Strings ¶

A string is series of characters, where a character is the same as a byte. This means that PHP only supports a 256-character set, and hence does not offer native Unicode support. See details of the string type.

> **Note**: On 32-bit builds, a string can be as large as up to 2GB (2147483647 bytes maximum)

Syntax ¶

A string literal can be specified in four different ways:

- single quoted
- double quoted
- heredoc syntax
- nowdoc syntax

Single quoted ¶

The simplest way to specify a string is to enclose it in single quotes (the character `'`).

To specify a literal single quote, escape it with a backslash (`\`). To specify a literal backslash, double it (`\\`). All other instances of backslash will be treated as a literal backslash: this means that the other escape sequences you might be used to, such as `\r` or `\n`, will be output literally as specified rather than having any special meaning.

> **Note**: Unlike the double-quoted and heredoc syntaxes, variables and escape sequences for special characters will *not* be expanded when they occur in single quoted strings.

```php
<?php
echo 'this is a simple string';

echo 'You can also have embedded newlines in
strings this way as it is
okay to do';

// Outputs: Arnold once said: "I'll be back"
echo 'Arnold once said: "I\'ll be back"';

// Outputs: You deleted C:\*.*?
echo 'You deleted C:\\*.*?';

// Outputs: You deleted C:\*.*?
echo 'You deleted C:\*.*?';

// Outputs: This will not expand: \n a newline
echo 'This will not expand: \n a newline';

// Outputs: Variables do not $expand $either
echo 'Variables do not $expand $either';
?>
```

Double quoted ¶

If the string is enclosed in double-quotes ("), PHP will interpret the following escape sequences for special characters:

Escaped characters

Sequence	Meaning
`\n`	linefeed (LF or 0x0A (10) in ASCII)

Sequence	Meaning
`\r`	carriage return (CR or 0x0D (13) in ASCII)
`\t`	horizontal tab (HT or 0x09 (9) in ASCII)
`\v`	vertical tab (VT or 0x0B (11) in ASCII)
`\e`	escape (ESC or 0x1B (27) in ASCII)
`\f`	form feed (FF or 0x0C (12) in ASCII)
`\\`	backslash
`\$`	dollar sign
`\"`	double-quote
`\[0-7]{1,3}`	the sequence of characters matching the regular expression is a character in octal notation, which silently overflows to fit in a byte (e.g. "\400" === "\000")
`\x[0-9A-Fa-f]{1,2}`	the sequence of characters matching the regular expression is a character in hexadecimal notation
`\u{[0-9A-Fa-f]+}`	the sequence of characters matching the regular expression is a Unicode codepoint, which will be output to the string as that codepoint's UTF-8 representation

As in single quoted strings, escaping any other character will result in the backslash being printed too.

The most important feature of double-quoted strings is the fact that variable names will be expanded. Seestring parsing for details.

Heredoc ¶

A third way to delimit strings is the heredoc syntax: `<<<`. After this operator, an identifier is provided, then a newline. The string itself follows, and then the same identifier again to close the quotation.

The closing identifier *must* begin in the first column of the line. Also, the identifier must follow the same naming rules as any other label in PHP: it must contain only alphanumeric characters and underscores, and must start with a non-digit character or underscore.

Warning

It is very important to note that the line with the closing identifier must contain no other characters, except a semicolon (`;`). That means especially that the identifier *may not be indented*, and there may not be any spaces or tabs before or after the semicolon. It's also important to realize that the first character before the closing identifier must be a newline as defined by the local operating system. This is `\n` on UNIX systems, including macOS. The closing delimiter must also be followed by a newline.

If this rule is broken and the closing identifier is not "clean", it will not be considered a closing identifier, and PHP will continue looking for one. If a proper closing identifier is not found before the end of the current file, a parse error will result at the last line.

Example #1 Invalid example

```
<?php
class foo {
    public $bar = <<<EOT
bar
    EOT;
}
// Identifier must not be indented
?>
```

Example #2 Valid example

```
<?php
class foo {
```

```
    public $bar = <<<EOT
bar
EOT;
}
?>
```

Heredocs containing variables can not be used for initializing class properties.

Heredoc text behaves just like a double-quoted string, without the double quotes. This means that quotes in a heredoc do not need to be escaped, but the escape codes listed above can still be used. Variables are expanded, but the same care must be taken when expressing complex variables inside a heredoc as with strings.

Example #3 Heredoc string quoting example

```
<?php
$str = <<<EOD
Example of string
spanning multiple lines
using heredoc syntax.
EOD;

/* More complex example, with variables. */
class foo
{
    var $foo;
    var $bar;

    function __construct()
    {
        $this->foo = 'Foo';
        $this->bar = array('Bar1', 'Bar2', 'Bar3');
    }
}

$foo = new foo();
$name = 'MyName';

echo <<<EOT
My name is "$name". I am printing some $foo->foo.
Now, I am printing some {$foo->bar[1]}.
This should print a capital 'A': \x41
EOT;
?>
```

The above example will output:

```
My name is "MyName". I am printing some Foo.
Now, I am printing some Bar2.
This should print a capital 'A': A
```

It is also possible to use the Heredoc syntax to pass data to function arguments:

Example #4 Heredoc in arguments example

```
<?php
var_dump(array(<<<EOD
foobar!
EOD
));
?>
```

It's possible to initialize static variables and class properties/constants using the Heredoc syntax:

Example #5 Using Heredoc to initialize static values

```php
<?php
// Static variables
function foo()
{
    static $bar = <<<LABEL
Nothing in here...
LABEL;
}

// Class properties/constants
class foo
{
    const BAR = <<<FOOBAR
Constant example
FOOBAR;

    public $baz = <<<FOOBAR
Property example
FOOBAR;
}
?>
```

The opening Heredoc identifier may optionally be enclosed in double quotes:

Example #6 Using double quotes in Heredoc

```php
<?php
echo <<<"FOOBAR"
Hello World!
FOOBAR;
?>
```

Nowdoc ¶

Nowdocs are to single-quoted strings what heredocs are to double-quoted strings. A nowdoc is specified similarly to a heredoc, but *no parsing is done* inside a nowdoc. The construct is ideal for embedding PHP code or other large blocks of text without the need for escaping. It shares some features in common with the SGML `<![CDATA[ ]]>` construct, in that it declares a block of text which is not for parsing.

A nowdoc is identified with the same `<<<` sequence used for heredocs, but the identifier which follows is enclosed in single quotes, e.g. `<<<'EOT'`. All the rules for heredoc identifiers also apply to nowdoc identifiers, especially those regarding the appearance of the closing identifier.

Example #7 Nowdoc string quoting example

```php
<?php
echo <<<'EOD'
Example of string spanning multiple lines
using nowdoc syntax. Backslashes are always treated literally,
e.g. \\ and \'.
EOD;
```

The above example will output:

```
Example of string spanning multiple lines
using nowdoc syntax. Backslashes are always treated literally,
e.g. \\ and \'.
```

Example #8 Nowdoc string quoting example with variables

```php
<?php
class foo
{
    public $foo;
    public $bar;
```

```
    function __construct()
    {
        $this->foo = 'Foo';
        $this->bar = array('Bar1', 'Bar2', 'Bar3');
    }
}

$foo = new foo();
$name = 'MyName';

echo <<<'EOT'
My name is "$name". I am printing some $foo->foo.
Now, I am printing some {$foo->bar[1]}.
This should not print a capital 'A': \x41
EOT;
?>
```

The above example will output:

```
My name is "$name". I am printing some $foo->foo.
Now, I am printing some {$foo->bar[1]}.
This should not print a capital 'A': \x41
```

Example #9 Static data example

```
<?php
class foo {
    public $bar = <<<'EOT'
bar
EOT;
}
?>
```

Variable parsing ¶

When a string is specified in double quotes or with heredoc, variables are parsed within it.

There are two types of syntax: a simple one and a complex one. The simple syntax is the most common and convenient. It provides a way to embed a variable, an array value, or an object property in a string with a minimum of effort.

The complex syntax can be recognised by the curly braces surrounding the expression.

Simple syntax

If a dollar sign (`$`) is encountered, the parser will greedily take as many tokens as possible to form a valid variable name. Enclose the variable name in curly braces to explicitly specify the end of the name.

```
<?php
$juice = "apple";

echo "He drank some $juice juice.".PHP_EOL;
// Invalid. "s" is a valid character for a variable name, but the variable is $juice.
echo "He drank some juice made of $juices.";
// Valid. Explicitly specify the end of the variable name by enclosing it in braces:
echo "He drank some juice made of ${juice}s.";
?>
```

The above example will output:

```
He drank some apple juice.
He drank some juice made of .
He drank some juice made of apples.
```

Similarly, an array index or an object property can be parsed. With array indices, the closing square bracket (]) marks the end of the index. The same rules apply to object properties as to simple variables.

Example #10 Simple syntax example

```
<?php
$juices = array("apple", "orange", "koolaid1" => "purple");

echo "He drank some $juices[0] juice.".PHP_EOL;
echo "He drank some $juices[1] juice.".PHP_EOL;
echo "He drank some $juices[koolaid1] juice.".PHP_EOL;

class people {
    public $john = "John Smith";
    public $jane = "Jane Smith";
    public $robert = "Robert Paulsen";

    public $smith = "Smith";
}

$people = new people();

echo "$people->john drank some $juices[0] juice.".PHP_EOL;
echo "$people->john then said hello to $people->jane.".PHP_EOL;
echo "$people->john's wife greeted $people->robert.".PHP_EOL;
echo "$people->robert greeted the two $people->smiths."; // Won't work
?>
```

The above example will output:

```
He drank some apple juice.
He drank some orange juice.
He drank some purple juice.
John Smith drank some apple juice.
John Smith then said hello to Jane Smith.
John Smith's wife greeted Robert Paulsen.
Robert Paulsen greeted the two .
```

As of PHP 7.1.0 also *negative* numeric indices are supported.

Example #11 Negative numeric indices

```
<?php
$string = 'string';
echo "The character at index -2 is $string[-2].", PHP_EOL;
$string[-3] = 'o';
echo "Changing the character at index -3 to o gives $string.", PHP_EOL;
?>
```

The above example will output:

```
The character at index -2 is n.
Changing the character at index -3 to o gives strong.
```

For anything more complex, you should use the complex syntax.

Complex (curly) syntax

This isn't called complex because the syntax is complex, but because it allows for the use of complex expressions.

Any scalar variable, array element or object property with a string representation can be included via this syntax. Simply write the expression the same way as it would appear outside the string, and then wrap it in {and }. Since { can not be escaped, this syntax will only be recognised when the $ immediately follows the {. Use {\$ to get a literal {$. Some examples to make it clear:

```php
<?php
// Show all errors
error_reporting(E_ALL);

$great = 'fantastic';

// Won't work, outputs: This is { fantastic}
echo "This is { $great}";

// Works, outputs: This is fantastic
echo "This is {$great}";

// Works
echo "This square is {$square->width}00 centimeters broad.";


// Works, quoted keys only work using the curly brace syntax
echo "This works: {$arr['key']}";


// Works
echo "This works: {$arr[4][3]}";

// This is wrong for the same reason as $foo[bar] is wrong  outside a string.
// In other words, it will still work, but only because PHP first looks for a
// constant named foo; an error of level E_NOTICE (undefined constant) will be
// thrown.
echo "This is wrong: {$arr[foo][3]}";

// Works. When using multi-dimensional arrays, always use braces around arrays
// when inside of strings
echo "This works: {$arr['foo'][3]}";

// Works.
echo "This works: " . $arr['foo'][3];

echo "This works too: {$obj->values[3]->name}";

echo "This is the value of the var named $name: {${$name}}";

echo "This is the value of the var named by the return value of getName(): {${getName()}}";

echo "This is the value of the var named by the return value of \$object->getName(): {${$object->getName()}}";

// Won't work, outputs: This is the return value of getName(): {getName()}
echo "This is the return value of getName(): {getName()}";
?>
```

It is also possible to access class properties using variables within strings using this syntax.

```php
<?php
class foo {
    var $bar = 'I am bar.';
}

$foo = new foo();
$bar = 'bar';
$baz = array('foo', 'bar', 'baz', 'quux');
echo "{$foo->$bar}\n";
echo "{$foo->{$baz[1]}}\n";
?>
```

The above example will output:

```
I am bar.
I am bar.
```

> **Note**:
>
> The value accessed from functions, method calls, static class variables, and class constants inside `{$}` will be interpreted as the name of a variable in the scope in which the string is defined. Using single curly braces (`{}`) will not work for accessing the return values of functions or methods or the values of class constants or static class variables.

```
<?php
// Show all errors.
error_reporting(E_ALL);

class beers {
    const softdrink = 'rootbeer';
    public static $ale = 'ipa';
}

$rootbeer = 'A & W';
$ipa = 'Alexander Keith\'s';

// This works; outputs: I'd like an A & W
echo "I'd like an {${beers::softdrink}}\n";

// This works too; outputs: I'd like an Alexander Keith's
echo "I'd like an {${beers::$ale}}\n";
?>
```

String access and modification by character ¶

Characters within strings may be accessed and modified by specifying the zero-based offset of the desired character after the string using square array brackets, as in *$str[42]*. Think of a string as an array of characters for this purpose. The functions substr() and substr_replace() can be used when you want to extract or replace more than 1 character.

> **Note**: As of PHP 7.1.0, negative string offsets are also supported. These specify the offset from the end of the string. Formerly, negative offsets emitted **`E_NOTICE`** for reading (yielding an empty string) and **`E_WARNING`** for writing (leaving the string untouched).

> **Note**: Prior to PHP 8.0.0, strings could also be accessed using braces, as in *$str{42}*, for the same purpose. This curly brace syntax was deprecated as of PHP 7.4.0 and no longer supported as of PHP 8.0.0.

Warning

Writing to an out of range offset pads the string with spaces. Non-integer types are converted to integer. Illegal offset type emits **`E_WARNING`**. Only the first character of an assigned string is used. As of PHP 7.1.0, assigning an empty string throws a fatal error. Formerly, it assigned a NULL byte.

Warning

Internally, PHP strings are byte arrays. As a result, accessing or modifying a string using array brackets is not multi-byte safe, and should only be done with strings that are in a single-byte encoding such as ISO-8859-1.

> **Note**: As of PHP 7.1.0, applying the empty index operator on an empty string throws a fatal error. Formerly, the empty string was silently converted to an array.

Example #12 Some string examples

```
<?php
// Get the first character of a string
$str = 'This is a test.';
$first = $str[0];
```

```php
// Get the third character of a string
$third = $str[2];

// Get the last character of a string.
$str = 'This is still a test.';
$last = $str[strlen($str)-1];

// Modify the last character of a string
$str = 'Look at the sea';
$str[strlen($str)-1] = 'e';

?>
```

String offsets have to either be integers or integer-like strings, otherwise a warning will be thrown.

Example #13 Example of Illegal String Offsets

```php
<?php
$str = 'abc';

var_dump($str['1']);
var_dump(isset($str['1']));

var_dump($str['1.0']);
var_dump(isset($str['1.0']));

var_dump($str['x']);
var_dump(isset($str['x']));

var_dump($str['1x']);
var_dump(isset($str['1x']));
?>
```

The above example will output:

```
string(1) "b"
bool(true)

Warning: Illegal string offset '1.0' in /tmp/t.php on line 7
string(1) "b"
bool(false)

Warning: Illegal string offset 'x' in /tmp/t.php on line 9
string(1) "a"
bool(false)
string(1) "b"
bool(false)
```

> **Note**:
>
> Accessing variables of other types (not including arrays or objects implementing the appropriate interfaces) using `[]` or `{}` silently returns `null`.
>
> **Note**:
>
> Characters within string literals can be accessed using `[]` or `{}`.
>
> **Note**:
>
> Accessing characters within string literals using the `{}` syntax has been deprecated in PHP 7.4. This has been removed in PHP 8.0.

Useful functions and operators ¶

Strings may be concatenated using the '.' (dot) operator. Note that the '+' (addition) operator will *not* work for this. See String operators for more information.

There are a number of useful functions for string manipulation.

See the string functions section for general functions, and the Perl-compatible regular expression functionsfor advanced find & replace functionality.

There are also functions for URL strings, and functions to encrypt/decrypt strings (Sodium and Hash).

Finally, see also the character type functions.

Converting to string ¶

A value can be converted to a string using the `(string)` cast or the strval() function. String conversion is automatically done in the scope of an expression where a string is needed. This happens when using the echoor print functions, or when a variable is compared to a string. The sections on Types and Type Juggling will make the following clearer. See also the settype() function.

A bool **`true`** value is converted to the string `"1"`. bool **`false`** is converted to `""` (the empty string). This allows conversion back and forth between bool and string values.

An int or float is converted to a string representing the number textually (including the exponent part for floats). Floating point numbers can be converted using exponential notation (`4.1E+6`).

> **Note**:
>
> As of PHP 8.0.0, the decimal point character is always `.`. Prior to PHP 8.0.0, the decimal point character is defined in the script's locale (category LC_NUMERIC). See the setlocale() function.

Arrays are always converted to the string `"Array"`; because of this, echo and print can not by themselves show the contents of an array. To view a single element, use a construction such as `echo $arr['foo']`. See below for tips on viewing the entire contents.

In order to convert objects to string magic method __toString must be used.

Resources are always converted to strings with the structure `"Resource id #1"`, where `1` is the resource number assigned to the resource by PHP at runtime. While the exact structure of this string should not be relied on and is subject to change, it will always be unique for a given resource within the lifetime of a script being executed (ie a Web request or CLI process) and won't be reused. To get a resource's type, use the get_resource_type() function.

`null` is always converted to an empty string.

As stated above, directly converting an array, object, or resource to a string does not provide any useful information about the value beyond its type. See the functions print_r() and var_dump() for more effective means of inspecting the contents of these types.

Most PHP values can also be converted to strings for permanent storage. This method is called serialization, and is performed by the serialize() function.

Details of the String Type ¶

The string in PHP is implemented as an array of bytes and an integer indicating the length of the buffer. It has no information about how those bytes translate to characters, leaving that task to the programmer. There are no limitations on the values the string can be composed of; in particular, bytes with value `0` (“NUL bytes”) are allowed anywhere in the string (however, a few functions, said in this manual not to be “binary safe”, may hand off the strings to libraries that ignore data after a NUL byte.)

This nature of the string type explains why there is no separate “byte” type in PHP – strings take this role. Functions that return no textual data – for instance, arbitrary data read from a network socket – will still return strings.

Given that PHP does not dictate a specific encoding for strings, one might wonder how string literals are encoded. For instance, is the string `"á"` equivalent to `"\xE1"` (ISO-8859-1), `"\xC3\xA1"` (UTF-8, C form),`"\x61\xCC\x81"` (UTF-8, D form) or any other possible representation? The answer is that string will be encoded in whatever fashion it is encoded in the script file. Thus, if the script is written in ISO-8859-1, the string will be encoded in ISO-8859-1 and so on. However, this does not apply if Zend Multibyte is enabled; in that case, the script may be written in an arbitrary encoding (which is explicitly declared or is detected) and then converted to a certain internal encoding, which is then the encoding that will be used for the string literals. Note that there are some constraints on the encoding of the script (or on the internal encoding, should Zend Multibyte be enabled) – this almost always means that this encoding should be a compatible superset of ASCII, such as UTF-8 or ISO-8859-1. Note, however, that state-dependent encodings where the same byte values can be used in initial and non-initial shift states may be problematic.

Of course, in order to be useful, functions that operate on text may have to make some assumptions about how the string is encoded. Unfortunately, there is much variation on this matter throughout PHP’s functions:

- Some functions assume that the string is encoded in some (any) single-byte encoding, but they do not need to interpret those bytes as specific characters. This is case of, for instance, substr(), strpos(), strlen() or strcmp(). Another way to think of these functions is that operate on memory buffers, i.e., they work with bytes and byte offsets.
- Other functions are passed the encoding of the string, possibly they also assume a default if no such information is given. This is the case of htmlentities() and the majority of the functions in the mbstringextension.
- Others use the current locale (see setlocale()), but operate byte-by-byte. This is the case of strcasecmp(), strtoupper() and ucfirst(). This means they can be used only with single-byte encodings, as long as the encoding is matched by the locale. For instance `strtoupper("á")` may return `"Á"` if the locale is correctly set and `á` is encoded with a single byte. If it is encoded in UTF-8, the correct result will not be returned and the resulting string may or may not be returned corrupted, depending on the current locale.
- Finally, they may just assume the string is using a specific encoding, usually UTF-8. This is the case of most functions in the intl extension and in the PCRE extension (in the last case, only when the `u`modifier is used). Although this is due to their special purpose, the function utf8_decode() assumes a UTF-8 encoding and the function utf8_encode() assumes an ISO-8859-1 encoding.

Ultimately, this means writing correct programs using Unicode depends on carefully avoiding functions that will not work and that most likely will corrupt the data and using instead the functions that do behave correctly, generally from the intl and mbstring extensions. However, using functions that can handle Unicode encodings is just the beginning. No matter the functions the language provides, it is essential to know the Unicode specification. For instance, a program that assumes there is only uppercase and lowercase is making a wrong assumption.