is $Count ";
}
```

DO...WHILE STATEMENTS

- Test the condition after executing a series of statements then repeats the execution as long as a given conditional expression evaluates to TRUE
- The syntax for the do...while statement is:

```
do {
        statement(s);
} while (conditional expression);
```

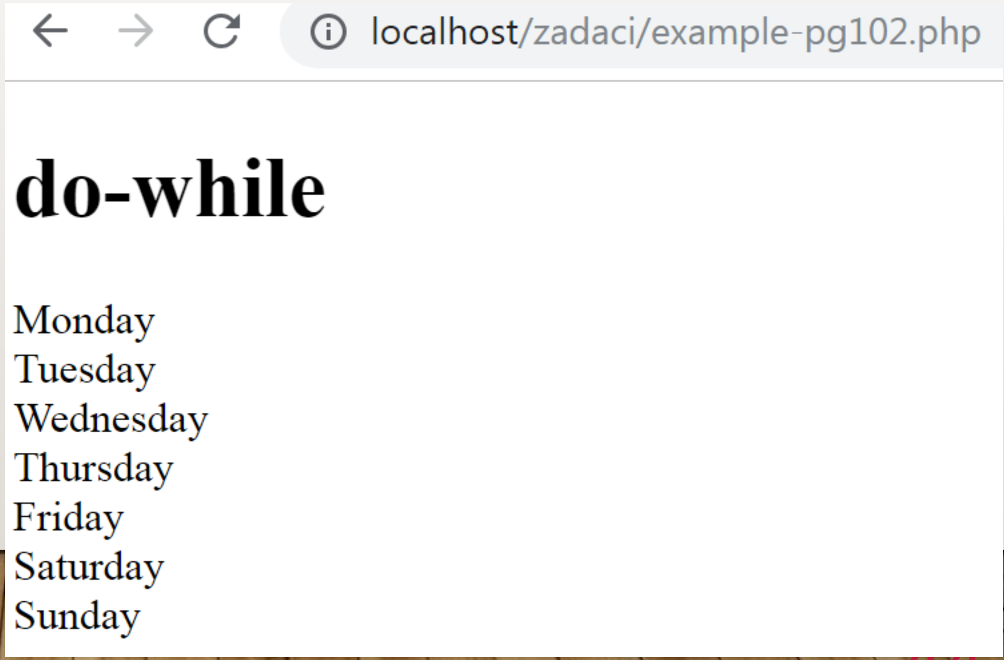
`DO...WHILE` STATEMENTS

- `do...while` statements always execute once, before a conditional expression is evaluated

```
$Count = 2;
do {
   echo "<p>The count is equal to $Count</p>";
   ++$Count;
} while ($Count < 2);
```

`DO...WHILE` STATEMENTS - EXAMPLE

```
$DaysOfWeek = array(" Monday ", " Tuesday ", " Wednesday ", "
                Thursday ", " Friday ", " Saturday ", " Sunday ");
$Count = 0;
do {
            echo $DaysOfWeek[$Count], "<br />";
            ++$Count;
} while ($Count < 7);
```



```
do-while

Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Sunday
```

`FOR` STATEMENTS

- Combine the initialize, conditional expression, and update portions of a loop into a single statement
- Repeat a statement or a series of statements as long as a given conditional expression evaluates to `TRUE`
- If the conditional expression evaluates to `TRUE`, the `for` statement executes and continues to execute repeatedly until the conditional expression evaluates to `FALSE`

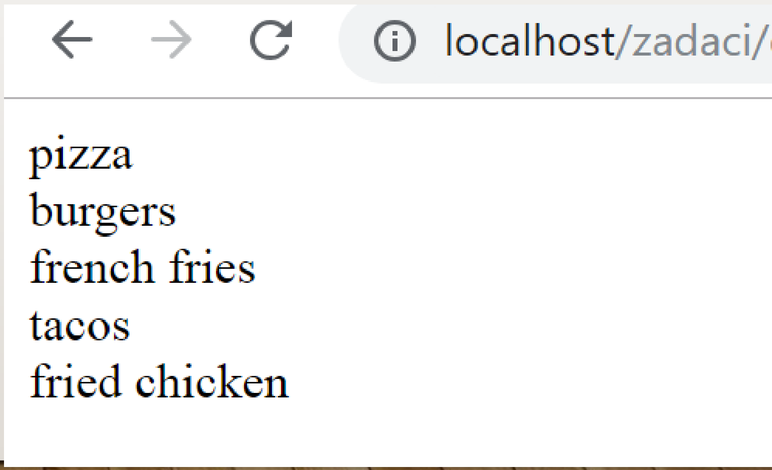
`FOR` STATEMENTS (CONTINUED)

- Can also include code that initializes a counter and changes its value with each iteration
- The syntax of the `for` statement is:

```
for (counter declaration and initialization;
  condition; update statement) {
      statement(s);
    }
```

FOR STATEMENTS - EXAMPLE

```
$FastFoods = array("pizza", "burgers", "french fries",
  "tacos", "fried chicken");
for ($Count = 0; $Count < 5; ++$Count) {
     echo $FastFoods[$Count], "<br />";
}
```



```
pizza
burgers
french fries
tacos
fried chicken
```

`FOREACH` STATEMENTS

- Used to iterate or loop through the elements in an array
- Do not require a counter, instead, you specify an array expression within a set of parentheses following the `foreach` keyword
- The syntax for the `foreach` statement is:

```
foreach ($array_name as $variable_name) {
        statements;
}
---
foreach ($array_name as $index_name => $variable_name) {
        statements;
}
```

FOREACH STATEMENTS - EXAMPLE

```
$DaysOfWeek = array("Monday", "Tuesday",
          "Wednesday", "Thursday", "Friday",
                   "Saturday", "Sunday");

foreach ($DaysOfWeek as $Day) {
     echo "<p>$Day</p>";
}

foreach ($DaysOfWeek as $DayNum => $Day) {
     echo "<p>$DayNum is $Day</p>";
}
```

INCLUDING FILES

- The `include` and `require` statements reuse content by allowing you to insert the content of an external file on multiple Web pages
  - The `include` statement generates a warning if the include file cannot be found
  - The `require` statement halts the processing of the Web page and displays an error if the include file cannot be found
- The `include_once` and `require_once` statements assure that the external file is added to the script only one time

INCLUDING FILES

- The file path can be either absolute or relative to the current document

```
require 'C:/wamp64/www/my_folder/script.php';

include('my_folder/script.php');
// include 'my_folder/script.php';
```

- When using a relative file path, it’s recommended to use `./` to indicate that the path begins in the current folder

```
include('./my_folder/script.php');
```