

## **Programming in PHP**

**X52.9224. Spring 2005**

Instructor : David Mintz <dmintz@davidmintz.org>

[http://davidmintz.org/php\\_course/](http://davidmintz.org/php_course/)

**Note:** Portions of this document are stolen from the official PHP documentation --

<http://php.net/manual/en/>

# **Session Four**

## **Outline**

require() and friends

include\_path in php.ini and ini\_set()

sending email

filesystem functions and file I/O

date and time functions

error reporting and handling

## **require(), include() and friends**

Rather than making you copy and paste all the code that you want to re-use in every script, PHP provides a mechanism to load external files into your script.

The include() statement includes and evaluates the specified file.

```
<?php include 'some_file.php'; ?>
```

The documentation below also applies to require(). The two constructs are identical in every way except how they handle failure. include() produces a Warning while require() results in a Fatal Error. In other words, use require() if you want a missing file to halt processing of the page. include() does not behave this way, the script will continue regardless. Be sure to have an appropriate include\_path setting as well (discussed below).

Be warned that parse error in a require() d file does not cause the main script to abort.

When a file is included, the code it contains inherits the variable scope of the line on which the include occurs. Any variables available at that line in the calling file will be available within the called file, from that point forward.

Conversely, any variables set (or reset) in the included file will be visible in the calling file when execution returns to the caller.

Scoping rules apply just as if you had copied and pasted the contents of the included file inline. That means if you include() inside a function body, the variables in the included file are local to that function.

When a file is included, parsing drops out of PHP mode and into HTML mode at the beginning of the target file, and resumes again at the end. For this reason, any code inside the target file which should be executed as PHP code must be enclosed within valid PHP start and end tags. (Conversely, if you just want to drop in a chunk of html stored in another file, you can).

## The `_once` variants

We said before that once a function is defined, it cannot be undefined or redefined. That means it's a fatal error to `include()` a file containing function definitions and then `include` that same file again in the lifetime of your program. In large applications where multiple files may depend on the same included file, it could be problematic to keep track of what's been included and what hasn't. Fortunately PHP provides `include_once()` and `require_once()`, which do exactly that: ensure that once loaded, the same file won't be reloaded. (Of course, it can't prevent you from trying to load same-named functions from *different* files. )

For more a few more subtleties about `include()` and friends, please see <http://us4.php.net/manual/en/function.include.php>

## The `include_path`

The `include_path` configuration directive can be set in `php.ini`, and/or set at run-time via

```
<?php ini_set("include_path", $my_include_path); ?>
```

(You can also set `include_path` and other PHP configuration directives in Apache's `httpd.conf` or, if your Apache configuration allows it, it can be set per-directory in a `.htaccess` file.)

`include_path` contains the set of directories in which you want PHP to look for included files, and PHP will search them in the order they appear from left to right. The list of directories is delimited by `:` (colon) on Unix-type systems and by `;` (semicolon) on Windows. On a Unix host, a typical `include_path` setting might look like this in `php.ini`:

```
./usr/local/lib/php
```

or perhaps, like this after it's been overridden by a user who wants PHP to search his personal includes directory:

```
/usr/local/lib/php:/usr/home/dmintz/inc/pear
```

Note that you can also pass `include()` the full path of the file you want, or a path relative to the current directory. You can also use a path of the form `"subdirectory/filename.php"`, where `subdirectory` itself is found in the top level of one of the directories in your `include_path`. This technique is preferable to having an large number of directories in your `include_path`, which can adversely affect performance, and be a headache to manage.

```
<?php
```

```
// relative path
include "../my_file.php" ;
// absolute path
include "/full/path/to/my_file.php" ;
?>
```

In short, there is no shortage of ways to help your PHP program find the files that it needs to `include()`.

## **ini\_set()**

We just introduced the `ini_set()` function above in connection with setting your include path at runtime. You can also use `ini_set()` to change many of PHP's other configuration directives at runtime – i.e., from within your script – and override the settings found in `php.ini` or in an Apache config file. From the official docs:

```
string ini_set ( string var_name, string new_value )
```

Sets the value of the given configuration option. Returns the old value on success, **FALSE** on failure. The configuration option will keep this new value during the script's execution, and will be restored at the script's ending. Not all the available options can be changed using `ini_set()`.

There is a list of all available configuration options at <http://php.net/manual/en/ini.php#ini.list>, with some of the entries linked to further explanation. Some config options that can only be set in `php.ini` or `httpd.conf` only; some that can be set in `php.ini` or `httpd.conf` or `.htaccess`; and some that can be set anywhere, i.e., within user scripts plus any of the above.

## **sending mail**

Sending email is about as easy as it can be with PHP: use the built-in `mail()` function. You first have to set a couple of directives in `php.ini`.

**In Unix-style OSes** (including Mac OS X), you should first set the `sendmail_path` directive appropriately. `sendmail_path` tells PHP where the sendmail program can be found, usually `/usr/sbin/sendmail` or `/usr/lib/sendmail`. Your installation may well already have it set correctly, so test it before you change it. Systems not using sendmail should set this directive to the sendmail wrapper/replacement their mail system offers, if any. For example, qmail users can normally set it to `/var/qmail/bin/sendmail` or `/var/qmail/bin/qmail-inject`.

A user comment on the `php.net` documentation says:

"Mac OS X users will need to 'enable' sendmail (v10.2 and earlier) or postfix(v10.3). Otherwise mail sent with `mail()` will just queue up on your computer and never be delivered anywhere. You will receive no error regarding this from php as php is doing its job just fine. Visit

<http://www.macupdate.com/> and search for postfix or sendmail and you'll find a convenient tool for 'enabling' sendmail."

**On Windows systems** the you should set the `smtp` directive in `php.ini` to your ISP's smtp server, and `sendmail_from` directive should be the email address that you want in the "From: " header in messages you send via PHP.

bool **mail** ( string *to*, string *subject*, string *message* [, string *additional\_headers* [, string *additional\_parameters*]])

**mail()** automatically mails the message specified in *message* to the receiver specified in *to*. Multiple recipients can be specified by putting a comma between each address in *to*. The optional fourth parameter is for adding extra headers such as "Cc " and "Reply-to." End each additional header with "\r\n" The optional fifth parameter is for passing additional arguments to sendmail on \*nix systems (and you will probably be able to do without it).

```
<?php
```

```
$to = 'somebody@somehost.com';
$subject = 'wassup?' ;
$body = <<<__END__
```

```
Hello,
```

```
How's it going? I'm just fine, learning all this cool PHP stuff.
```

```
OK, gotta go. Later,
```

```
me
```

```
__END__;
```

```
mail($to, $subject, $body,
    "From: webmaster@{$_SERVER['SERVER_NAME']}\r\n" .
    "Reply-To: webmaster@{$_SERVER['SERVER_NAME']}\r\n" .
    "X-Mailer: PHP/" . phpversion());
?>
```

Returns true if the message was accepted for delivery, false otherwise.

## Filesystem functions

Reading data from and writing data to files is easy in PHP. You do need to make sure that the user as which your process is running has sufficient permissions to do whatever it is trying to do.

To read or write to a file, the first thing you do is call `fopen()` to get a hold of a *resource*, meaning the PHP data type that represents some sort of data that is external to PHP – such as a

database connection or database query result, or, in this instance, what is commonly known as a *file handle*. You use **fopen()** to bind a named resource, specified by *filename*, to a *stream*:

```
resource fopen ( string filename, string mode [, bool use_include_path [, resource zcontext]])
```

The *filename* is simply the name of the file (e.g. '/data/foo.txt' ). Note that PHP understands / as the directory delimiter on both Windows and Unix platforms, so if you are developing on Windows and deploying on Unix, an easy way to keep it portable is simply to use forward slashes consistently.

The *mode* is any of those listed at <http://us4.php.net/manual/en/function.fopen.php>. However, we will be talking about 'r' 'w' and 'a' -- for read, write and append respectively. These flags tell PHP what you intend to do with the file – read from it, write to it, or append to it.

The optional third *use\_include\_path* parameter can be set to '1' or TRUE if you want to search for the file in the *include\_path* for the file *filename*. (The last parameter *zcontext* has to do with a rather exotic and under-documented feature, and we won't be using it – but if you are curious, try <http://php.net/manual/en/ref.stream.php>)

`fopen()` returns the file handle if it succeeds, or false on failure. It fails if, for example, you try to read a file that does not exist, or try to write to a file where the server process doesn't have write permission.

After you have the resource in hand, you can use a `while` loop to read (or write) successive chunks of data with `fgets()` and `fputs()`, respectively.

```
string fgets ( resource handle [, int length])
```

Returns a string of up to *length* - 1 bytes read from the file pointed to by *handle*. Reading ends when *length* - 1 bytes have been read, on a newline (which is included in the return value), or on EOF (End Of File), whichever comes first. If no *length* is specified, the *length* defaults to 1k, or 1024 bytes. A linebreak character and EOF are both treated as stopping points.

The function you will use in the conditional expression of such a `while` loop is `feof()`, which tests for end-of-file on a file pointer.

```
bool feof ( resource handle)
```

Returns **TRUE** if the file pointer is at EOF or an error occurs; otherwise returns **FALSE**.

You should always *check whether you have a valid handle* before you try to do anything with it. The error suppression operator `@` can be useful for precisely this sort of situation – it suppresses any error message that the expression following it might otherwise emit.

When you are finished working with *handle*, you should close it with `fclose($handle)`.

## **fopen() example: reading a file**

```
<?php
$handle = @fopen("/tmp/inputfile.txt", "r") or die("Oops, could not open
file");
while (!feof($handle)) {
    $buffer = fgets($handle, 4096);
    echo $buffer;
}
fclose($handle);
?>
```

## **fopen() example: writing data to a file**

```
<?php
/* snippet for a primitive guestbook application */

// after you checked whether comments was empty...
$comments = strip_tags($_POST['comments']);

// open file for appending
$handle = @fopen('./comments.txt','a' ) or die('oops');

// write the comments to the file
fputs($handle,"\n" . $comments);

// close
fclose($handle);
?>
```

If you successfully open a file in write mode and that file already exists, it will be overwritten; if it doesn't already exist, it will be created. If you open a file in append mode that already exists, you will append to the file (the existing contents will be preserved); otherwise, it will be created for you.

## **Opening remote files**

A setting in php.ini called `allow_url_fopen` determines whether `fopen` can open files on remote systems with syntax such as the following:

```
<?php
    $handle = fopen('http://example.org/foo.txt' );
?>
```

There have been PHP applications with severe security vulnerabilities that allowed badguys to execute arbitrary code on systems where `allow_url_fopen` was enabled. It can now (in modern PHP versions) only be enabled in `php.ini` – it is not settable at runtime via `ini_set()`. If you are on a machine where you (1) can't modify `php.ini`, and (2) `allow_url_fopen` feature is disabled, and (3) your PHP program needs to read something like a remote web page, there are at least three other solutions: use the `cURL` extension; use a PEAR package called `HTTP`; or open a network socket connection and speak `HTTP`. Later in the course we will see examples of each.

## More file-reading convenience functions

array **file** ( string *filename* [, int *use\_include\_path* [, resource *context*]])

Returns the file in an array. Each element of the array corresponds to a line in the file, with the newline still attached. Upon failure, **file()** returns **FALSE**.

string **file\_get\_contents** ( string *filename* [, bool *use\_include\_path* [, resource *context*]])

Reads entire file into a string. Identical to **file()** except that **file\_get\_contents()** returns the file in a string. On failure, **file\_get\_contents()** will return **FALSE**.

**file\_get\_contents()** is the preferred way to read the contents of a file into a string. It will use memory mapping techniques if supported by your OS to enhance performance.

int **file\_put\_contents** ( string *filename*, string *data* [, int *flags* [, resource *context*]])

Identical to calling **fopen()**, **fwrite()**, and **fclose()** successively. Returns the number of bytes that were written to the file. *Introduced in PHP 5.0.*

int **readfile** ( string *filename* [, bool *use\_include\_path* [, resource *context*]] )

Reads a file and writes it to the output buffer. Returns the number of bytes read from the file. If an error occurs, **FALSE** is returned and unless the function was called as **@readfile()**, an error message is printed.

Why use the long-winded approach (**fopen()**, **fgets()**, **fclose()**) at all if functions like **file\_get\_contents()** are available? One answer is that **file\_get\_contents()** was introduced in PHP version 4.3.0; **file\_put\_contents()** was introduced in 5.0; the older functions remain in place alongside the newer ones for backward compatibility. Another answer: is that if the file you are working with is quite large, then you might consume too much memory if you read the whole thing into memory at once, so it is more efficient to process it chunk by chunk.

## All the rest

There are several dozen more filesystem functions available for tasks like reading directory listings, copying, deleting, and renaming files, checking available disk space, getting file statistics like file size, file last modification time, parsing a configuration file, and more. See <http://php.net/manual/en/ref.filesystem.php> and <http://www.php.net/manual/en/ref.dir.php>.

## Date and Time Functions

Writing date-and-time-aware PHP pages is a good way to start putting PHP to work for you and easing your manual maintenance burden.

For efficiency's sake, computer programs often store dates and times internally as *Unix timestamps*, the number of seconds elapsed since 12:00 am January 1, 1970 (also known as *epoch seconds*). This makes it easy to compare date/times for equality (e.g., is it before or after November 2, 2004?) This scheme typically supports dates from Fri, 13 Dec 1901 20:45:54 GMT to Tue, 19 Jan 2038 03:14:07 GMT (the dates that correspond to the minimum and maximum values for a 32-bit signed integer). On Windows this range is limited from 01-01-1970 to 19-01-2038.

A natural question is, how do I display the current date and time with PHP? The `time()` function returns the system time as a Unix timestamp. Per the official documentation:

int **time** ( void )

Returns the current time measured in the number of seconds since the Unix Epoch (January 1 1970 00:00:00 GMT).

```
<?php echo time() ?>
```

At this writing, the above code outputs:

```
1097621560
```

Cute, but how do you make it human-readable? One solution is to use the `date()` function.

string **date** ( string *format* [, int *timestamp*])

Returns a *string* formatted according to the given *format* string using the given integer *timestamp*. If no *timestamp* is given, guess what: it defaults to the value of `time()`, meaning now according to your server's clock.

Date format strings, unless you use them quite frequently, are one of those things that will keep you going back to a reference like <http://php.net/manual/en/function.date.php> to consult the table of symbols and meanings. As with `printf()`, the function will expand the characters it recognizes and pass the rest through untouched. If you want to print literally a character that also happens to be a format character, you have to escape it with a `\`. That can get rather annoying:

```
<?php echo date('C\u\r\r\e\n\t \t\i\me: l d-M-Y g:i a' ); ?>
```

so you will find it easier to say

```
<?php echo "Current time: " . date('l d-M-Y g:i a' ); ?>
```

In either case the output (at this writing) is:

```
Current time: Tuesday 12-Oct-2004 8:32 pm
```

The converse (in a sense) of `date()` is `mktime()`. You pass it some date information, and it returns the corresponding Unix timestamp.

```
int mktime ( [int hour [, int minute [, int second [, int month [, int day [, int year [, int is_dst]]]]]]])
```

The bracket notation above (as in all function signatures) means the argument is optional. Any argument you leave out defaults to whatever `time()` would say. Generally speaking, it's probably wise to give `mktime()` at least six arguments.

One of the powerful features of `mktime()` is that you can use it for date arithmetic and validation, as it will automatically calculate the correct value for out-of-range input. For example, each of the following lines produces the string "Jan-01-1998".

```
<?php
echo date("M-d-Y", mktime(0, 0, 0, 12, 32, 1997));
echo date("M-d-Y", mktime(0, 0, 0, 13, 1, 1997));
echo date("M-d-Y", mktime(0, 0, 0, 1, 1, 1998));
echo date("M-d-Y", mktime(0, 0, 0, 1, 1, 98));
?>
```

Thus, if you want to display some content on a web page until a certain time on a date that is a known number of days from today, you can be lazy about it. All you need to know is the current date and time. Suppose today is October 12, 2004, and you have some content you want to display only until 5:30 pm 30 days from today, whenever that is:

```
<?php
if (time() <= mktime(17,30,0,12+30,10,2004)) { //note the 4th arg (-:
    // content that you want to expire
}
```

You will still want to clean up this page eventually, but it's convenient that you can literally sleep right through the critical date/time and not need to worry about stale content hanging out.

Here is an example of how you might compare the current time to a specific date/time, and make your PHP page react accordingly:

```
?php
// Can you think of a simpler solution than this?
printf("Gabriela was born %s<br />",
    date("d-M-Y g:i a",mktime(1,21,0,5,16,2003)));

// store Gabriela's second birthday timestamp as $then
$then = mktime(1,21,0,5,16,2005) ;

// store current unix timestamp as $now
$now = time();

// report current date and time
printf("The current date and time is %s<br />\n",date('d-M-Y g:i a',$now));

// if Gabriela's second birthday is in the future...
```

```

if ($now < $then) {
    //... compute the days, hours, and minutes to go
    $diff = $then - $now;
    $days = floor($diff/(60*60*24));
    echo "$days days, " ;
    $diff -= $days * 60*60*24;
    $hours = floor($diff/3600);
    echo "$hours hours, " ;
    $diff -= $hours * 3600;
    $minutes = floor($diff/60);
    echo "$minutes minutes " ;
    echo "to go before Gabriela turns two." ;
} elseif (date('Ymd',$now) == date('Ymd',$then)) {
    echo "Today is Gabriela's second birthday.\n" ;
} else {
    echo "Gabriela Cloé is at least two years old.\n" ;
}

```

Output (at this writing):

```

Gabriela was born 16-May-2003 1:21 am
The current date and time is 03-Mar-2005 5:39 pm
73 days, 6 hours, 41 minutes to go before Gabriela turns two.

```

Another interesting date/time function that PHP provides is `strtotime()`, which can parse about any English textual datetime description into a Unix timestamp  
**int strtotime ( string *time* [, int *now*])**

Given a string containing an English date format, tries to parse that format into a Unix timestamp relative to the timestamp given in *now*, or the current time if none is supplied. Upon failure, *-1* is returned.

The first line of the preceding code example could have used `strtotime()` like so:

```
$then = strtotime("May 16, 2005");
```

and the output would be identical.

`strtotime()` supports relative dates in more or less plain English:

```

<?php
echo "I am now preparing for a PHP class which will take place on "
. date('l d-M-Y g:i a' ,strtotime("this Thursday 6:20 pm" ));
?>

```

Compared to `mktime()`, `strtotime()` is more convenient and readable, but PHP has to work harder to evaluate it, so it's slower (but it is exceedingly cool). See <http://php.net/manual/en/function strtotime.php> to appreciate the full power of `strtotime()`.

Other functions that you might find useful:

array **getdate** ( [int *timestamp*])

Returns an associative array containing the date information of the timestamp, or the current local time if no timestamp is given, as in

```
<?php print_r(getdate()); ?>
```

which on October 12, 2004 at 10:26 pm, generated this output:

```
Array
(
    [seconds] => 10
    [minutes] => 26
    [hours] => 22
    [mday] => 12
    [wday] => 2
    [mon] => 10
    [year] => 2004
    [yday] => 285
    [weekday] => Tuesday
    [month] => October
    [0] => 1097634370
)
```

Among the other functions worth having in your toolchest are `gmdate()` and `gmmktime()`.

`gmmktime()` is identical to [mktime\(\)](#) except the passed parameters represents a GMT date.

`gmtime()` is identical to `time()` except that the time returned is GMT.

Greenwich Mean Time is impervious to daylight savings, and gives you an absolute reference in cases where time zones are an issue.

`checkdate()` is also good to know about:

```
bool checkdate ( int month, int day, int year)
```

Returns true if the date given is valid, otherwise false.

## Error handling and reporting

On your development machine you will usually want as much diagnostic and debugging information as possible. On a production machine it is considered unwise from a security standpoint to display too much information when an error condition occurs, and bad style from a user experience point of view. PHP allows for fine tuning of error handling and reporting by setting directives in `php.ini`, by setting them at runtime with `ini_set()` and other functions, and by creating custom error handlers and log messages.

When we speak of errors we should distinguish between things like user input validation errors, which we can control, and environmental problems, which we cannot. You can recover from invalid user input and keep going, but if some essential resource such as a critical file or a database is unavailable, you may prefer in a production environment to log the error (with verbose information), redirect the user to a generic error page, or display a generic error message, and abort the program (we will talk about redirection in due course). The user should see appropriately apologetic and polite but technically vague information, while the developer or site maintainer needs to see as much information as possible.

## Development environment

In `php.ini`, set `error_reporting` to `E_ALL` and `display_errors` to "on." You'll also do well to experiment with the `log_errors` and `error_log` directives and learn how they work, but strictly speaking you can develop without them. (If you have trouble locating `php.ini`, run `phpinfo()`)

## Production environment

Settings here should be almost the converse of what they are on your development machine. `display_errors = off` and `log_errors = on` are recommended. You can set the `error_log` to point to your own personal log file -- though the web server process has to be able to write to it.

You might want to set `error_reporting` down to `E_ALL & ~E_NOTICE`. If you're wondering what that funny stuff means: `error_reporting` is a *bit field* and the `&` and `~` are *bitwise and* and *bitwise not* operators, respectively. `E_ALL` and `E_NOTICE` are predefined constants. See <http://php.net/manual/en/language.operators.bitwise> if you are curious about the bitwise operators. Run this snippet if you'd like to see how the decimal and binary representations of these constants compare.

```
<pre>
<?
$err_names = preg_grep("/^E_/",array_keys(get_defined_constants()));
foreach($err_names as $err) {
    printf("%20s%8s  %12b\n",
           $err, constant($err) , constant($err)
    );
}
echo "\n" ;
printf("E_ALL except E_NOTICE:          %01b\n",
       E_ALL & ~E_NOTICE);
</pre>
```

See [http://en.wikipedia.org/wiki/Binary\\_arithmetic](http://en.wikipedia.org/wiki/Binary_arithmetic) if you're interested.

## Custom error handlers

The `set_error_handler()` function allows you *register* your own error-handling function. You pass it the (string) name of the error handling function that you defined.

`set_error_handler()` tells PHP to call your function to handle errors during runtime, for example in applications in which you need to do cleanup of data/files when a critical error

happens, or when you need to trigger an error yourself (using `trigger_error()`). This is sometimes called a *callback* function because you normally don't call it yourself directly; instead, it gets called when a particular event occurs.

mixed **set\_error\_handler** ( callback *error\_handler* [, int *error\_types*])

Returns a string containing the previously defined error handler (if any), or **FALSE** on error. (The second parameter *error\_types* was introduced in PHP 5 and can be used to mask the triggering of the *error\_handler* function just like the `error_reporting` ini setting controls which errors are shown. Without this mask set the *error\_handler* will be called for every error regardless to the setting of the `error_reporting` setting.)

Your custom error handler has to accept two obligatory and may accept three optional arguments.

**handler** ( int *errno*, string *errstr* [, string *errfile* [, int *errline* [, array *errcontext*]]])

The first two are the level of the error raised as an integer; and a string describing the error. The rest are optional (but useful): the full path to the file in which the error happened, as a string; the line number where the error happened (integer); and an array reflecting the state your program was in when it happened, suitable for passing to `var_dump` or `print_r`. Here is a crude custom error handler:

```
<?php
ini_set('display_errors','off');
ini_set('log_errors','on');
ini_set('error_log','./my_log.txt');

function custom_err_handler($errno,$errstr,$errfile,$errline,$context) {

    echo "errno: is $errno\n";
    //echo "context: "; print_r($context["GLOBALS"]);// print_r
($context);
    switch ($errno) {

    case E_ERROR:
        // this, evidently, is unreachable because
        // by definition we die before we get here
        // you can't really intercept a fatal, but you can
        // get PHP to log a message about it
        echo "custom E_ERROR: $errstr\n at $errfile, $errline";
        die("time to die.");
        break;

    case E_WARNING:
        echo "custom E_WARNING: $errstr\n at $errfile, $errline";
        break;

    case E_NOTICE:
        echo "custom E_NOTICE: $errstr\n at $errfile, $errline";
        break;

    case E_USER_NOTICE:
        echo "E_USER_NOTICE: $errstr\n at $errfile, $errline";
        break;
```

```

    case E_USER_WARNING:
        echo "E_USER_WARNING:  $errstr\n at $errfile, $errline";
        break;
    case E_USER_ERROR:
        echo "E_USER_ERROR:  $errstr\n at $errfile, $errline. Aborting.";
        die();

    default:
        echo "unknown error $errno\n\ at $errfile, $errline.
    }
    echo "\n-----\n";
}

```

Again, this error handler will end up getting called when triggered by an error event. You can trigger such an event by calling `trigger_error`.

```

<?php
// assume we defined the above error handler
ini_set('display_errors','off');
ini_set('log_errors','on');
ini_set('error_log','./my_log.txt');
set_error_handler("custom_err_handler");

trigger_error("I just don't like it the way things are
going.",E_USER_WARNING);

?>

```

output (to a browser) is:

```

errno: is 512
E_USER_WARNING:  I just don't like the way things are going
  at C:\data\docs\php_course\examples\week4\custom_handler.php, 51
-----

```

One nuance worth noting is that if set, a custom error handling function overrides the `error_reporting` setting. And, as the PHP documentation so eloquently puts it, "it is your responsibility to `die()` if necessary".

## error\_logging

Two configuration settings to be aware of are `log_errors` and `error_log`.

`log_errors` is a boolean that tells PHP whether error logging is on or off. `error_log` is the (string) name of the file where script errors should be logged. Obviously if `log_errors` is off, the `error_log` setting is irrelevant. The log file has to be writeable by the web server.

With `error_logging` on, `display_errors` off, and `error_reporting` at a reasonably verbose level, anything weird that happens will be recorded automatically in the log file and no undesirable error output will go to the browser. If you want to manually record something in the error log, use the `error_log()` function.

int **error\_log** ( string *message* [, int *message\_type* [, string *destination* [, string *extra\_headers*]])

In its simplest usage, you call `error_log()` with a string *message* argument and it gets written to the log file (as set in the `error_log` directive) along with a timestamp in this format: [14-Oct-2004 13:01:56].

If there is a second argument of 1, it means you want to send an email, in which case the required third argument *destination* is the email address. Then, the optional fourth argument is a string containing extra headers for the email message.

A value of 2 in the second argument means you want to send debug information via a socket connection but this is only available in PHP 3 and then only if PHP is compiled with this feature enabled, so don't worry about it.

A value of 3 as the second argument means append the *message* to the destination *file*. This allows you to log messages to some file other than your primary error log.

For the following snippet to work, `my_log.txt` will need to be server-writable and located in the same directory as the file containing the following:

```
<pre>
<?php
ini_set('display_errors','off');
ini_set('log_errors','on');           // logging, yes or no
ini_set('error_log','./my_log.txt');  // where to log
error_reporting(E_ALL); // also try E_ALL & ~E_NOTICE

printf("Current setting of display_errors is: %s\n",      ini_get
('display_errors'));

printf("Current setting of error_reporting is: %s\n", ini_get
('error_reporting'));

trigger_error("I just don't like the way things are going",E_USER_WARNING);
trigger_error("I'm utterly displeased",E_USER_ERROR);

echo "\n";

$num = 7 / 0; // divide by zero, that's a warning

$fp = fopen('nonexistent_file','r');
echo "\n";
fclose($fp);
echo "\nstill here!";

error_log( sprintf(
    "I logged this pointless message from %s line %s",
    __FILE__,
    __LINE__
));
?>
if you don't see this, I died before getting this far.
</pre>
```

## PHP 5: Exceptions

PHP 5 provides a more sophisticated error handling mechanism known as Exceptions, which is comparable to exceptions found in other languages. A full discussion is more than we can jam into this session, and at this point it's probably wise to write PHP that runs under PHP 4, which has no notion of exceptions. You also need to understand some Object Oriented Programming for a full appreciation of the PHP 5 exception model, and we have yet to cover OOP. So we will slide over the topic with barely a nod of acknowledgement. If you're curious, see <http://www.php.net/manual/en/language.exceptions.php>

# Assignment

## Exercises

Not required to be handed in, but just for your own health and fitness:

- make a PHP page that displays the current date and time in whatever format you like.
- make that PHP page display your age in years.
- next, make it tell you whether the next PHP class has yet to happen, is currently in progress, or is already over. You can use these values to compare to time():

```
$next_class_start_time = strtotime("21 October 2004 6:20pm");  
$next_class_end_time = $next_class_start_time + (3*3600);
```

- now make it report the date and time that the page itself was last modified. (see <http://php.net/manual/en/function.getlastmod.php> for inspiration. )
- finish the assignment, then add a feature that emails you a notification whenever a new comment is posted.
- Try exercising require and/or include by making a well formed HTML document using the following pattern:

```
<?php  
  
$TITLE = "My PHP/HTML Document";  
  
// header.php contains the entire HEAD element  
// including <title><?php echo $TITLE ?></title>
```

```
// et cetera, all the way through the opening BODY tag

include("header.php");

    // here, insert dynamic content -- stuff that changes depending on user input or some other
    // variable -- such as the date-aware code you wrote above

include("footer.html") // static html you want at the bottom of your docs, e.g., closing tags.
?>
```

## Assignment

For this assignment we will make a simple guestbook called `guestbook.php` using the code from the Week 3 assignment as a point of departure. This *not* going to be a production quality application; it's just an exercise.

We are getting rid of the phone number field and replacing it with a field named 'name' instead.

Use the code provided at [http://davidmintz.org/php\\_course/4/guestbook.assignment.txt](http://davidmintz.org/php_course/4/guestbook.assignment.txt), filling in only the function bodies for the `display_comments()` `save_comment()` functions. All told, you will only need to add about 10 or 12 lines of code.

In your `save_comment()`: open the comments text file in append mode; write the comment to it, followed by a newline (`\n`); append a string that says the type ('subject') of comment date and time that it was posted, and by whom (according to the `$_POST['name']` variable); finally, append two newlines (`\n\n`). Then `fclose()` the file handle.

The `display_comments()` will open the comments file in read mode, and print it. Use [nl2br\(\)](#) and [htmlentities\(\)](#).

In both functions, check the return value of your `fopen()` call. If the `fopen()` call fails, you don't have to `die()`, just `return false` and let the caller worry about it. Use the `@` error suppression operator. Try making the program fail by renaming the `comments.txt` file and observe what happens.

*Remember that the last line of your function has to return true.* Otherwise the calling code will think something went wrong, and abort the script.

Once everything is working, cut all of the global variable declarations and function definitions from the beginning of `guestbook.php`, and move them into a separate file called `guestbook.inc`, and save it in the same directory as `guestbook.php` (in real life we might well move it somewhere else, but for simplicity's sake we are keeping the files together in the same folder). Then, put a `require('./guestbook.inc')` statement at the top of your `guestbook.php`

Please hand in two files for this assignment: `guestbook.php` and `guestbook.inc`

The rest of the code is provided, and after you complete it, it behaves like so:

When the guestbook is initially accessed (i.e., if it is not a POST request/form submission), it displays the comments followed by the form.

If there is a POST submission and a validation error, the form is re-displayed.

Otherwise, if the data validates, the comment is appended to a text file called comments.txt. Then the comments are displayed, followed by the comment form once again. For a demo of how your solution should work, see [http://davidmintz.org/php\\_course/4/guestbook.php](http://davidmintz.org/php_course/4/guestbook.php)

## Reading

- include() documentation: <http://php.net/manual/en/function.include.php>
- date/time function documentation: <http://php.net/manual/en/ref.datetime.php>
- mail() function documentation: <http://php.net/manual/en/function.mail.php>
- Error handling and logging documentation: <http://php.net/manual/en/ref.errorfunc.php>
- Sklar, *Learning PHP 5* Chapter 9; Chapter 10; Chapter 12; Appendix A, Modifying PHP Configuration Directives
- "PHundamentals: Error Handling"  
[http://education.nyphp.org/phundamentals/PH\\_error\\_handle.php](http://education.nyphp.org/phundamentals/PH_error_handle.php)
- David Skar, "PHP Debugging Basics"  
<http://www.onlamp.com/pub/a/php/2004/08/12/DebuggingPHP.html>