CHAPTER 3

Understanding and Using the CLI SAPI

As mentioned in the previous chapter, PHP CLI scripting involves using the PHP CLI SAPI. It's therefore important to have a good grasp of how to use it, know the options for configuring and running it, and understand how it differs from the web-based SAPIs you are used to using. Luckily, the differences are minimal, and many are intuitive.

What's Different About the CLI SAPI?

The following are the main differences between the CLI SAPI and the standard web implementation:

- No HTTP headers are written to the output by default. This makes sense because they hold no meaning in the command line and so would be just extraneous text printed before your genuine output. If your output will later be funneled out to a web browser, you will need to manually add any necessary headers (for instance, by using the `header()` PHP function).
- PHP does not change the working directory to that of the PHP script being executed. To do this manually, use `getcwd()` and `chdir()` to get and set the current directory. Otherwise, the current working directory will be that from which you invoked the script. For instance, if you are currently in `/home/rob` and you type `php /home/peter/some_script.php`, the working directory used in PHP will be `/home/rob`, not `/home/peter`.
- Any error or warning messages are output in plain text, rather than HTML-formatted text. If you want HTMLified errors, for instance, if you are producing static HTML files, you can override this by setting the `html_errors` runtime configuration directive to true in your script using `ini_set('html_errors', 1);`.

R. Aley, *PHP CLI*, DOI 10.1007/978-1-4842-2238-6_3

- PHP implicitly "flushes" all output immediately and doesn't buffer by default. Online performance can often be harmed by sending output straight to a browser, so instead output is buffered and sent in optimal-sized chunks when the chunk is full. Offline this is not likely to be an issue, so HTML blocks and output from constructs such as `print` and `echo` are sent to the shell straightaway. There is no need to use `flush()` to clear a buffer when you are waiting for further output. You can still use PHP's output buffering functions to capture and control output if you want; see the "Output Control Functions" section in the PHP manual for more information.
- There is no execution time limit set. Your script will run continuously until it exits of its own volition; PHP will not terminate it even if it hangs. If you want to set a time limit to rein in misbehaving scripts, you can do so from within the script using the `set_time_limit()` function.
- The variables `$argc` and `$argv`, which describe any command-line arguments passed to your script, are automatically set. These are discussed fully later in this chapter.
- PHP defines the constants `STDIN`, `STDOUT`, and `STDERR`, relating to the standard streams of the same name, and automatically opens input/output (I/O) streams for them. These give your application instant access to "standard input" (`STDIN`), "standard output" (`STDOUT`), and "standard error" (`STDERR`) streams.

Further Reading

- "Output Control Functions" section in the PHP manual
  - *http://www.php.net/manual/en/ref.outcontrol.php*
- "Standard streams" (`STDIN`, `STDOUT`, `STDERR`) on Wikipedia
  - *http://en.wikipedia.org/wiki/Standard_streams*

CLI SAPI Installation

To use the PHP CLI SAPI, you may need to install it first. Appendix A gives details on installing (and compiling, where necessary) PHP. However, you may find that it is already installed if you have PHP installed (often in a folder called `sapi/cli` in the PHP program folders), and if not, it is usually available in modern OS software repositories. (For example, in Ubuntu a package called `php5-cli` exists and can be installed from any package manager or via the command line with `sudo apt-get install php5-cli`.) If it is installed in the command-line search path, typing `php -v` on the command line will print the version details, confirming it is indeed installed.

PHP Command-Line Options

The PHP binary will accept a number of command-line options/switches/arguments that affect its operation. You can see a full list by typing `php -h`. Although some apply only to the CGI SAPI (used when there is not a "module" such as the PHP Apache module), the following are some of the more interesting and common ones used when interacting with the CLI SAPI:

- `-f` *or* `--file`: This allows you to specify the file name of the script to be run and is optional. The `-f` option exists to allow compatibility with software and scripts such as automation software, which can programmatically call command-line programs but require file-name arguments to be formed in this way. It also allows default file-type handlers to be easily set on Windows for PHP scripts. The only real difference in usage between the two versions of the earlier command come when interpreting command-line arguments passed to the script, which we look at in the "Command-Line Arguments for Your Script" section. In most cases, the two following lines are mostly equivalent:

  ```
  ~$ php -f myscript.php
  ~$ php myscript.php
  ```

- `-a` *or* `--interactive`: This runs PHP interactively, which allows you to type in PHP code, line by line, rather than executing a saved PHP script. This mode of operation is often called a "REPL" (Read-Eval-Print-Loop). As well as providing an interactive interface for testing and developing code, it can act as an enhanced PHP-enabled shell or command line, and I'll cover this more closely later in this chapter.

- `-c` *or* `--php-ini`: This specifies the PHP `.ini` file that PHP will use for this application. This is particularly useful if you are also running web services using PHP on the same machine; if it is not specified, PHP will look in various default locations for `php.ini` and may end up using the same one as your web service. By providing one specifically for your CLI applications, you can "open up" various restrictions that make more sense for offline applications. Note that by using the CLI SAPI, PHP will automatically override several `php.ini` settings regardless of whether you specify a custom `.ini` file using this option. These overridden settings are those that affect the behavior outlined in the "What's Different About the CLI SAPI?" section, and while the `php.ini` file is ignored in these cases, you can revert or change these settings directly in your code using the `ini_set()` function or similar. You can also use the `-d` or `--define` option to set options (for example, `php -d max_execution_time=2000 myscript.php`). If you are deploying software onto machines that you do not control (for example, if you are selling software for users to install on their own machines), it makes sense to use one of these mechanisms to ensure that PHP will be running with the settings you expect, not the settings the user may happen to have. See `-n` next as well.

- `-n` *or* `--no-php-ini`: This tells PHP not to load a `php.ini` file at all. This can be useful if you do not want to ship one with your application and instead set all of the settings directly within your code using `ini_set()` or similar. PHP will use its default settings if no `.ini` file is provided, and it is worth remembering that these default settings may change from version to version of PHP (and indeed have done so in the past). You shouldn't rely on the current defaults being suitable for your application. You can use `php --ini` to show the default path that PHP will look for `.ini` files when the `-n` option isn't used and `-c` isn't used to specify a file.
- `-e` *or* `--profile-info`: This puts PHP into Extended Information Mode (EIM). EIM generates extra information for use by profilers and debuggers. If you're not using a profiler or debugger that requires this mode, you should not enable it because it can degrade performance. You can find more information on profilers and debuggers in Chapter 4.
- `-i` *or* `--info`: This calls the `phpinfo()` function and prints the output. This outputs a *large* range of information about the PHP installation, in plain-text format rather than the usual HTML (it detects you are calling it from the CLI SAPI). This can be useful in tracking down issues with the installation, as well as giving you version information, lists of extensions installed, relevant file paths, and so on. As with any other shell command, the output can be piped to other commands, such as grep. So if you wanted to check whether IPv6 was enabled in your PHP binary for instance, on Linux or OS X you could try the following:

```
~$ php -i | grep -i "ipv6"
```

On Windows you could try the following:

```
> php -i | finstr /I ipv6
```

- `-l` *or* `--syntax-check`: This parses the file, checking for syntax errors. This is a basic "lint" type checker; more advanced static code analysis tools are discussed in the next chapter. Be aware that this option checks only for basic syntax errors—the sort that cause the PHP engine to fail. More subtle bugs, problems in your program logic, and errors that are created at run time will not be detected. Your code is not executed, so it can help pick up basic errors before running code that may alter data and cause problems if it fails. Even when you run such code in a testing environment, resetting data and setting up for another test can take time, so a quick check for basic syntax errors first can be a time-saver. Some integrated development environments (IDEs) and text editors run `php -l` in the background to highlight syntax errors as you type. For instance, the `linter-php1` package in GitHub's Atom editor uses this method for live linting of PHP code.

- `-m` *or* `--modules`: This lists all the loaded PHP and Zend modules/extensions. These are modules that PHP has been compiled with and may include things such as `core`, `mysql`, `PDO`, `json`, and more. This is useful for checking the PHP installation has the functionality that your application requires. You can also check from within your scripts using the `extension_loaded()` function or by calling the `phpinfo()` function. `-m` provides a subset of the information given with the `-i` flag described earlier, and `-i` (or the `phpinfo()` function) will return more information about the configuration, version, and so on, of the modules.
- `-r` *or* `--run`: This runs a line of PHP code supplied as the argument, rather than executing it from a file. The line of code should be enclosed by single quotes because shells like bash will try to interpolate PHP variables as if they were shell variables if you use double quotes. This performs a similar role to the `-a` interactive mode, except that PHP's "state" is cleared after each line is executed. This means that the line of code supplied is treated as the whole script to be executed, and execution is terminated once it has been run. Here's an example that will print out "4" followed by a new line character:

  ```
  ~$ php -r "echo (2+2).\"\n\";"
  ```

  Note that the line must be well-formed syntactically correct PHP, so don't miss the semicolon at the end! I will return to `-r` later in this chapter in the section "The Many Ways to Call PHP Scripts."
- `-B` *or* `--process-begin`

  `-R` *or* `--process-code`

  `-F` *or* `--process-file`

  `-E` *or* `--process-end`: These four arguments allow you to specify PHP code to be executed before, during, and after input from `STDIN` is processed by PHP. `-B` specifies a line of code to execute before the input is processed, `-R` specifies a line of code to execute for every line of input, and `-F` specifies a PHP file to execute for each line. Finally, `-E` executes a line of code at the end of the input process. In `-R` and `-F`, two special variables are available; `$argn` sets the text of the line being processed, and `$argi` sets the number of the line being processed. This is mainly useful when using PHP directly in shell scripts. For instance, to print a text file with line numbers, you can do something like this:

```
~$  more my_text_file.txt | php -B "echo \"Lets add line
numbers...\n\";" -R "echo \"$argi: $argn\n\";" -E "echo \"That's
the end folks\n\";"
```

This code will output something like this:

```
Lets add line numbers...
1: Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do
2: eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad
3: minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip
4: ex ea commodo consequat. Duis aute irure dolor in reprehenderit in
That's the end folks
```

- `-s` *or* `--syntax-highlight`: This outputs an HTML version of the PHP script, with colored syntax highlighting. The PHP script is not executed or validated; it's simply made "pretty." The pretty HTML is printed to `STDOUT` and can be useful when pouring over code looking for errors, issues, and optimizations. This works only with PHP in files, not with code provided by the `-r` option. Most modern IDEs and code editors provide syntax highlighting by default; however, this can be useful if your only access to a machine is on the command line and the editor you are using doesn't do syntax highlighting. In this case, use `-s` to create a colored version of your script and either download it or view it through your web browser if the machine has a web server installed.

- `-v` *or* `--version`: This outputs the PHP version information. This can also be found in the output of the `-i` option described earlier. Be careful when assuming a particular format; some package repositories (Ubuntu, for instance) include their name and their own build numbers in the version string, so don't just filter it for any numerics.

- `-w` *or* `--strip`: This outputs the contents of the source code with any unnecessary white space and any comments removed. This can be used only with code files (not with lines of code supplied by `-r`) and does not work with the syntax highlighting option shown earlier. This is used to "minify" a file, in other words, reduce the file size. Contrary to popular opinion, this will not speed up most scripts; the overhead of parsing comments and white space is extremely negligible. You should also be wary of support and debugging issues, even if a copy of the "full" code is kept, as line numbers in error reports will no longer match between the original and stripped versions. It also does not minify identifies such as variable names and so cannot be used to obfuscate your code. There are few reason to use this option these days. To make a file smaller for distribution, using proper compression (for example, adding it to a zip file) is usually a better method.

- `-z` *or* `--zend-extension`: This specifies the file name/path for a Zend extension to be loaded before your script is run. This allows dynamic loading of extensions, which can alternatively be specified in the `php.ini` file if they are always to be loaded.

- `--rf` *or* `--rfunction`

  `--rc` *or* `--rclass`

  `--re` *or* `--rextension`

  `--rz` *or* `--rzendextension`

  `--ri` *or* `--rextinfo`: These options allow you to explore PHP structures using reflection. Reflection is the process by which PHP can perform runtime *introspection*, which is the means to allow you to look into elements and structures of your code at run time. The first three options print reflection information about a named function, class, or extension. The last two print basic information about a Zend extension or a standard extension, as returned by the `phpinfo()` function. This reflection information, which is very detailed, is available only if PHP is compiled with reflection support. These options can be used as a quick but precise reference guide to the entities listed earlier and are particularly useful in interrogating unknown code written by others.

Further Reading

- Reflection information in the PHP manual
  - *http://www.php.net/manual/en/book.reflection.php*
- "Introspection and Reflection in PHP" by Octavia Anghel
  - *http://www.sitepoint.com/introspection-and-reflection-in-php/*

Command-Line Arguments for Your Script

As you've seen, passing arguments to PHP is straightforward and done in the normal way. However, passing arguments for use by your PHP script is a little more complicated, as PHP needs to know where its own arguments stop and where your script's start. The best way to examine how PHP deals with this is through some examples. Consider the following PHP script:

```
<?

echo "Number of arguments given :".$argc."\n";

echo "List of arguments given :\n";

print_r($argv);
```

There are two special variables in the previous script.

- `$argc`: This records the number of command-line arguments passed to the script.
- `$argv`: This is an array of the actual arguments passed.

Let's save the script as `arguments.php`. Now let's call it as follows:

```
~$ php -e arguments.php -i -b=big -l red white "and blue"
```

You will get the following output:

```
Number of arguments given :7
List of arguments given :
Array
(
    [0] => arguments.php
    [1] => -i
    [2] => -b=big
    [3] => -l
    [4] => red
    [5] => white
    [6] => and blue
)
```

As you can see, all the arguments given from the file name onward in the command are passed to the script. The first, `-e`, which is used by PHP itself, is not passed through. So, as a general rule, everything after the file name is treated as an argument to the script, anything before the file name is treated as an argument for PHP itself, and the file name is shared between the two.

There is, of course, an exception. As you learned earlier, in addition to specifying the file name of your script on its own, you can also pass it as part of the `-f` flag. So if you execute the following command

```
~$ php -e -f arguments.php -i -b=big -l red white "and blue"
```

you get the following unexpected output:

```
phpinfo()
PHP Version => 5.4.6-1ubuntu1.3

System => Linux dev-system 3.5.0-37-generic #58-Ubuntu SMP Mon Jul 8
22:10:28 UTC 2013 i686
Build Date => Jul 15 2013 18:23:34
Server API => Command Line Interface
Virtual Directory Support => disabled
Configuration File (php.ini) Path => /etc/php5/cli
<rest of output removed for brevity>
```

You may recognize this as the output of calling `php -i`. Rather than treating arguments after the file name as belonging to the script, PHP has treated the `-i` argument (and those afterward) as one of its own. As `-i` is a valid PHP argument, it decides that it was what you wanted and invokes its "information" mode. If you need to pass the file name as part of the `-f` flag rather than as an argument on its own, you will need to separate your scripts arguments using two dashes (`--`). So, for the previous command to work as expected, you need to alter it to read as follows:

```
~$ php -e -f arguments.php -- -i -b=big -l red white "and blue"
```

Everything after the `--`, plus the script file name, is passed as arguments to the script, and you get the expected output.

This can make your scripts a little messy, particularly if you are passing lots of arguments, so you may want to look at the sections below on self executing scripts, which show you how to embed PHPs arguments within the script, allowing the script to claim any and all arguments passed as its own.

The Many Ways to Call PHP Scripts

As you can probably tell from the command-line options in the previous section, there are several ways to execute PHP code when using the CLI SAPI. Although I've covered a couple of these already, I will discuss them here again for completeness.

From a File

You can tell PHP to execute a particular PHP source code file. Here's an example:

```
~$ php myscript.php
~$ php -f myscript.php
```

Note that `-f` is optional; the previous two lines are functionally equivalent. The PHP command-line options detailed earlier, where appropriate, work in this method. This example

```
~$ php -e myscript.php
```

will execute the file `myscript.php` in Extended Information Mode.

As with the web version of PHP, source files can be interpolated (mixed) with HTML (or, more usefully on the command line, plain text). So, you will still need your opening `<?` or `<?php` tags; otherwise, your source code will just be printed straight out without being executed.

From a String

You can execute a single line of code with the `-r` flag, as shown here:

```
~$ php -r "echo(\"Hello World!\n\");"
```

Many of the other command-line options are not available with the `-r` method, such as syntax highlighting. Watch out for shell variable substitution (use single quotes rather than double quotes around your code) and other mangling of your code by the shell. Unless it really is a quick one-off, it is likely safer and easier to pop the relevant line into a file and execute that instead. One common use of the `-r` option is for executing PHP generated by other (possibly non-PHP) shell commands where the whole shell script needs to execute in memory without touching the disk (for instance, where permissions prohibit disk write access).

From STDIN

If you do not specify a file or use the `-r` option, PHP will treat the contents of STDIN as the PHP code to be executed, as shown here (note `echo` only works like this on Linux or OS X):

```
~$ echo '<? echo "hello\n";?>' | php
```

You can also use this method with `-B`, `-R`, `-F`, and `-E` to make PHP a first-class citizen in shell scripting, giving you the ability to pipe data in and out of PHP. For instance, to reverse every line of a file (or any data source that you pipe into it), on Linux or OS X use the following:

```
~$ cat file.txt | php -R 'echo strrev($argn)."\n";' | grep olleh
```

On Windows use the following:

```
> more file.txt | php -R "echo strrev($argn).\"\n\";" | findstr olleh
```

In this line of code, you pipe the contents of a text file into PHP. The `-R` option tells PHP to execute the following PHP code on each line of input, where the line is stored in the special variable `$argn`. In this case, you reverse `$argn` using the string-reversing function `strrev()` and then echo the reversed string out again. Any `echo`'d output goes to `STDOUT`, which either is printed to the shell or, as in this case, can be piped to another shell command. In this case, you then use `grep` to display only the lines containing the string `olleh`, which is *hello* backward. You can find more details on `-R` and its siblings in the previous section.

If you want to use options like `-R` but have too much PHP code to fit comfortably on the command line, you can put the code in a normal PHP source code file and include it with `include()`. Here's an example:

```
~$ cat something.txt > php -R 'include("complicated.php");'
```

If it is a nontrivial PHP script, it may be more efficient to package it up into functions and include it once with `-B` (`-B` means it's executed before the main code) and then execute the function each time with `-R`. The following example loads the content of `my_functions.php` once at the start, and then the function `complicated()` from that file is called on each line (each `$argn`) from the data file (`data.txt`).

```
~$ php -B 'include("my_functions.php");' -R 'complicated($argn);' -f
'data.txt'
```

Although these commands look relatively simple, there is of course no arbitrary limit to the PHP code you can put behind them. You can use classes and objects, multiple files, and most of the code and techniques explored in this book, exposing only functions or methods at the shell level as an interface for the user. You can also open the standard streams as PHP streams within PHP and access their file pointers to read data in from, negating the need to use `-R`, as discussed in the next chapter.

As a Self-executing Script: Unix/Linux

On Unix/Linux systems you can turn a PHP script file into a directly executable shell command. Simply make the first line of the script file a `#!` line (usually pronounced "shebang line" or "hashbang line") with a path to the PHP binary, as in this example:

```
#!/usr/bin/php
<?

echo('Hello World!');
```

Then set the executable bit using `chmod` or similar. Here's an example:

```
~$ chmod a+x myscript.php
```

Simply typing `./myscript.php` at the command line will execute it. You can also rename the file to remove the `.php` extension (assuming you had one in the first place), so you would just type the following at the shell prompt to run it:

```
~$ ./myscript
```

You can further simplify it to remove the initial `./` by moving it to a directory somewhere in your shell's search path. Note that when running a script in this manner, any command-line options are passed directly to the script and not to PHP. In fact, you cannot pass extra command-line parameters to PHP at runtime using this method; you must include them in the shebang line when constructing your script. For instance, in the previous example, if you wanted to use Extended Information Mode, you would alter the first line of the script to read as follows:

```
#!/usr/bin/php -e
```

If you were to instead call the script as follows

```
~$ myscript -e
```

then the `-e` flag would be passed as an argument to the script, not to PHP directly, and so PHP would not enter EIM. This is useful for scripts that have lots of user-supplied arguments but also makes options like `-B` and `-R` discussed in the previous method cumbersome to use for processing `STDIN` data because you have to include all the PHP on the shebang line where it is harder to change. However, you can simply use `include()` to include the necessary files and use standard file streams to process the `STDIN` stream (created and opened by the CLI SAPI automatically for you) line by line instead.

If your script may be used on other systems, please bear in mind that the PHP binary will quite often be located in a different directory than the one on your system. In this scenario, you will need to change the shebang line for each system if you hard-code the location in it. Fortunately, if installed correctly, PHP sets an environment variable with its location, available via the `/usr/bin/env` shell command. So if you change the shebang line as follows, your script should be executable wherever PHP is located:

```
#!/usr/bin/env php
```

On Windows, the shebang line can be left in because PHP will recognize it and ignore it. However, it will not execute the file as it does on *nix.

Further Reading

- Standard I/O streams information in the PHP manual
    - *http://php.net/manual/en/features.commandline.io-streams.php*

As a Self-executing Script: Windows

In a similar manner, scripts can be executed by calling them directly under Windows. However, the process for setting up Windows to do this is slightly more involved.

First, you need to add your PHP directory (the directory containing `php.exe`, `php-win.exe`, or `php-cli.exe`) to the Windows search path (specified in the environment variable `PATH`) so that you can call PHP without having to specify the full directory path. To do this, follow these steps:

1. From the Start menu, go to the Control Panel and select the System icon from the System and Security group.
2. On the Advanced tab, click the Environment Variables button.
3. In the System Variables pane, find the Path entry (you may need to scroll to find it).
4. Double-click the Path entry to edit it and add your PHP directory at the end, including a semicolon (;) before it (for example, `;C:\php`). Make sure that you do not overwrite or remove any of the text already in the path box.

You also need to amend the `PATHEXT` environment variable in the same way, so find the `PATHEXT` entry in the same window and add `.PHP`, again using a semicolon to separate it from the rest of the entries while taking care not to alter them.

Next you need to associate the `.php` file extension with a file type and then tell Windows which program to run for files of that type. To do this, run the following two commands in the Windows command prompt, which you should run as administrator. Make sure to change the path/file name in the second command to match your installation.

```
assoc .php=phpfile
ftype phpfile="C:\PHP5\php.exe" -f "%1" -- %~2
```

These changes will allow you to run `myscript` rather than `C:\php\php.exe myscript.php`. Note that under Windows 10 you will not be able to run scripts in this way in an elevated (administrator) command prompt because the PHP executable is not run as administrator by default. To fix this, right-click the `php.exe` executable, select Properties and Compatibility, and select “Run this program as an administrator” in Settings. Apply the change to all users. Scripts should now execute as expected in all command prompts.

Windows php-win.exe

PHP for Windows also ships with `php-win.exe`, which is similar to the CLI build of PHP, except that it does not open a command-line window. This is useful for running system software in the background or running scripts that create their own graphical interface.

Quitting Your Script

You've looked at starting your scripts, but what happens when it comes time to finish running them?

Like web-based PHP scripts, CLI scripts will terminate happily when you hit the end of the script file and will tidy up all the resources used in the same way. Likewise, if you want to end early, you can call the `exit` (or equivalent `die`) language construct.

However, in the world of CLI scripts, this isn't considered very polite. Because CLI command are designed to work together, often in chains of commands, most shell programs and scripts will provide an "exit code" when they terminate to let the other programs around them know *why* they finished. Were they done? Did they encounter an error? Were they called incorrectly? Inquiring minds want to know.

It is particularly important to supply an exit code when your script may be the last item in a shell script, as the exit code of the shell script as a whole is taken to be the last exit code returned within it. You can make your PHP script provide an exit code simply by including it as a parameter to `exit` or `die`. An exit code is an integer, and there are a number of common exit codes.

- 0: Success. You've exited normally.
- 1: General error. This is usually used for application/language-specific errors and syntax errors.
- 2: Incorrect usage.
- 126: Command is not executable. This is usually permissions related.
- 127: Command not found.
- 128+N (up to 165): Command terminated by POSIX signal number N. For example, in the case of `kill -9 myscript.php`, it should return code 137 (128+9).
- 130: Command terminated by Ctrl-C (Ctrl-C is POSIX code 2, so, as earlier, 128 + 2 = 130).

Further Reading

- "POSIX signals" on Wikipedia
  - http://en.wikipedia.org/wiki/Unix_signal#POSIX_signals

Any other positive integer is generally construed as exiting because of an unspecified error. So, for instance, if you decide the command-line arguments provided by your user are not in the correct format, you should terminate your script using `exit(2)`. If instead all goes well and your script continues to the end of its script file, you can actually let it exit by itself (or by calling `exit` without a parameter) because it returns status code 0 by default.

As with web scripts, you can register functions to be executed when your PHP script exits using the `register_shutdown_function()` function. One use for this may be to check that all is well and evaluate which exit code should be returned. The exit code used as the parameter to `exit` or `die` within a registered shutdown function overrides the code used in the initial exit call that initiated shutdown. This means you can happily exit with `exit(0)` everywhere and then exit with `exit(76)` from your shutdown function if you detect that the foo conflaganation isn't aligned with the bar initispations in your metaspacialatific object. Or similar.

PHP REPLs

When you want to test a few lines of PHP, your default instinct may be to create a new PHP file, save it, and then execute it with PHP. There is a better, faster, and more interactive way, however. The PHP "interactive shell," also known as the PHP REPL, is a quick and easy way to type in code and have it execute immediately. Unlike executing single lines of code using `php -r`, the REPL (started by calling `php -a`) keeps the script's state (for example, contents of variables and objects) in between each line that you type until you exit. You can use all of PHP's functions, although no libraries are loaded by default, and you can use `include()` or `require()` to include existing files of PHP code. This latter capability is useful for debugging the final output of a problematic script; simply use `include()` to include your script, which will execute the script, and as long the script doesn't terminate prematurely, then you can use `echo()` or `print_r()` (or otherwise) to explore the state of the variables and other resources at the end of the run. Other brands of REPL are available and are listed later in this section. By its nature, it can also be used as a CLI/shell in its own right, calling other PHP and non-PHP programs as you would in, for instance, a bash shell.

The following example is a capture of an actual interactive REPL session using the standard PHP REPL:

```
~$ php -a
Interactive shell

php > # As we can type any valid PHP, I have added comments
php > # directly to the REPL, rather than afterwards in editing!
php >
php > # Lets start with some simple assignments :
php >
php > $a = 5;
php > $b = 6;
php >
php > # The REPL will throw Notices, Warnings and Errors as appropriate,
php > # in real-time :
php >
php > $c = nothingdefined;
PHP Notice:  Use of undefined constant nothingdefined - assumed
'nothingdefined' in php shell code on line 1
php >
php > # Just as with normal PHP source files, we can split commands across
php > # lines. The interpreter only kicks in when it hits the terminating
php > # semicolon :
php >
php > $d
php > =
php > 7
php > ;
php >
php > # The following shows that the state in the variables above has been
php > # kept :
php >
php > echo $a + $b + $c + $d ."\n";
18
php >
php > # Next, a more interesting example. Use the REPL instead of the
php > # shell to get the first line from a file :
php >
php > echo file ('/proc/version')[0];
Linux version 3.5.0-21-generic (buildd@roseapple) (gcc version 4.7.2
(Ubuntu/Linaro 4.7.2-2ubuntu1) ) #32-Ubuntu SMP Tue Dec 11 18:52:46 UTC
2012
php >
php > # Of course all of the usual protocol wrappers are available, so we
php > # can see what is happening in the world...
php >
php > $page = file ('http://news.bbc.co.uk');
php >
php > echo $page[0];
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML+RDFa 1.0//EN" "http://www.w3.org/
MarkUp/DTD/xhtml-rdfa-1.dtd">
php >
php > # and maybe get a hash of that...
php >
php > echo md5 ( implode ( $page, "\n" ) ) . "\n";
0319bf4e62db39fb2c89210e48783d70
php >
php > # when we are done ...
php >
php > exit;
php >
php > # doesn't work, as its just evaluated as PHP (and the REPL ignores
php > # exit/die calls. To exit the REPL, enter the word 'exit' on its own
php > # on a new line
php >
php > exit
~$
```

Sometimes you'll want to execute your commands within the "environment" of other scripts. For instance, you may have a script that declares constants, sets up database connections, and does other routine tasks that you normally include with `include()` at the start of your main PHP scripts. As noted earlier, you can include these files in the REPL too using `include()`, but you may forget to do so and then wonder why things didn't work as they should. One facility PHP offers you, which applies not only to the REPL but to all forms of PHP execution, is the `auto_prepend_file` configuration directive. This tells PHP to execute a given file each time PHP is run before it starts to do anything else (such as executing the script you have asked it to execute). This can be set either in `php.ini` or via the `-d` flag on the command line. The following is an example of presetting some constants/variables. First, you create a script called `initialise.php` with the following content:

```
<?php

const FOUR = 4; # Declare a constant value

$five = 5; # Instantiate a variable with another value
```

Then, at the command line, start and run a REPL session as follows, using `-d` to execute the `initialise.php` script first:

```
~$ php -d auto_prepend_file=initialise.php -a
Interactive shell

php > echo (FOUR + $five)."\n";
9
php > exit
~$
```

As you can see, the constant and variable you had set up in the `initialise.php` file were available for use from the REPL without having to manually declare them. The `-d` flag is used here, but the option could be set in `php.ini` as well if you want to always use the same file. If you regularly use a few different initialization files like this, you can create shell aliases to commands using the `-d` flag. For instance, you could add lines similar to the following to your `~/.bash_profile`:

```
alias php-cl="php -d auto_prepend_file=clientSetup.php -a"
alias php-in="php -d auto_prepend_file=ourSiteSetup.php -a"
```

As well as the built-in PHP REPL explored earlier, there are a number of third-party REPLs available, some of which include features such as a history of commands typed, tab-completion of commands, protection from fatal errors, and even abbreviated function documentation.

Toolbox	phpsh

Developed at Facebook, phpsh is an interactive shell for PHP that features readline history, tab completion, and quick access to documentation.

Main website and documentation	`http://phpsh.org`
Installation information	`https://github.com/facebook/phpsh/blob/master/README.md`

Toolbox	Boris

A small but robust REPL for PHP

Main documentation and installation information	`https://github.com/d11wtq/boris`
Extension for Symfony and Drupal	`http://vvv.tobiassjosten.net/php/php-repl-for-symfony-and-drupal/`

Toolbox	phpa

A simple replacement for `php -a`, written in PHP

Main website, installation information and documentation	`http://david.acz.org/phpa/`

Toolbox	PHP Interactive

A web-based REPL that allows better support of displaying HTML output. The project is an Alpha release.

Main website	`http://www.hping.org/phpinteractive/`

Toolbox	Sublime-worksheet

An inline REPL for the Sublime Text editor

Main website	`https://github.com/jcartledge/sublime-worksheet`

Toolbox	iPHP

An extensible PHP shell

Main website	`https://github.com/apinstein/iphp`