

3.3: PHP Errors

Before starting, an understanding of errors will help you quickly recognize where problems exist (and if they are truly problems) in your code, which will lend to faster debugging and understanding where to look for problems.

To start with, we can tell PHP what kind of errors we want to know about before we even run a script. While the full list of supported reporting levels (see [Table 8 PHP Errors](#)) covers a variety of concerns, there are a few (notices, errors, and warnings) that cover what we will run into most often.

Notices

1. Notice: Undefined index: message in /home/example.php on line 9

Notices, technically, are not errors. They will notify us of things that we may have intended or wanted, but that PHP can do without. For example, using a variable on a page without declaring it first will generate a notice. PHP will create the variable as soon as it is called, even if we did not declare it, without creating a formal error (other languages would consider this an error worthy of breaking execution). By notifying us but still continuing, if we had already declared or used the variable elsewhere, this notice would indicate a spelling error or mistyped variable name.

Warnings

1. Warning: main(): Failed opening 'noFileHere.php' for inclusion on line 2

Warnings still will not stop our script from running, but indicate that something has gone wrong during execution of a script. Attempting to use `include()` on a file that does not exist would create a warning.

Errors

1. PHP Fatal error: Undefined class constant 'MYSQL_ATTR_USE_BUFFERED_QUERY' in database.inc on line 43

Finally, errors are unrecoverable (execution will stop). Typical causes of errors are parsing errors like missing semi-colons, function, or class definitions, or other problems the engine does not know how to resolve. If we used `require()` on a file instead of `include`, an error would be triggered instead.

Most errors that we will receive are parsing errors. They are typically problems caused by *what* we wrote in our code, like missing brackets, semi-colons, or typos. When we receive an error, the compiler will tell us what problem it discovered and where. Keep in mind that we are being told where an error was *found* not necessary where the source of the problem exists. For example, a missing semi colon or bracket may have occurred several lines before it created a problems for the compiler.

The other category of errors we will run into are logical. These are errors caused by *how* we wrote our code, and can be much more frustrating. Logical errors are usually discovered when the script does not behave as expected. The source can be mistakes in what code we run in different parts of an if/then statement or even an error in math used in a function that gives us the wrong solution.

Resolving errors can be something of an art form. With parse errors, the engine can guide you to the area to help you begin looking for the source of the error. Logical errors can usually be resolved by adding extra, temporary outputs to follow the value of a variable or trace execution of logic statements through a script. This technique can help you find where what happens differs from what you expect. Unit testing your functions will go a long way toward preventing many of these issues, as does iterative programming.

To dictate what errors we do and do not wish to see in our script output, we will use the `error_reporting()` function. By passing one or more of the constants below, we control what is reported. For example, maybe we want information on warnings and errors, but do not care about notices. To do this, we can call `error_reporting(E_WARNING | E_ERROR)`. The pipe symbol (`|`) works as an or in this case. If we want to see everything except notices we can use `E_ALL` but leave out notices with the carrot (`^`) character to indicate an exception with `error_reporting(E_ALL ^ E_NOTICE)`. It is good practice to set your error reporting level close to the top of your script, so you can easily find it and change settings:

1. `<?php`
2. `error_reporting(E_WARNING | E_ERROR);`
3. `//This next line will trigger a notice that the variable does not exist, but we will not see it`
4. `echo $test;`
5. `?>`

```
6. <?php
7. error_reporting(E_ALL);
8. //This time we will see the notice
9. echo $test;
10. ?>
11. Notice: Undefined variable: test on line 3
```

You may be wondering why we would selectively show or hide errors; when we are developing our code, the system errors we will need to see in order to debug are different from what we would want end users to see for a system in production. Revealing, verbatim, the system generated error message is not only confusing to non-programmers but can expose sensitive information to those with malicious intent. Instead, we would provide a message we chose in the error's place. Take a look at the full list of error reporting levels:

Table 3.3.1 PHP Errors

Constant	Description
E_ERROR	Fatal run-time errors. These indicate errors that cannot be recovered from, such as a memory allocation problem. Execution of the script is halted.
E_WARNING	Run-time warnings (non-fatal errors). Execution of the script is not halted.
E_PARSE	Compile-time parse errors. Parse errors should only be generated by the parser.
E_NOTICE	Run-time notices. Indicate that the script encountered something that could indicate an error, but could also happen in the normal course of running a script.
E_CORE_ERROR	Fatal errors that occur during PHP's initial startup. This is like an E_ERROR , except it is generated by the core of PHP.
E_CORE_WARNING	Warnings (non-fatal errors) that occur during PHP's initial startup. This is like an E_WARNING , except it is generated by the core of PHP.
E_COMPILE_ERROR	Fatal compile-time errors. This is like an E_ERROR , except it is generated by the Zend Scripting Engine.
E_COMPILE_WARNING	Compile-time warnings (non-fatal errors). This is like an E_WARNING , except it is generated by the Zend Scripting Engine.
E_USER_ERROR	User-generated error message. This is like an E_ERROR , except it is generated in PHP code by using the PHP function trigger_error() .
E_USER_WARNING	User-generated warning message. This is like an E_WARNING , except it is generated in PHP code by using the PHP function trigger_error() .
E_USER_NOTICE	User-generated notice message. This is like an E_NOTICE , except it is generated in PHP code by using the PHP function trigger_error() .
E_STRICT	Enable to have PHP suggest changes to your code which will ensure the best interoperability and forward compatibility of your code.

Constant	Description
E_RECOVERABLE_ERROR	Catchable fatal error. It indicates that a probably dangerous error occurred, but did not leave the Engine in an unstable state. If the error is not caught by a user defined handle (see also set_error_handler()), the application aborts as it was an E_ERROR .
E_DEPRECATED	Run-time notices. Enable this to receive warnings about code that will not work in future versions.
E_USER_DEPRECATED	User-generated warning message. This is like an E_DEPRECATED , except it is generated in PHP code by using the PHP function <code>trigger_error()</code> .
E_ALL	All errors and warnings, as supported, except of level E_STRICT prior to PHP 5.4.0.

Adapted from [php.net](#), [Creative Commons 3.0 Attribution Unported](#)

3.3: PHP Errors is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

- **3.3: PHP Errors** by [Michael Mendez](#) has no license indicated.