
lib537
Release 1.0a1

Chad W. L. Whitacre

December 4, 2006

Zeta Design & Development
<http://www.zetadev.com/software/lib537/>
Email: chad@zetaweb.com

Abstract

lib537 is a Python library.

Installation

lib537 can be installed using either `distutils` or `setuptools`. That is, you can either download a tarball, unpack it, and run:

```
$ python setup.py install
```

Or you can run:

```
$ easy_install lib537
```


Library Reference Manual

2.1 `httpy` — Smooth out WSGI’s worst warts

The request side of WSGI—the “commons” of the *environ* mapping—is quite nice. It honors the tradition of CGI, and it’s just a mapping. Simple.

The response-side API is a little stiffer, because WSGI has to support edge cases like serving large files, complex exception handling, and HTTP/1.1 features. This results in warts like `start_response`, and the requirement that apps return an iterable. The intention in [PEP 333](#) is that these warts be smoothed over at other layers; this is such a layer.

`httpy` provides the following classes:

class `Responder` (*app*)

Constructs a new `Responder` object. *app* is an extended WSGI application: it may alternately return a string, or return or raise a `Response` object.

class `Response` (*[code]* [*, body*] [*, headers*])

Constructs a new `Response` object. If given, *code* must be an integer; the default is `200` (see [the HTTP spec](#) for other values that will be meaningful to most HTTP clients). *body* must be a string. *headers* may be a dictionary or a list of 2-tuples. *body* is second rather than *headers* because one more often wants to specify a body without headers than vice versa.

2.1.1 Responder Objects

Instances of `httpy.Responder` are callables that speak plain WSGI on the server side, but they also accept strings and `Response` objects from the application side. When a `Responder` receives a `Response` object, that becomes the WSGI endpoint. When a `Responder` receives a string, it creates a `Response` object with the string as the *body*, and the `Content-Type`: set to `text/html`. This implicit `Response` then becomes the endpoint.

2.1.2 Response Objects

Instances of `httpy.Response` are plain WSGI applications, with the following data attributes. Note that values are only validated in the constructor, so it is currently possible to return/raise a malformed `Response` by setting instance attributes post-instantiation.

code

The [HTTP code](#) as an integer.

body

The message body as a string.

headers

The message headers as an instance of the standard library's `email.Message.Message`.

When called, `Response` instances call `start_response` with adaptations of `code` and `headers`, and return a one-item list containing `body`.

2.1.3 Example

Here is an example:

```
Python 2.5 (r25:52005, Sep 25 2006, 21:37:36)
[GCC 3.4.4 [FreeBSD] 20050518] on freebsd6
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> import httpy
>>> app = lambda env, start: "Greetings, program!"
>>> app = httpy.Responder(app)
>>>
>>> from wsgiref.simple_server import make_server
>>> server = make_server('', 8080, app)
>>> server.serve_forever() # now hit http://localhost:8080/
>>>
192.168.1.100 - - [09/Nov/2006 23:52:45] "GET / HTTP/1.1" 200 19
```

2.2 mode — Manage the application life-cycle

It is often valuable to maintain a distinction between various phases of an application's lifecycle. The `mode` module calls these phases *modes*, and identifies four of them, given here in conceptual life-cycle order:

Mode	Description
debugging	The application is being actively debugged; exceptions may trigger an interactive debugger.
development	The application is being actively developed; however, exceptions should not trigger interactive debugging.
staging	The application is deployed in a mock-production environment.
production	The application is in live use by its end users.

The expectation is that various aspects of the application—logging, exception handling, data sourcing—will adapt to the current mode. The mode is set in the `PYTHONMODE` environment variable. This module provides API for interacting with this variable. If `PYTHONMODE` is unset, it will be set to `development` when this module is imported.

2.2.1 Members

The module defines the following functions:

`get()`

Return the current `PYTHONMODE` setting as a lowercase string; will raise `EnvironmentError` if the (case-insensitive) setting is not one of `debugging`, `development`, `staging`, or `production`.

set (*mode*)

Given a mode, set the PYTHONMODE environment variable and refresh the module's boolean members. If given a bad mode, ValueError is raised.

setAPI ()

Refresh the module's boolean members. Call this if you ever change PYTHONPATH directly in the os.environ mapping.

The module also defines a number of boolean attributes reflecting the current mode setting, including abbreviations and combinations. Uppercase versions of each of the following are also defined (e.g., DEBUGGING).

debugging, deb

True if PYTHONMODE is set to debugging.

development, dev

True if PYTHONMODE is set to development.

staging, st

True if PYTHONMODE is set to staging.

production, prod

True if PYTHONMODE is set to production.

debugging_or_development, debdev, devdeb

True if PYTHONMODE is set to debugging or development.

staging_or_production, stprod

True if PYTHONMODE is set to staging or production.

2.2.2 Example

Example usage:

```
>>> import mode
>>> mode.set('development')      # can set the mode at runtime
>>> mode.get()                  # and access the current mode
'development'
>>> mode.development            # module defines boolean constants
True
>>> mode.PRODUCTION             # uppercase versions are also defined
False
>>> mode.dev                    # as are abbreviations
True
>>> mode.DEBDEV, mode.stprod    # and combinations
(True, False)
```

2.3 `restarter` — Automatically restart your program when module files change

This module solves the problem of refreshing Python modules in memory when the source files change, without manually restarting the program. There are two basic ways to solve this problem:

1. **Reload modules within a single process.** The basic trick is to delete items from `sys.modules`, forcing a refresh the next time they are loaded. This gets tough because imported modules depend on each other. Look at

Zope and RollbackImporter for two implementations.

2. **Maintain two processes.** In this solution, a parent process continuously restarts a child process. This gets around module import dependencies, but the downside is that you lose any program state on restart, and it magnifies the shutdown/start-up time of your program.

This module implements the second solution.

2.3.1 Members

`restarter` provides the following members:

CHILD, PARENT

These are booleans indicating whether the current process is the parent or child.

launch_child()

Continuously relaunch the current program in a sub-process until the exit code is something other than 75.

should_restart()

Return a boolean indicating whether the child process should be restarted. If called in the parent process, it always returns `False`.

Our implementation uses a thread in the child process (started when the module is imported) to monitor all library source files. Your program is responsible for periodically calling `mods_changed`, exiting with code 75 whenever it returns `True` (presumably after cleanly shutting down). Exit code 75 seemed appropriate to use because of its meaning on UNIX systems (from `/usr/include/sys/exits.h`, FreeBSD 6.1-RELEASE):

```
EX_TEMPFAIL -- temporary failure, indicating something that
*             is not really an error.  In sendmail, this means
*             that a mailer (e.g.) could not create a connection,
*             and the request should be reattempted later.
[...]
#define EX_TEMPFAIL    75        /* temp failure; user is invited to retry */
```

This module requires the `subprocess` module. Included in the standard library since Python 2.4, it can also be found here:

<http://www.lysator.liu.se/~astrand/popen5/>

2.3.2 Example

Here is an example:

```
import restarter

def main():
    while 1:
        # program logic here
        if restarter.should_restart():
            # shutdown code here
            raise SystemExit(75)

if restarter.PARENT:
    restarter.launch_child()
else:
    main()
```


Credits and Legalese

The `restarter` module is based on [work by Ian Bicking and the CherryPy team](#). It is used under the MIT and/or new BSD licenses.

All original work is:

```
Copyright (c) 2006 Chad Whitacre <chad@zetaweb.com>
```

```
Permission is hereby granted, free of charge, to any person obtaining a copy of
this software and associated documentation files (the "Software"), to deal in
the Software without restriction, including without limitation the rights to
use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of
the Software, and to permit persons to whom the Software is furnished to do so,
subject to the following conditions:
```

```
The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.
```

```
THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS
FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR
COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER
IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

```
<http://opensource.org/licenses/mit-license.php>
```

See Also:

[MIT License](#)

The MIT license is used for parts of `restarter` and all original work.

[New BSD License](#)

The new BSD license is used for parts of `restarter`.