

Strings



- [Syntax](#)
 - [Single quoted](#)
 - [Double quoted](#)
 - [Heredoc](#)
 - [Nowdoc](#)
 - [String interpolation](#)
 - [Basic syntax](#)
 - [Advanced \(curly\) syntax](#)
 - [String access and modification by character](#)
 - [Useful functions and operators](#)
 - [Converting to string](#)
 - [Details of the String Type](#)
-

A `string` is a 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 up to 2GB (2147483647 bytes) in length.

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 `string`s.

Syntax Variants

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

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

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

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

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

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

// Outputs: Variables do not $expand $either
echo 'Variables do not $expand $either', PHP_EOL;
```

?>

Double quoted

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

Sequence	Meaning
<code>\n</code>	linefeed (LF or 0x0A (10) in ASCII)
<code>\r</code>	carriage return (CR or 0x0D (13) in ASCII)
<code>\t</code>	horizontal tab (HT or 0x09 (9) in ASCII)
<code>\v</code>	vertical tab (VT or 0x0B (11) in ASCII)
<code>\e</code>	escape (ESC or 0x1B (27) in ASCII)
<code>\f</code>	form feed (FF or 0x0C (12) in ASCII)
<code>\\</code>	backslash
<code>\\$</code>	dollar sign
<code>\"</code>	double-quote
<code>\[0-7]{1,3}</code>	Octal: the sequence of characters matching the regular expression <code>[0-7]{1,3}</code> is a character in octal notation (e.g. <code>"\101" === "A"</code>), which silently overflows to fit in a byte (e.g. <code>"\400" === "\000"</code>)
<code>\x[0-9A-Fa-f]{1,2}</code>	Hexadecimal: the sequence of characters matching the regular expression <code>[0-9A-Fa-f]{1,2}</code> is a character in hexadecimal notation (e.g. <code>"\x41" === "A"</code>)
<code>\u{[0-9A-Fa-f]+}</code>	Unicode: the sequence of characters matching the regular expression <code>[0-9A-Fa-f]+</code> is a Unicode codepoint, which will be output to the string as that codepoint's UTF-8 representation. The braces are required in the sequence. E.g. <code>"\u{41}" === "A"</code>

Escaped characters

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

The most important feature of double-quoted `string`s is the fact that variable names will be expanded. See [string interpolation](#) for details.

Heredoc

A third way to delimit `string`s is the heredoc syntax: `ENTITYENTITYENTITY`. 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 may be indented with spaces or tabs. Whitespace equal to the closing identifier's indentation is then removed from the beginning of every line in the heredoc string. Prior to PHP 7.3.0, the closing identifier *must* begin in the first column of the line.

Also, the closing 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.

Basic Heredoc example as of PHP 7.3.0

```
<?php
// closing identifier not indented
echo <<<END
    a
    b
    c
\n
END;

// closing identifier indented with 3 spaces; 3 spaces stripped from each line
echo <<<END
    a
    b
    c
END;
```

ENTITY

```
    a
    b
    c

a
```

```
b
c
```

If the closing identifier is indented further than any lines of the body, then a `ParseError` will be thrown:

Closing identifier must not be indented further than any lines of the body

```
<?php
echo <<<END
    a
    b
    c
    END;
```

ENTITY

```
Parse error: Invalid body indentation level (expecting an indentation level of at least 3) in
example.php on line 4
```

If the closing identifier is indented, tabs can be used as well. However, tabs and spaces *must not* be intermixed. The indentation of the body and the closing identifier must use one or the other consistently. If tabs and spaces are mixed, then a `ParseError` will be thrown. These whitespace constraints have been included because mixing tabs and spaces for indentation is harmful to legibility.

Different indentation for body and closing identifier

```
<?php
// All of the following code does not work.

// different indentation for body (spaces) and ending marker (tabs)
{
    echo <<<END
    a
        END;
}

// mixing spaces and tabs in body
```

```

{
    echo <<<END
        a
    END;
}

// mixing spaces and tabs in ending marker
{
    echo <<<END
        a
    END;
}

```

ENTITY

Parse error: Invalid indentation - tabs and spaces cannot be mixed in example.php line 8

The closing identifier for the body string is not required to be followed by a semicolon or newline. For example, the following code is allowed as of PHP 7.3.0:

Continuing an expression after a closing identifier

```

<?php
$values = [<<<END
a
    b
        c
    END, 'd e f'];
var_dump($values);

```

ENTITY

```

array(2) {
    [0] =>
        string(11) "a
        b
        c"
}

```

```
[1] =>
string(5) "d e f"
}
```

“ [!WARNING] If the closing identifier is found at the start of a line, then regardless of whether it is a part of another word, it is considered as the closing identifier and causes a ParseError.

Closing identifier in body of the string tends to cause ParseError

```
<?php
$values = [<<<END
a
b
END ING
END, 'd e f'];
```

ENTITY

```
Parse error: syntax error, unexpected identifier "ING", expecting "]" in
example.php on line 5
```

To avoid this problem, it is safe to follow the simple rule: *do not choose a word that appears in the body of the text as a closing identifier.*

“ [!WARNING] Prior to PHP 7.3.0, 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.

Invalid example, prior to PHP 7.3.0

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

Valid example, even prior to PHP 7.3.0

```
<?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 `string`s.

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;
?>

```

ENTITY

```

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:

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:

Using Heredoc to initialize static values

```
<?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:

Using double quotes in Heredoc

```
<?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 string interpolation 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 `ENTITY![CDATA[ ]]ENTITY` construct, in that it declares a block of text which is not for parsing.

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

Nowdoc string quoting example

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

ENTITY

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

Nowdoc string quoting example with variables

```
<?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;
?>
```

ENTITY

```
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
```

Static data example

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

```
}  
?>
```

String interpolation

When a `string` is specified in double quotes or with heredoc, [variables](#) can be substituted within it.

There are two types of syntax: a [basic](#) one and an [advanced](#) one. The basic 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.

Basic syntax

If a dollar sign (`$`) is encountered, the characters that follow it which can be used in a variable name will be interpreted as such and substituted.

String Interpolation

```
<?php  
$juice = "apple";  
  
echo "He drank some $juice juice." . PHP_EOL;  
  
?>
```

ENTITY

```
He drank some apple juice.
```

Formally, the structure for the basic variable substitution syntax is as follows:

```
string-variable::  
    variable-name (offset-or-property)?  
    | ${ expression }  
  
offset-or-property::
```

```
offset-in-string
| property-in-string
```

offset-in-string::

```
[ name ]
| [ variable-name ]
| [ integer-literal ]
```

property-in-string::

```
-> name
```

variable-name::

```
$ name
```

name::

```
[a-zA-Z_\x80-\xff][a-zA-Z0-9_\x80-\xff]*
```

“ [!WARNING] The `${ expression }` syntax is deprecated as of PHP 8.2.0, as it can be interpreted as [variable variables](#):

```
<?php
const foo = 'bar';
$foo = 'foo';
$bar = 'bar';
var_dump("${foo}");
var_dump("${(foo)}");
?>
```

ENTITY

Deprecated: Using \${var} in strings is deprecated, use {\$var} instead in file on line 6

Deprecated: Using \${expr} (variable variables) in strings is deprecated, use \${\${expr}} instead in file on line 9

```
string(3) "foo"
string(3) "bar"
```

ENTITY

```
string(3) "foo"
string(3) "bar"
```

The [advanced](#) string interpolation syntax should be used instead.

“ [!NOTE] If it is not possible to form a valid name the dollar sign remains verbatim in the string:

```
<?php
echo "No interpolation $  has happened\n";
echo "No interpolation $\n has happened\n";
echo "No interpolation $2 has happened\n";
?>
```

ENTITY

```
No interpolation $  has happened
No interpolation $
  has happened
No interpolation $2 has happened
```

Interpolating the value of the first dimension of an array or property

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

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

```
echo "He drank some $juices[string_key] juice.";
echo PHP_EOL;

class A {
    public $s = "string";
}

$o = new A();

echo "Object value: $o->s.";
?>
```

ENTITY

```
He drank some apple juice.
He drank some orange juice.
He drank some purple juice.
Object value: string.
```

“ [!NOTE] The array key must be unquoted, and it is therefore not possible to refer to a constant as a key with the basic syntax. Use the [advanced](#) syntax instead.

As of PHP 7.1.0, *negative* numeric indices are supported too.

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;
?>
```

ENTITY

The character at index -2 is n.

Changing the character at index -3 to o gives strong.

For anything more complex, the [advanced](#) syntax must be used.

Advanced (curly) syntax

The advanced syntax permits the interpolation of *variables* with arbitrary accessors.

Any scalar variable, array element or object property (static or not) with a `string` representation can be included via this syntax. The expression is written the same way as it would appear outside the `string`, and then wrapped 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:

Curly Syntax

```
<?php
const DATA_KEY = 'const-key';
$great = 'fantastic';
$arr = [
    '1',
    '2',
    '3',
    [41, 42, 43],
    'key' => 'Indexed value',
    'const-key' => 'Key with minus sign',
    'foo' => ['foo1', 'foo2', 'foo3']
];

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

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

// To show curly braces in the output:
echo "This is {{ $great }}" . PHP_EOL;

class Square {
    public $width;
```

```

    public function __construct(int $width) { $this->width = $width; }
}

$square = new Square(5);

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

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

// Works
echo "This works: {$arr[3][2]}" . PHP_EOL;

echo "This works: {$arr[DATA_KEY]}" . PHP_EOL;

// When using multidimensional arrays, always use braces around arrays
// when inside of strings
echo "This works: {$arr['foo'][2]}" . PHP_EOL;

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

echo "This works: {$obj->$staticProp}" . PHP_EOL;

// Won't work, outputs: C:\directory\{fantastic}.txt
echo "C:\directory\{$great}.txt" . PHP_EOL;

// Works, outputs: C:\directory\fantastic.txt
echo "C:\\directory\\{$great}.txt" . PHP_EOL;
?>

```

“ [!NOTE] As this syntax allows arbitrary expressions, it is possible to use [variable variables](#) within the advanced syntax.

String access and modification by character

Characters within `string`s 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, `string`s 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 an integer. An 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.

Some string examples

```

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

// Get the third character of a string
$third = $str[2];
var_dump($third);

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

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

```

String offsets must be either integers or integer-like strings. As of PHP 8.0.0, any other string throws a `TypeError`, except for a string starting with an integer followed by other characters, which emits `E_WARNING` and is interpreted as that leading integer. Prior to PHP 8.0.0, no `TypeError` was thrown: an illegal offset emitted `E_WARNING` instead.

Example of Illegal String Offsets

```

<?php
$str = 'abc';

$keys = [ '1', '1.0', 'x', '1x' ];

foreach ($keys as $keyToTry) {
    var_dump(isset($str[$keyToTry]));

    try {
        var_dump($str[$keyToTry]);
    } catch (TypeError $e) {
        // This block is not reached in PHP 8.0.0
    }
}

```

```
    } catch (TypeError $e) {
        echo $e->getMessage(), PHP_EOL;
    }

    echo PHP_EOL;
}
?>
```

ENTITY

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

bool(false)
Cannot access offset of type string on string

bool(false)
Cannot access offset of type string on string

bool(false)

Warning: Illegal string offset "1x" in example.php on line 10
string(1) "b"
```

ENTITY

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

bool(false)

Warning: Illegal string offset '1.0' in example.php on line 10
string(1) "b"

bool(false)

Warning: Illegal string offset 'x' in example.php on line 10
string(1) "a"
```

```
bool(false)
```

```
Notice: A non well formed numeric value encountered in example.php on line 10  
string(1) "b"
```

“ [!NOTE] Accessing variables of other types (not including arrays or objects implementing the appropriate interfaces) using `[]` or `{}` silently returns ENTITY.

“ [!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

`String`s 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 functions](#) for advanced find ENTITY 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 `echo` or `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` ENTITY value is converted to the `string` `"1"`. `bool` ENTITY 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 `float`s). 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 a period (`"."`). Prior to PHP 8.0.0, the decimal point character is defined in the script's locale (category `LC_NUMERIC`). See the `setlocale` function.

`Array`s 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 `object`s to `string`, the magic method `__toString` must be used.

`Resource`s are always converted to `string`s 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.

ENTITY 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 `string`s 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 the 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 the case of, for instance, `substr`, `strpos`, `strlen` or `strcmp`. Another way to think of these functions is that they operate on memory buffers, i.e., they work with bytes and byte offsets.
- Other functions receive the encoding of the string as a parameter or assume a default when it's not provided. This is the case of `htmlentities` and the majority of the functions in the [mbstring](#) extension.
- Others use the current locale (see `setlocale`), but operate byte-by-byte.
- 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).

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.