Mod_python Manual

Release 3.1.3

Gregory Trubetskoy

February 17, 2004

E-mail: grisha@apache.org

Abstract

Mod_python allows embedding Python within the Apache server for a considerable boost in performance and added flexibility in designing web based applications.

This document aims to be the only necessary and authoritative source of information about mod_python, usable as a comprehensive reference, a user guide and a tutorial all-in-one.

See Also:

Python Language Web Site
(http://www.python.org/)
for information on the Python language

Apache Server Web Site
(http://httpd.apache.org/)
for information on the Apache server

CONTENTS

CHAPTER

ONE

Introduction

1.1 Performance

One of the main advantages of mod_python is the increase in performance over traditional CGI. Below are results of a very crude test. The test was done on a 1.2GHz Pentium machine running Red Hat Linux 7.3. *Ab* was used to poll 4 kinds of scripts, all of which imported the standard cgi module (because this is how a typical Python cgi script begins), then output a single word '`Hello!`'. The results are based on 10000 requests with concurrency of 1.

```
Standard CGI:              23 requests/s
Mod_python cgihandler:    385 requests/s
Mod_python publisher:     476 requests/s
Mod_python handler:      1203 requests/s
```

1.2 Flexibility

Apache processes requests in phases (e.g. read the request, parse headers, check access, etc.). These phases can be implemented by functions called handlers. Traditionally, handlers are written in C and compiled into Apache modules. Mod_python provides a way to extend Apache functionality by writing Apache handlers in Python. For a detailed description of the Apache request processing process, see the *Apache API Notes*, as well as the *Mod_python - Integrating Python with Apache* paper.

To ease migration from CGI, a standard mod_python handler is provided that simulates the CGI environment allowing a user to run legacy scripts under mod_python with no changes to the code (in most cases).

See Also:

http://dev.apache.org/
: Apache Developer Resources

http://www.modpython.org/python10/
: Mod_Python - Integrating Python with Apache, presented at Python 10

1.3 History

Mod_python originates from a project called *Httpdapy* (1997). For a long time Httpdapy was not called mod_python because Httpdapy was not meant to be Apache-specific. Httpdapy was designed to be cross-platform and in fact was initially written for the Netscape server (back then it was called Nsapy (1997).

This excerpt from the Httpdapy README file describes well the challenges and the solution provided by embedding Python within the HTTP server:

```
While developing my first WWW applications a few years back, I found
that using CGI for programs that need to connect to relational
databases (commercial or not) is too slow because every hit requires
loading of the interpreter executable which can be megabytes in size,
any database libraries that can themselves be pretty big, plus, the
database connection/authentication process carries a very significant
overhead because it involves things like DNS resolutions, encryption,
memory allocation, etc.. Under pressure to speed up the application, I
nearly gave up the idea of using Python for the project and started
researching other tools that claimed to specialize in www database
integration. I did not have any faith in MS's ASP; was quite
frustrated by Netscape LiveWire's slow performance and bugginess; Cold
Fusion seemed promising, but I soon learned that writing in html-like
tags makes programs as readable as assembly. Same is true for
PHP. Besides, I *really* wanted to write things in Python.

Around the same time the Internet Programming With Python book came
out and the chapter describing how to embed Python within Netscape
server immediately caught my attention.  I used the example in my
project, and developed an improved version of what I later called
Nsapy that compiled on both Windows NT and Solaris.

Although Nsapy only worked with Netscape servers, it was a very
intelligent generic OO design that, in the spirit of Python, that lent
itself for easy portability to other web servers.

Incidently, the popularity of Netscape's servers was taking a turn
south, and so I set out to port Nsapy to other servers starting with
the most popular one, Apache. And so from Nsapy was born Httpdapy.
```

...continuing this saga, yours truly later learned that writing Httpdapy for every server is a task a little bigger and less interesting than I originally imagined.

Instead, it seemed like providing a Python counterpart to the popular Perl Apache extension mod_perl that would give Python users the same (or better) capability would be a much more exciting thing to do.

And so it was done. The first release of mod_python happened in May of 2000.

CHAPTER

TWO

Installation

Note: By far the best place to get help with installation and other issues is the mod_python mailing list. Please take a moment to join the mod_python mailing list by sending an e-mail with the word '`subscribe`' in the subject to mod_python-request@modpython.org.

2.1 Prerequisites

- Python 2.2.1 or later. Earlier versions of Python will not work.
- Apache 2.0.40 or later (For Apache 1.3.x, use mod_python version 2.7.x).

In order to compile mod_python you will need to have the include files for both Apache and Python, as well as the Python library installed on your system. If you installed Python and Apache from source, then you already have everything needed. However, if you are using prepackaged software (e.g. Red Hat Linux RPM, Debian, or Solaris packages from sunsite, etc) then chances are, you have just the binaries and not the sources on your system. Often, the Apache and Python include files and libraries necessary to compile mod_python are part of separate "development" package. If you are not sure whether you have all the necessary files, either compile and install Python and Apache from source, or refer to the documentation for your system on how to get the development packages.

2.2 Compiling

There are two ways in which modules can be compiled and linked to Apache - statically, or as a DSO (Dynamic Shared Object).

DSO is a more popular approach nowadays and is the recommended one for mod_python. The module gets compiled as a shared library which is dynamically loaded by the server at run time.

The advantage of DSO is that a module can be installed without recompiling Apache and used as needed. A more detailed description of the Apache DSO mechanism is available at http://httpd.apache.org/docs-2.0/dso.html.

At this time only DSO is supported by mod_python.

Static linking is an older approach. With dynamic linking available on most platforms it is used less and less. The main drawback is that it entails recompiling Apache, which in many instances is not a favorable option.

2.2.1 Running ./configure

The **./configure** script will analyze your environment and create custom Makefiles particular to your system. Aside from all the standard autoconf stuff, **./configure** does the following:

- Finds out whether a program called **apxs** is available. This program is part of the standard Apache distribution, and is necessary for DSO compilation. If apxs cannot be found in your PATH or in /usr/local/apache/bin, DSO compilation will not be available.

  You can manually specify the location of apxs by using the **--with-apxs** option, e.g.:

  ```
  $ ./configure --with-apxs=/usr/local/apache/bin/apxs
  ```

  It is recommended that you specify this option.

- Checks your Python version and attempts to figure out where **libpython** is by looking at various parameters compiled into your Python binary. By default, it will use the **python** program found in your PATH.

  If the fist Python binary in the path is not suitable or not the one desired for mod_python, you can specify an alternative location with the **--with-python** options, e.g:

  ```
  $ ./configure --with-python=/usr/local/bin/python2.2
  ```

2.2.2 Running make

- To start the build process, simply run

  ```
  $ make
  ```

2.3 Installing

2.3.1 Running make install

- This part of the installation needs to be done as root.

  ```
  $ su
  # make install
  ```

  - This will simply copy the library into your Apache libexec directory, where all the other modules are.
  - Lastly, it will install the Python libraries in site-packages and compile them.

  NB: If you wish to selectively install just the Python libraries or the DSO (which may not always require superuser privileges), you can use the following **make** targets: **install_py_lib** and **install_dso**

2.3.2 Configuring Apache

- If you compiled mod_python as a DSO, you will need to tell Apache to load the module by adding the following line in the Apache configuration file, usually called httpd.conf or apache.conf:

  ```
  LoadModule python_module libexec/mod_python.so
  ```

  The actual path to **mod_python.so** may vary, but make install should report at the very end exactly where **mod_python.so** was placed and how the `LoadModule` directive should appear.

2.4 Testing

1. Make some directory that would be visible on your web site, for example, htdocs/test.

2. Add the following Apache directives, which can appear in either the main server configuration file, or .htaccess. If you are going to be using the .htaccess file, you will not need the `<Directory>` tag below (the directory then becomes the one in which the .htaccess file is located), and you will need to make sure the `AllowOverride` directive applicable to this directory has at least `FileInfo` specified. (The default is `None`, which will not work.)

```
<Directory /some/directory/htdocs/test>
    AddHandler mod_python .py
    PythonHandler mptest
    PythonDebug On
</Directory>
```

   (Substitute /some/directory above for something applicable to your system, usually your Apache ServerRoot)

3. At this time, if you made changes to the main configuration file, you will need to restart Apache in order for the changes to take effect.

4. Edit mptest.py file in the htdocs/test directory so that is has the following lines (be careful when cutting and pasting from your browser, you may end up with incorrect indentation and a syntax error):

```
from mod_python import apache

def handler(req):
    req.write("Hello World!")
    return apache.OK
```

5. Point your browser to the URL referring to the mptest.py; you should see '`Hello World!`'. If you didn't - refer to the troubleshooting section next.

6. If everything worked well, move on to Chapter 3, *Tutorial*.

2.5 Troubleshooting

There are a few things you can try to identify the problem:

- Carefully study the error output, if any.
- Check the server error log file, it may contain useful clues.
- Try running Apache from the command line in single process mode:

```
./httpd -X
```

  This prevents it from backgrounding itself and may provide some useful information.

- Ask on the mod_python list. Make sure to provide specifics such as:
  - Mod_python version.
  - Your operating system type, name and version.
  - Your Python version, and any unusual compilation options.

- Your Apache version.
- Relevant parts of the Apache config, .htaccess.
- Relevant parts of the Python code.

CHAPTER

THREE

Tutorial

So how can I make this work?

This is a quick guide to getting started with mod_python programming once you have it installed. This is ***not*** *an installation manual!*

It is also highly recommended to read (at least the top part of) Section 4, Python API *after completing this tutorial.*

3.1 A Quick Start with the Publisher Handler

This section provides a quick overview of the Publisher handler for those who would like to get started without getting into too much detail. A more thorough explanation of how mod_python handlers work and what a handler actually is follows on in the later sections of the tutorial.

The `publisher` handler is provided as one of the standard mod_python handlers. To get the publisher handler working, you will need the following lines in your config:

```
AddHandler mod_python .py
PythonHandler mod_python.publisher
PythonDebug On
```

The following example will demonstrate a simple feedback form. The form will ask for the name, e-mail address and a comment and construct an e-mail to the webmaster using the information submitted by the user. This simple application consists of two files: form.html - the form to collect the data, and form.py - the target of the form's action.

Here is the html for the form:

```
<html>
    Please provide feedback below:
<p>
<form action="form.py/email" method="POST">

    Name:    <input type="text" name="name"><br>
    Email:   <input type="text" name="email"><br>
    Comment: <textarea name="comment" rows=4 cols=20></textarea><br>
    <input type="submit">

</form>
</html>
```

Note the `action` element of the `<form>` tag points to `form.py/email`. We are going to create a file called form.py, like this:

```
import smtplib

WEBMASTER = "webmaster"   # webmaster e-mail
SMTP_SERVER = "localhost" # your SMTP server

def email(req, name, email, comment):

    # make sure the user provided all the parameters
    if not (name and email and comment):
        return "A required parameter is missing, \
               please go back and correct the error"

    # create the message text
    msg = """\
From: %s
Subject: feedback
To: %s

I have the following comment:

%s

Thank You,

%s

""" % (email, WEBMASTER, comment, name)

    # send it out
    conn = smtplib.SMTP(SMTP_SERVER)
    conn.sendmail(email, [WEBMASTER], msg)
    conn.quit()

    # provide feedback to the user
    s = """\
<html>

Dear %s,<br>
Thank You for your kind comments, we
will get back to you shortly.

</html>""" % name

    return s
```

When the user clicks the Submit button, the publisher handler will load the `email` function in the `form` module, passing it the form fields as keyword arguments. It will also pass the request object as `req`.

Note that you do not have to have `req` as one of the arguments if you do not need it. The publisher handler is smart enough to pass your function only those arguments that it will accept.

The data is sent back to the browser via the return value of the function.

Even though the Publisher handler simplifies mod_python programming a great deal, all the power of mod_python is still available to this program, since it has access to the request object. You can do all the same things you can do with a "native" mod_python handler, e.g. set custom headers via `req.headers_out`, return errors by raising `apache.SERVER_ERROR` exceptions, write or read directly to and from the client via `req.write()` and `req.read()`, etc.

Read Section 6.1 *Publisher Handler* for more information on the publisher handler.

3.2 Quick Overview of how Apache Handles Requests

If you would like delve in deeper into the functionality of mod_python, you need to understand what a handler is.

Apache processes requests in *phases*. For example, the first phase may be to authenticate the user, the next phase to verify whether that user is allowed to see a particular file, then (next phase) read the file and send it to the client. A typical static file request involves three phases: (1) translate the requested URI to a file location (2) read the file and send it to the client, then (3) log the request. Exactly which phases are processed and how varies greatly and depends on the configuration.

A *handler* is a function that processes one phase. There may be more than one handler available to process a particular phase, in which case they are called by Apache in sequence. For each of the phases, there is a default Apache handler (most of which by default perform only very basic functions or do nothing), and then there are additional handlers provided by Apache modules, such as mod_python.

Mod_python provides every possible handler to Apache. Mod_python handlers by default do not perform any function, unless specifically told so by a configuration directive. These directives begin with '`Python`' and end with '`Handler`' (e.g. `PythonAuthenHandler`) and associate a phase with a Python function. So the main function of mod_python is to act as a dispatcher between Apache handlers and Python functions written by a developer like you.

The most commonly used handler is `PythonHandler`. It handles the phase of the request during which the actual content is provided. Because it has no name, it is sometimes referred to as as *generic* handler. The default Apache action for this handler is to read the file and send it to the client. Most applications you will write will override this one handler. To see all the possible handlers, refer to Section 5, *Apache Directives*.

3.3 So what Exactly does Mod-python do?

Let's pretend we have the following configuration:

```
<Directory /mywebdir>
    AddHandler mod_python .py
    PythonHandler myscript
    PythonDebug On
</Directory>
```

NB: /mywebdir is an absolute physical path.

And let's say that we have a python program (Windows users: substitute forward slashes for backslashes) '/mywedir/myscript.py' that looks like this:

```
from mod_python import apache

def handler(req):

    req.content_type = "text/plain"
    req.write("Hello World!")

    return apache.OK
```

Here is what's going to happen: The `AddHandler` directive tells Apache that any request for any file ending with '.py' in the '/mywebdir' directory or a subdirectory thereof needs to be processed by mod_python. The '`PythonHandler myscript`' directive tells mod_python to process the generic handler using the `myscript` script. The '`PythonDebug On`' directive instructs mod_python in case of an Python error to send error output to the client (in addition to the logs), very useful during development.

When a request comes in, Apache starts stepping through its request processing phases calling handlers in mod_python. The mod_python handlers check whether a directive for that handler was specified in the configuration. (Remember, it acts as a dispatcher.) In our example, no action will be taken by mod_python for all handlers

except for the generic handler. When we get to the generic handler, mod_python will notice '`PythonHandler myscript`' directive and do the following:

1. If not already done, prepend the directory in which the `PythonHandler` directive was found to `sys.path`.
2. Attempt to import a module by name `myscript`. (Note that if `myscript` was in a subdirectory of the directory where `PythonHandler` was specified, then the import would not work because said subdirectory would not be in the `sys.path`. One way around this is to use package notation, e.g. '`PythonHandler subdir.myscript`'.)
3. Look for a function called `handler` in `myscript`.
4. Call the function, passing it a request object. (More on what a request object is later)
5. At this point we're inside the script:

   - ```
     from mod_python import apache
     ```

     This imports the apache module which provides us the interface to Apache. With a few rare exceptions, every mod_python program will have this line.
   - ```
     def handler(req):
     ```

     This is our *handler* function declaration. It is called '`handler`' because mod_python takes the name of the directive, converts it to lower case and removes the word '`python`'. Thus '`PythonHandler`' becomes '`handler`'. You could name it something else, and specify it explicitly in the directive using '`::`'. For example, if the handler function was called '`spam`', then the directive would be '`PythonHandler myscript::spam`'.
     Note that a handler must take one argument - the request object. The request object is an object that provides all of the information about this particular request - such as the IP of client, the headers, the URI, etc. The communication back to the client is also done via the request object, i.e. there is no "response" object.
   - ```
     req.content_type = "text/plain"
     ```

     This sets the content type to '`text/plain`'. The default is usually '`text/html`', but since our handler doesn't produce any html, '`text/plain`' is more appropriate.
   - ```
     req.write("Hello World!")
     ```

     This writes the '`Hello World!`' string to the client. (Did I really have to explain this one?)
   - ```
     return apache.OK
     ```

     This tells Apache that everything went OK and that the request has been processed. If things did not go OK, that line could be return `apache.HTTP_INTERNAL_SERVER_ERROR` or return `apache.HTTP_FORBIDDEN`. When things do not go OK, Apache will log the error and generate an error message for the client.

Some food for thought: If you were paying attention, you noticed that the text above didn't specify that in order for the handler code to be executed, the URL needs to refer to myscript.py. The only requirement was that it refers to a .py file. In fact the name of the file doesn't matter, and the file referred to in the URL doesn't have to exist. So, given the above configuration, '`http://myserver/mywebdir/myscript.py`' and '`http://myserver/mywebdir/montypython.py`' would give the exact same result. The important thing to understand here is that a handler augments the server behaviour when processing a specific type of file, not an individual file.

At this point, if you didn't understand the above paragraph, go back and read it again, until you do.
     ```

3.4 Now something More Complicated - Authentication

Now that you know how to write a primitive handler, let's try something more complicated.

Let's say we want to password-protect this directory. We want the login to be '`spam`', and the password to be '`eggs`'.

First, we need to tell Apache to call our *authentication* handler when authentication is needed. We do this by adding the `PythonAuthenHandler`. So now our config looks like this:

```
<Directory /mywebdir>
    AddHandler mod_python .py
    PythonHandler myscript
    PythonAuthenHandler myscript
    PythonDebug On
</Directory>
```

Notice that the same script is specified for two different handlers. This is fine, because if you remember, mod_python will look for different functions within that script for the different handlers.

Next, we need to tell Apache that we are using Basic HTTP authentication, and only valid users are allowed (this is fairly basic Apache stuff, so we're not going to go into details here). Our config looks like this now:

```
<Directory /mywebdir>
   AddHandler mod_python .py
   PythonHandler myscript
   PythonAuthenHandler myscript
   PythonDebug On
   AuthType Basic
   AuthName "Restricted Area"
   require valid-user
</Directory>
```

Now we need to write an authentication handler function in 'myscript.py'. A basic authentication handler would look like this:

```
from mod_python import apache

def authenhandler(req):

    pw = req.get_basic_auth_pw()
    user = req.user

    if user == "spam" and pw == "eggs":
       return apache.OK
    else:
       return apache.HTTP_UNAUTHORIZED
```

Let's look at this line by line:

- `def authenhandler(req):`

  This is the handler function declaration. This one is called `authenhandler` because, as we already described above, mod_python takes the name of the directive (`PythonAuthenHandler`), drops the word '`Python`' and converts it lower case.

- `pw = req.get_basic_auth_pw()`

This is how we obtain the password. The basic HTTP authentication transmits the password in base64 encoded form to make it a little bit less obvious. This function decodes the password and returns it as a string. Note that we have to call this function before obtaining the user name.

- ```
  user = req.user
  ```

  This is how you obtain the username that the user entered.

- ```
  if user == "spam" and pw == "eggs":
      return apache.OK
  ```

  We compare the values provided by the user, and if they are what we were expecting, we tell Apache to go ahead and proceed by returning `apache.OK`. Apache will then consider this phase of the request complete, and proceed to the next phase. (Which in this case would be `handler()` if it's a `.py` file).

- ```
  else:
      return apache.HTTP_UNAUTHORIZED
  ```

  Else, we tell Apache to return `HTTP_UNAUTHORIZED` to the client, which usually causes the browser to pop a dialog box asking for username and password.
  ```

CHAPTER

FOUR

Python API

4.1 Multiple Interpreters

When working with mod_python, it is important to be aware of a feature of Python that is normally not used when using the language for writing scripts to be run from command line. This feature is not available from within Python itself and can only be accessed through the *C language API*.

Python C API provides the ability to create *subinterpreters*. A more detailed description of a subinterpreter is given in the documentation for the `Py_NewInterpreter()` function. For this discussion, it will suffice to say that each subinterpreter has its own separate namespace, not accessible from other subinterpreters. Subinterpreters are very useful to make sure that separate programs running under the same Apache server do not interfere with one another.

At server start-up or mod_python initialization time, mod_python initializes an interpreter called *main* interpreter. The main interpreter contains a dictionary of subinterpreters. Initially, this dictionary is empty. With every request, as needed, subinterpreters are created, and references to them are stored in this dictionary. The dictionary is keyed on a string, also known as *interpreter name*. This name can be any string. The main interpreter is named '`main_interpreter`'. The way all other interpreters are named can be controlled by `PythonInterp*` directives. Default behaviour is to name interpreters using the Apache virtual server name (`ServerName` directive). This means that all scripts in the same virtual server execute in the same subinterpreter, but scripts in different virtual servers execute in different subinterpreters with completely separate namespaces. `PythonInterpPerDirectory` and `PythonInterpPerDirective` directives alter the naming convention to use the absolute path of the directory being accessed, or the directory in which the `Python*Handler` was encountered, respectively. `PythonInterpreter` can be used to force the interpreter name to a specific string overriding any naming conventions.

Once created, a subinterpreter will be reused for subsequent requests. It is never destroyed and exists until the Apache process dies.

You can find out the name of the interpreter under which you're running by peeking at `req.interpreter`.

See Also:

Python C Language API
(http://www.python.org/doc/current/api/api.html)
Python C Language API

4.2 Overview of a Request Handler

A *handler* is a function that processes a particular phase of a request. Apache processes requests in phases - read the request, process headers, provide content, etc. For every phase, it will call handlers, provided by either the Apache core or one of its modules, such as mod_python which passes control to functions provided by the user and written in Python. A handler written in Python is not any different from a handler written in C, and follows these rules:

A handler function will always be passed a reference to a request object. (Throughout this manual, the request object is often referred to by the `req` variable.)

Every handler can return:

- `apache.OK`, meaning this phase of the request was handled by this handler and no errors occurred.
- `apache.DECLINED`, meaning this handler has not handled this phase of the request to completion and Apache needs to look for another handler in subsequent modules.
- `apache.`*`HTTP_ERROR`*, meaning an HTTP error occurred. *HTTP_ERROR* can be any of the following:

```
HTTP_CONTINUE                     = 100
HTTP_SWITCHING_PROTOCOLS          = 101
HTTP_PROCESSING                   = 102
HTTP_OK                           = 200
HTTP_CREATED                      = 201
HTTP_ACCEPTED                     = 202
HTTP_NON_AUTHORITATIVE            = 203
HTTP_NO_CONTENT                   = 204
HTTP_RESET_CONTENT                = 205
HTTP_PARTIAL_CONTENT              = 206
HTTP_MULTI_STATUS                 = 207
HTTP_MULTIPLE_CHOICES             = 300
HTTP_MOVED_PERMANENTLY            = 301
HTTP_MOVED_TEMPORARILY            = 302
HTTP_SEE_OTHER                    = 303
HTTP_NOT_MODIFIED                 = 304
HTTP_USE_PROXY                    = 305
HTTP_TEMPORARY_REDIRECT           = 307
HTTP_BAD_REQUEST                  = 400
HTTP_UNAUTHORIZED                 = 401
HTTP_PAYMENT_REQUIRED             = 402
HTTP_FORBIDDEN                    = 403
HTTP_NOT_FOUND                    = 404
HTTP_METHOD_NOT_ALLOWED           = 405
HTTP_NOT_ACCEPTABLE               = 406
HTTP_PROXY_AUTHENTICATION_REQUIRED= 407
HTTP_REQUEST_TIME_OUT             = 408
HTTP_CONFLICT                     = 409
HTTP_GONE                         = 410
HTTP_LENGTH_REQUIRED              = 411
HTTP_PRECONDITION_FAILED          = 412
HTTP_REQUEST_ENTITY_TOO_LARGE     = 413
HTTP_REQUEST_URI_TOO_LARGE        = 414
HTTP_UNSUPPORTED_MEDIA_TYPE       = 415
HTTP_RANGE_NOT_SATISFIABLE        = 416
HTTP_EXPECTATION_FAILED           = 417
HTTP_UNPROCESSABLE_ENTITY         = 422
HTTP_LOCKED                       = 423
HTTP_FAILED_DEPENDENCY            = 424
HTTP_INTERNAL_SERVER_ERROR        = 500
HTTP_NOT_IMPLEMENTED              = 501
HTTP_BAD_GATEWAY                  = 502
HTTP_SERVICE_UNAVAILABLE          = 503
HTTP_GATEWAY_TIME_OUT             = 504
HTTP_VERSION_NOT_SUPPORTED        = 505
HTTP_VARIANT_ALSO_VARIES          = 506
```

```
HTTP_INSUFFICIENT_STORAGE          = 507
HTTP_NOT_EXTENDED                  = 510
```

As an alternative to *returning* an HTTP error code, handlers can signal an error by *raising* the `apache.SERVER_RETURN` exception, and providing an HTTP error code as the exception value, e.g.

```
raise apache.SERVER_RETURN, apache.HTTP_FORBIDDEN
```

Handlers can send content to the client using the `req.write()` method.

Client data, such as POST requests, can be read by using the `req.read()` function.

Note: The directory of the Apache `Python*Handler` directive in effect is prepended to the `sys.path`. If the directive was specified in a server config file outside any `<Directory>`, then the directory is unknown and not prepended.

An example of a minimalistic handler might be:

```
from mod_python import apache

def requesthandler(req):
    req.content_type = "text/plain"
    req.write("Hello World!")
    return apache.OK
```

4.3 Overview of a Filter Handler

A *filter handler* is a function that can alter the input or the output of the server. There are two kinds of filters - *input* and *output* that apply to input from the client and output to the client respectively.

At this time mod_python supports only request-level filters, meaning that only the body of HTTP request or response can be filtered. Apache provides support for connection-level filters, which will be supported in the future.

A filter handler receives a *filter* object as its argument. The request object is available as well via `filter.req`, but all writing and reading should be done via the filter's object read and write methods.

Filters need to be closed when a read operation returns None (indicating End-Of-Stream).

The return value of a filter is ignored. Filters cannot decline processing like handlers, but the same effect can be achieved by using the `filter.pass_on()` method.

Filters must first be registered using `PythonInputFilter` or `PythonOutputFilter`, then added using the Apache `Add/SetInputFilter` or `Add/SetOutputFilter` directives.

Here is an example of how to specify an output filter, it tells the server that all .py files should processed by CAPITALIZE filter:

```
PythonOutputFilter capitalize CAPITALIZE
AddOutputFilter CAPITALIZE .py
```

And here is what the code for the 'capitalize.py' might look like:

```
from mod_python import apache

def outputfilter(filter):
```

```
    s = filter.read()
    while s:
        filter.write(s.upper())
        s = filter.read()

    if s is None:
        filter.close()
```

When writing filters, keep in mind that a filter will be called any time anything upstream requests an IO operation, and the filter has no control over the amount of data passed through it and no notion of where in the request processing it is called. For example, within a single request, a filter may be called once or five times, and there is no way for the filter to know beforehand that the request is over and which of calls is last or first for this request, thought encounter of an EOS (None returned from a read operation) is a fairly strong indication of an end of a request.

Also note that filters may end up being called recursively in subrequests. To avoid the data being altered more than once, always make sure you are not in a subrequest by examining the `req.main` value.

For more information on filters, see *http://httpd.apache.org/docs-2.0/developer/filters.html*.

4.4 Overview of a Connection Handler

A *connection handler* handles the connection, starting almost immediately from the point the TCP connection to the server was made.

Unlike HTTP handlers, connection handlers receive a *connection* object as an argument.

Connection handlers can be used to implement protocols. Here is an example of a simple echo server:

Apache configuration:

```
PythonConnectionHandler echo
```

Contents of echo.py file:

```
from mod_python import apache

def connectionhandler(conn):

    while 1:
        conn.write(conn.readline())

    return apache.OK
```

4.5 apache – Access to Apache Internals.

The Python interface to Apache internals is contained in a module appropriately named `apache`, located inside the `mod_python` package. This module provides some important objects that map to Apache internal structures, as well as some useful functions, all documented below. (The request object also provides an interface to Apache internals, it is covered in its own section of this manual.)

The `apache` module can only be imported by a script running under mod_python. This is because it depends on a built-in module `_apache` provided by mod_python.

It is best imported like this:

```
from mod_python import apache
```

`mod_python.apache` module defines the following functions and objects. For a more in-depth look at Apache internals, see the *Apache Developer page*

4.5.1 Functions

`log_error`(*message*[, *level, server*])

An interface to the Apache `ap_log_error()` function. *message* is a string with the error message, *level* is one of the following flags constants:

```
APLOG_EMERG
APLOG_ALERT
APLOG_CRIT
APLOG_ERR
APLOG_WARNING
APLOG_NOTICE
APLOG_INFO
APLOG_DEBUG
APLOG_NOERRNO
```

server is a reference to a `req.server` object. If *server* is not specified, then the error will be logged to the default error log, otherwise it will be written to the error log for the appropriate virtual server. When *server* is not specified, the setting of LogLevel does not apply, the LogLevel is dictated by an httpd compile-time default, usually `warn`.

If you have a reference to a request object available, consider using `req.log_error` instead, it will prepend request-specific information such as the source IP of the request to the log entry.

`import_module`(*module_name*[, *autoreload=1, log=0, path=None*])

This function can be used to import modules taking advantage of mod_python's internal mechanism which reloads modules automatically if they have changed since last import.

module_name is a string containing the module name (it can contain dots, e.g. `mypackage.mymodule`); *autoreload* indicates whether the module should be reloaded if it has changed since last import; when *log* is true, a message will be written to the logs when a module is reloaded; *path* allows restricting modules to specific paths.

Example:

```
from mod_python import apache
mymodule = apache.import_module('mymodule', log=1)
```

`allow_methods`([**args*])

A convenience function to set values in `req.allowed`. `req.allowed` is a bitmask that is used to construct the '`Allow:`' header. It should be set before returning a `HTTP_NOT_IMPLEMENTED` error.

Arguments can be one or more of the following:

```
M_GET
M_PUT
M_POST
M_DELETE
M_CONNECT
```

```
M_OPTIONS
M_TRACE
M_PATCH
M_PROPFIND
M_PROPPATCH
M_MKCOL
M_COPY
M_MOVE
M_LOCK
M_UNLOCK
M_VERSION_CONTROL
M_CHECKOUT
M_UNCHECKOUT
M_CHECKIN
M_UPDATE
M_LABEL
M_REPORT
M_MKWORKSPACE
M_MKACTIVITY
M_BASELINE_CONTROL
M_MERGE
M_INVALID
```

config_tree()
: Returns the server-level configuration tree. This tree does not include directives from .htaccess files. This is a *copy* of the tree, modifying it has no effect on the actual configuration.

server_root()
: Returns the value of ServerRoot.

make_table()
: This function is obsolete and is an alias to `table` (see below).

mpm_query(*code*)
: Allows querying of the MPM for various parameters such as numbers of processes and threads. The return value is one of three constants:

```
AP_MPMQ_NOT_SUPPORTED      = 0  # This value specifies whether
                                # an MPM is capable of
                                # threading or forking.
AP_MPMQ_STATIC             = 1  # This value specifies whether
                                # an MPM is using a static # of
                                # threads or daemons.
AP_MPMQ_DYNAMIC            = 2  # This value specifies whether
                                # an MPM is using a dynamic # of
                                # threads or daemons.
```

The *code* argument must be one of the following:

```
AP_MPMQ_MAX_DAEMON_USED    = 1  # Max # of daemons used so far
AP_MPMQ_IS_THREADED        = 2  # MPM can do threading
AP_MPMQ_IS_FORKED          = 3  # MPM can do forking
AP_MPMQ_HARD_LIMIT_DAEMONS = 4  # The compiled max # daemons
AP_MPMQ_HARD_LIMIT_THREADS = 5  # The compiled max # threads
AP_MPMQ_MAX_THREADS        = 6  # # of threads/child by config
AP_MPMQ_MIN_SPARE_DAEMONS  = 7  # Min # of spare daemons
AP_MPMQ_MIN_SPARE_THREADS  = 8  # Min # of spare threads
AP_MPMQ_MAX_SPARE_DAEMONS  = 9  # Max # of spare daemons
```

```
AP_MPMQ_MAX_SPARE_THREADS  = 10 # Max # of spare threads
AP_MPMQ_MAX_REQUESTS_DAEMON= 11 # Max # of requests per daemon
AP_MPMQ_MAX_DAEMONS        = 12 # Max # of daemons by config
```

Example:

```
if apache.mpm_query(apache.AP_MPMQ_IS_THREADED):
    # do something
else:
    # do something else
```

4.5.2 Table Object (mp_table)

class table([*mapping-or-sequence*])

Returns a new empty object of type `mp_table`. See Section 4.5.2 for description of the table object. The *mapping-or-sequence* will be used to provide initial values for the table.

The table object is a wrapper around the Apache APR table. The table object behaves very much like a dictionary (including the Python 2.2 features such as support of the `in` operator, etc.), with the following differences:

- Both keys and values must be strings.
- Key lookups are case-insensitive.
- Duplicate keys are allowed (see `add()` below). When there is more than one value for a key, a subscript operation returns a list.

Much of the information that Apache uses is stored in tables. For example, `req.headers_in` and `req.headers_out`.

All the tables that mod_python provides inside the request object are actual mappings to the Apache structures, so changing the Python table also changes the underlying Apache table.

In addition to normal dictionary-like behavior, the table object also has the following method:

add(*key, val*)

`add()` allows for creating duplicate keys, which is useful when multiple headers, such as `Set-Cookie:` are required.

New in version 3.0.

4.5.3 Request Object

The request object is a Python mapping to the Apache `request_rec` structure. When a handler is invoked, it is always passed a single argument - the request object.

You can dynamically assign attributes to it as a way to communicate between handlers.

Request Methods

add_common_vars()

Calls the Apache `ap_add_common_vars()` function. After a call to this method, `req.subprocess_env` will contain a lot of CGI information.

add_handler(*htype, handler*[, *dir*])

Allows dynamic handler registration. *htype* is a string containing the name of any of the apache request (but not filter or connection) handler directives, e.g. '`PythonHandler`'. *handler* is a string containing the name of the module and the handler function. Optional *dir* is a string containing the name of the directory to be added to the pythonpath. If no directory is specified, then, if there is already a handler of the same type specified, its directory is inherited, otherwise the directory of the presently executing handler is used. If there is a `PythonPath` directive in effect, then `sys.path` will be set exactly according to it (no directories added, the *dir* argument is ignored).

A handler added this way only persists throughout the life of the request. It is possible to register more handlers while inside the handler of the same type. One has to be careful as to not to create an infinite loop this way.

Dynamic handler registration is a useful technique that allows the code to dynamically decide what will happen next. A typical example might be a `PythonAuthenHandler` that will assign different `PythonHandlers` based on the authorization level, something like:

```
if manager:
    req.add_handler("PythonHandler", "menu::admin")
else:
    req.add_handler("PythonHandler", "menu::basic")
```

Note: There is no checking being done on the validity of the handler name. If you pass this function an invalid handler it will simply be ignored.

allow_methods(*methods*[, *reset*])

Adds methods to the `req.allowed_methods` list. This list will be passed in `Allowed:` header if `HTTP_METHOD_NOT_ALLOWED` or `HTTP_NOT_IMPLEMENTED` is returned to the client. Note that Apache doesn't do anything to restrict the methods, this list is only used to construct the header. The actual method-restricting logic has to be provided in the handler code.

methods is a sequence of strings. If *reset* is 1, then the list of methods is first cleared.

document_root()

Returns DocumentRoot setting.

get_basic_auth_pw()

Returns a string containing the password when Basic authentication is used.

get_config()

Returns a reference to the table object containing the mod_python configuration in effect for this request except for `Python*Handler` and `PythonOption` (The latter can be obtained via `req.get_options()`. The table has directives as keys, and their values, if any, as values.

get_remote_host([*type, str_is_ip*])

This method is used to determine remote client's DNS name or IP number. The first call to this function may entail a DNS look up, but subsequent calls will use the cached result from the first call.

The optional *type* argument can specify the following:

- `apache.REMOTE_HOST` Look up the DNS name. Return None if Apache directive `HostNameLookups` is `off` or the hostname cannot be determined.
- `apache.REMOTE_NAME` *(Default)* Return the DNS name if possible, or the IP (as a string in dotted decimal notation) otherwise.
- `apache.REMOTE_NOLOOKUP` Don't perform a DNS lookup, return an IP. Note: if a lookup was performed prior to this call, then the cached host name is returned.
- `apache.REMOTE_DOUBLE_REV` Force a double-reverse lookup. On failure, return None.

If *str_is_ip* is `None` or unspecified, then the return value is a string representing the DNS name or IP address.

If the optional *str_is_ip* argument is not `None`, then the return value is an `(address, str_is_ip)` tuple, where *str_is_ip* is non-zero if `address` is an IP address string.

On failure, `None` is returned.

`get_options``()`
Returns a reference to the table object containing the options set by the `PythonOption` directives.

`internal_redirect``(`*new_uri*`)`
Internally redirects the request to the *new_uri*. *new_uri* must be a string.

The httpd server handles internal redirection by creating a new request object and processing all request phases. Within an internal redirect, `req.prev` will contain a reference to a request object from which it was redirected.

`log_error``(`*message*`[`, *level*`])`
An interface to the Apache `ap_log_rerror` function. *message* is a string with the error message, *level* is one of the following flags constants:

```
APLOG_EMERG
APLOG_ALERT
APLOG_CRIT
APLOG_ERR
APLOG_WARNING
APLOG_NOTICE
APLOG_INFO
APLOG_DEBUG
APLOG_NOERRNO
```

If you need to write to log and do not have a reference to a request object, use the `apache.log_error` function.

`requires``()`
Returns a tuple of strings of arguments to `require` directive.

For example, with the following apache configuration:

```
AuthType Basic
require user joe
require valid-user
```

`requires()` would return `('user joe', 'valid-user')`.

`read``([`*len*`])`
Reads at most *len* bytes directly from the client, returning a string with the data read. If the *len* argument is negative or omitted, reads all data given by the client.

This function is affected by the `Timeout` Apache configuration directive. The read will be aborted and an `IOError` raised if the `Timeout` is reached while reading client data.

This function relies on the client providing the `Content-length` header. Absence of the `Content-length` header will be treated as if `Content-length: 0` was supplied.

Incorrect `Content-length` may cause the function to try to read more data than available, which will make the function block until a `Timeout` is reached.

`readline``([`*len*`])`
Like `read()` but reads until end of line.

Note: In accordance with the HTTP specification, most clients will be terminating lines with '`\r\n`' rather than simply '`\n`'.

readlines([*sizehint*])
Reads all or up to *sizehint* bytes of lines using `readline` and returns a list of the lines read.

register_cleanup(*callable*[, *data*])
Registers a cleanup. Argument *callable* can be any callable object, the optional argument *data* can be any object (default is `None`). At the very end of the request, just before the actual request record is destroyed by Apache, *callable* will be called with one argument, *data*.

It is OK to pass the request object as data, but keep in mind that when the cleanup is executed, the request processing is already complete, so doing things like writing to the client is completely pointless.

If errors are encountered during cleanup processing, they should be in error log, but otherwise will not affect request processing in any way, which makes cleanup bugs sometimes hard to spot.

If the server is shut down before the cleanup had a chance to run, it's possible that it will not be executed.

sendfile(*path*[, *offset, len*])
Sends *len* bytes of file *path* directly to the client, starting at offset *offset* using the server's internal API. *offset* defaults to 0, and *len* defaults to -1 (send the entire file).

This function provides the most efficient way to send a file to the client.

write(*string*[, *flush=1*])
Writes *string* directly to the client, then flushes the buffer, unless flush is 0.

flush()
Flushes the output buffer.

set_content_length(*len*)
Sets the value of `req.clength` and the '`Content-Length`' header to len. Note that after the headers have been sent out (which happens just before the first byte of the body is written, i.e. first call to `req.write()`), calling the method is meaningless.

Request Members

connection
A `connection` object associated with this request. See Connection Object below for details. *(Read-Only)*

server
A server object associate with this request. See Server Object below for details. *(Read-Only*)

next
If this is an internal redirect, the request object we redirect to. *(Read-Only*)

prev
If this is an internal redirect, the request object we redirect from. *(Read-Only*)

main
If this is a sub-request, pointer to the main request. *(Read-Only*)

the_request
String containing the first line of the request. *(Read-Only*)

assbackwards
Indicates an HTTP/0.9 "simple" request. This means that the response will contain no headers, only the body. Although this exists for backwards compatibility with obsolescent browsers, some people have figred out that setting assbackwards to 1 can be a useful technique when including part of the response from an internal redirect to avoid headers being sent.

proxyreq
A proxy request: one of `apache.PROXYREQ_*` values. *(Read-Only*)

header_only
A boolean value indicating HEAD request, as opposed to GET. *(Read-Only)*

protocol
Protocol, as given by the client, or '`HTTP/0.9`'. Same as CGI SERVER_PROTOCOL. *(Read-Only)*

proto_num
Integer. Number version of protocol; 1.1 = 1001 *(Read-Only)*

hostname
String. Host, as set by full URI or Host: header. *(Read-Only)*

request_time
A long integer. When request started. *(Read-Only)*

status_line
Status line. E.g. '`200 OK`'. *(Read-Only)*

status
Status. One of `apache.HTTP_*` values.

method
A string containing the method - 'GET', 'HEAD', 'POST', etc. Same as CGI REQUEST_METHOD. *(Read-Only)*

method_number
Integer containing the method number. *(Read-Only)*

allowed
Integer. A bitvector of the allowed methods. Used to construct the Allowed: header when responding with `HTTP_METHOD_NOT_ALLOWED` or `HTTP_NOT_IMPLEMENTED`. This field is for Apache's internal use, to set the Allowed: methods use `req.allow_methods()` method, described in section 4.5.3. *(Read-Only)*

allowed_xmethods
Tuple. Allowed extension methods. *(Read-Only)*

allowed_methods
Tuple. List of allowed methods. Used in relation with `METHOD_NOT_ALLOWED`. This member can be modified via `req.allow_methods()` described in section 4.5.3. *(Read-Only)*

sent_bodyct
Integer. Byte count in stream is for body. (?) *(Read-Only)*

bytes_sent
Long integer. Number of bytes sent. *(Read-Only)*

mtime
Long integer. Time the resource was last modified. *(Read-Only)*

chunked
Boolean value indicating when sending chunked transfer-coding. *(Read-Only)*

range
String. The `Range:` header. *(Read-Only)*

clength
Long integer. The "real" content length. *(Read-Only)*

remaining
Long integer. Bytes left to read. (Only makes sense inside a read operation.) *(Read-Only)*

read_length
Long integer. Number of bytes read. *(Read-Only)*

read_body
Integer. How the request body should be read. *(Read-Only*)

read_chunked
Boolean. Read chunked transfer coding. *(Read-Only*)

expecting_100
Boolean. Is client waiting for a 100 (`HTTP_CONTINUE`) response. *(Read-Only*)

headers_in
A table object containing headers sent by the client.

headers_out
A `table` object representing the headers to be sent to the client.

err_headers_out
These headers get send with the error response, instead of headers_out.

subprocess_env
A `table` object containing environment information typically usable for CGI. You may have to call `req.add_common_vars()` first to fill in the information you need.

notes
A `table` object that could be used to store miscellaneous general purpose info that lives for as long as the request lives. If you need to pass data between handlers, it's better to simply add members to the request object than to use `notes`.

phase
The phase currently being being processed, e.g. '`PythonHandler`'. *(Read-Only)*

interpreter
The name of the subinterpreter under which we're running. *(Read-Only)*

content_type
String. The content type. Mod_python maintains an internal flag (`req._content_type_set`) to keep track of whether `content_type` was set manually from within Python. The publisher handler uses this flag in the following way: when `content_type` isn't explicitly set, it attempts to guess the content type by examining the first few bytes of the output.

handler
The name of the handler currently being processed. This is the handler set by mod_mime, not the mod_python handler. In most cases it will be "'`mod_python`'. *(Read-Only*)

content_encoding
String. Content encoding. *(Read-Only*)

vlist_validator
Integer. Variant list validator (if negotiated). *(Read-Only*)

user
If an authentication check is made, this will hold the user name. Same as CGI REMOTE_USER. *(Read-Only*)

Note: `req.get_basic_auth_pw()` must be called prior to using this value.

ap_auth_type
Authentication type. Same as CGI AUTH_TYPE. *(Read-Only*)

no_cache
Boolean. No cache if true. *(Read-Only*)

no_local_copy
Boolean. No local copy exists. *(Read-Only*)

unparsed_uri

The URI without any parsing performed. *(Read-Only*)

uri
The path portion of the URI. *(Read-Only*)

filename
String. File name being requested.

canonical_filename
String. The true filename (`req.filename` is canonicalized if they don't match). *(Read-Only)*

path_info
String. What follows after the file name, but is before query args, if anything. Same as CGI PATH_INFO. *(Read-Only*)

args
String. Same as CGI QUERY_ARGS. *(Read-Only*)

finfo
Tuple. A file information structure, analogous to POSIX stat, describing the file pointed to by the URI. `(mode, ino, dev, nlink, uid, gid, size, atime, mtime, ctime, fname, name)`. The `apache` module defines a set of `FINFO_*` constants that should be used to access elements of this tuple. Example:

```
fname = req.finfo[apache.FINFO_FNAME]
```

(Read-Only)

parsed_uri
Tuple. The URI broken down into pieces. `(scheme, hostinfo, user, password, hostname, port, path, query, fragment)`. The `apache` module defines a set of `URI_*` constants that should be used to access elements of this tuple. Example:

```
fname = req.parsed_uri[apache.URI_PATH]
```

(Read-Only)

used_path_info
Flag to accept or reject path_info on current request. *(Read-Only*)

eos_sent
Boolean. EOS bucket sent. *(Read-Only*)

4.5.4 Connection Object (mp_conn)

The connection object is a Python mapping to the Apache conn_rec structure.

Connection Methods

read([*length*])
Reads at most *length* bytes from the client. The read blocks indefinitely until there is at least one byte to read. If length is -1, keep reading until the socket is closed from the other end (This is known as `EXHAUSTIVE` mode in the http server code).

This method should only be used inside *Connection Handlers*.

Note: The behaviour of this method has changed since version 3.0.3. In 3.0.3 and prior, this method would block until *length* bytes was read.

readline([*length*])
Reads a line from the connection or up to *length* bytes.

This method should only be used inside *Connection Handlers*.

write(*string*)
Writes *string* to the client.

This method should only be used inside *Connection Handlers*.

Connection Members

base_server
A `server` object for the physical vhost that this connection came in through. *(Read-Only)*

local_addr
The (address, port) tuple for the server. *(Read-Only)*

remote_addr
The (address, port) tuple for the client. *(Read-Only)*

remote_ip
String with the IP of the client. Same as CGI REMOTE_ADDR. *(Read-Only)*

remote_host
String. The DNS name of the remote client. None if DNS has not been checked, `""` (empty string) if no name found. Same as CGI REMOTE_HOST. *(Read-Only)*

remote_logname
Remote name if using RFC1413 (ident). Same as CGI REMOTE_IDENT. *(Read-Only)*

aborted
Boolean. True is the connection is aborted. *(Read-Only)*

keepalive
Integer. 1 means the connection will be kept for the next request, 0 means “undecided”, -1 means “fatal error”. *(Read-Only)*

double_reverse
Integer. 1 means double reverse DNS lookup has been performed, 0 means not yet, -1 means yes and it failed. *(Read-Only)*

keepalives
The number of times this connection has been used. (?) *(Read-Only)*

local_ip
String with the IP of the server. *(Read-Only)*

local_host
DNS name of the server. *(Read-Only)*

id
Long. A unique connection id. *(Read-Only)*

notes
A `table` object containing miscellaneous general purpose info that lives for as long as the connection lives.

4.5.5 Filter Object (mp_filter)

A filter object is passed to mod_python input and output filters. It is used to obtain filter information, as well as get and pass information to adjacent filters in the filter stack.

Filter Methods

pass_on()
Passes all data through the filter without any processing.

read([*length*])
Reads at most *len* bytes from the next filter, returning a string with the data read or None if End Of Stream (EOS) has been reached. A filter *must* be closed once the EOS has been encountered.

If the *len* argument is negative or omitted, reads all data currently available.

readline([*length*])
Reads a line from the next filter or up to *length* bytes.

write(*string*)
Writes *string* to the next filter.

flush()
Flushes the output by sending a FLUSH bucket.

close()
Closes the filter and sends an EOS bucket. Any further IO operations on this filter will throw an exception.

disable()
Tells mod_python to ignore the provided handler and just pass the data on. Used internally by mod_python to print traceback from exceptions encountered in filter handlers to avoid an infinite loop.

Filter Members

closed
A boolean value indicating whether a filter is closed. *(Read-Only*)

name
String. The name under which this filter is registered. *(Read-Only*)

req
A reference to the request object. *(Read-Only*)

is_input
Boolean. True if this is an input filter. *(Read-Only*)

handler
String. The name of the Python handler for this filter as specified in the configuration. *(Read-Only*)

4.5.6 Server Object (mp_server)

The request object is a Python mapping to the Apache `request_rec` structure. The server structure describes the server (possibly virtual server) serving the request.

Server Methods

get_config()
Similar to `req.get_config()`, but returns a config pointed to by `server->module_config` Apache config vector.

register_cleanup(*request, callable*[, *data*])
Registers a cleanup. Very similar to `req.register_cleanup()`, except this cleanup will be executed at child termination time. This function requires one extra argument - the request object.

Server Members

defn_name
String. The name of the configuration file where the server definition was found. *(Read-Only*)

defn_line_number
Integer. Line number in the config file where the server definition is found. *(Read-Only*)

server_admin
Value of the `ServerAdmin` directive. *(Read-Only*)

server_hostname
Value of the `ServerName` directive. Same as CGI SERVER_NAME.*(Read-Only*)

port
Integer. TCP/IP port number. Same as CGI SERVER_PORT. *This member appears to be 0 on Apache 2.0, look at req.connection.local_addr instead (Read-Only*)

error_fname
The name of the error log file for this server, if any. *(Read-Only*)

loglevel
Integer. Logging level. *(Read-Only*)

is_virtual
Boolean. True if this is a virtual server. *(Read-Only*)

timeout
Integer. Value of the `Timeout` directive. *(Read-Only*)

keep_alive_timeout
Integer. Keepalive timeout. *(Read-Only*)

keep_alive_max
Maximum number of requests per keepalive. *(Read-Only*)

keep_alive
Use persistent connections? *(Read-Only*)

path
String. Path for `ServerPath` *(Read-Only*)

pathlen
Integer. Path length. *(Read-Only*)

limit_req_line
Integer. Limit on size of the HTTP request line. *(Read-Only*)

limit_req_fieldsize
Integer. Limit on size of any request header field. *(Read-Only*)

limit_req_fields
Integer. Limit on number of request header fields. *(Read-Only*)

4.6 `util` – Miscellaneous Utilities

The `util` module provides a number of utilities handy to a web application developer similar to those in the standard library `cgi` module. The implementations in the `util` module are much more efficient because they call directly into Apache API's as opposed to using CGI which relies on the environment to pass information.

The recommended way of using this module is:

```
from mod_python import util
```

See Also:

Common Gateway Interface RFC Project Page
(http://CGI-Spec.Golux.Com/)
for detailed information on the CGI specification

4.6.1 FieldStorage class

Access to form data is provided via the `FieldStorage` class. This class is similar to the standard library module `cgi` `FieldStorage`.

class `FieldStorage`(*req*[, *keep_blank_values, strict_parsing*])
This class provides uniform access to HTML form data submitted by the client. *req* is an instance of the mod_python request object.

The optional argument *keep_blank_values* is a flag indicating whether blank values in URL encoded form data should be treated as blank strings. The default is false, which means that blank values are ignored as if they were not included.

The optional argument *strict_parsing* is not yet implemented.

During initialization, `FieldStorage` class reads all of the data provided by the client. Since all data provided by the client is consumed at this point, there should be no more than one `FieldStorage` class instantiated per single request, nor should you make any attempts to read client data before or after instantiating a `FieldStorage`.

The data read from the client is then parsed into separate fields and packaged in `Field` objects, one per field. For HTML form inputs of type `file`, a temporary file is created that can later be accessed via the `file` attribute of a `Field` object.

The `FieldStorage` class has a mapping object interface, i.e. it can be treated like a dictionary. When used as a mapping, the keys are form input names, and the returned dictionary value can be:

- An instance of `StringField`, containing the form input value. This is only when there is a single value corresponding to the input name. `StringField` is a subclass of `str` which provides the additional `value` attribute for compatibility with standard library `cgi` module.
- An instances of a `Field` class, if the input is a file upload.
- A list of `StringField` and/or `Field` objects. This is when multiple values exist, such as for a `<select>` HTML form element.

Note: Unlike the standard library `cgi` module `FieldStorage` class, a `Field` object is returned *only* when it is a file upload. In all other cases the return is an instance of `StringField`. This means that you do not need to use the `.value` attribute to access values of fields in most cases.

In addition to standard mapping object methods, `FieldStorage` objects have the following attributes:

`list`
This is a list of `Field` objects, one for each input. Multiple inputs with the same name will have multiple elements in this list.

`FieldStorage` methods:

`getfirst`(*name*[, *default*])
Always returns only one value associated with form field *name*. If no such form field or value exists then the method returns the value specified by the optional parameter *default*. This parameter defaults to `None` if not specified.

getlist(*name*)
This method always returns a list of values associated with form field *name*. The method returns an empty list if no such form field or value exists for *name*. It returns a list consisting of one item if only one such value exists.

4.6.2 Field class

class Field()
This class is used internally by `FieldStorage` and is not meant to be instantiated by the user. Each instance of a `Field` class represents an HTML Form input.

`Field` instances have the following attributes:

name
The input name.

value
The input value. This attribute can be used to read data from a file upload as well, but one has to exercise caution when dealing with large files since when accessed via `value`, the whole file is read into memory.

file
This is a file object. For file uploads it points to a temporary file. For simple values, it is a `StringIO` object, so you can read simple string values via this attribute instead of using the `value` attribute as well.

filename
The name of the file as provided by the client.

type
The content-type for this input as provided by the client.

type_options
This is what follows the actual content type in the `content-type` header provided by the client, if anything. This is a dictionary.

disposition
The value of the first part of the `content-disposition` header.

disposition_options
The second part (if any) of the `content-disposition` header in the form of a dictionary.

See Also:

RFC 1867, "*Form-based File Upload in HTML*"
for a description of form-based file uploads

4.6.3 Other functions

parse_qs(*qs*[, *keep_blank_values, strict_parsing*])
This function is functionally equivalent to the standard library `cgi parse_qs`, except that it is written in C and is much faster.

Parse a query string given as a string argument (data of type application/x-www-form-urlencoded). Data are returned as a dictionary. The dictionary keys are the unique query variable names and the values are lists of values for each name.

The optional argument *keep_blank_values* is a flag indicating whether blank values in URL encoded queries should be treated as blank strings. A true value indicates that blanks should be retained as blank strings. The default false value indicates that blank values are to be ignored and treated as if they were not included.

Note: The *strict_parsing* argument is not yet implemented.

parse_qsl(*qs*[, *keep_blank_values, strict_parsing*])

This function is functionally equivalent to the standard library `cgi parse_qsl`, except that it is written in C and is much faster.

Parse a query string given as a string argument (data of type application/x-www-form-urlencoded). Data are returned as a list of name, value pairs.

The optional argument *keep_blank_values* is a flag indicating whether blank values in URL encoded queries should be treated as blank strings. A true value indicates that blanks should be retained as blank strings. The default false value indicates that blank values are to be ignored and treated as if they were not included.

Note: The *strict_parsing* argument is not yet implemented.

redirect(*req, location*[, *permanent=0, text=None*])

This is a convenience function to redirect the browser to another location. When *permanent* is true, `MOVED_PERMANENTLY` status is sent to the client, otherwise it is `MOVED_TEMPORARILY`. A short text is sent to the browser informing that the document has moved (for those rare browsers that do not support redirection); this text can be overridden by supplying a *text* string.

If this function is called after the headers have already been sent, an `IOError` is raised.

This function raises `apache.SERVER_RETURN` exception to abandon any further processing of the handle. If you do not want this, you can wrap the call to `redirect` in a try/except block catching the `apache.SERVER_RETURN`.

4.7 `Cookie` – HTTP State Management

The `Cookie` module provides convenient ways for creating, parsing, sending and receiving HTTP Cookies, as defined in the specification published by Netscape.

Note: Even though there are official IETF RFC's describing HTTP State Management Mechanism using cookies, the de facto standard supported by most browsers is the original Netscape specification. Furthermore, true compliance with IETF standards is actually incompatible with many popular browsers, even those that claim to be RFC-compliant. Therefore, this module supports the current common practice, and is not fully RFC compliant.

See Also:

Persistent Client State - HTTP Cookies
(http://wp.netscape.com/newsref/std/cookie_spec.html)
for the original Netscape specification.

RFC 2109, "*HTTP State Management Mechanism*"
for the first RFC on Cookies.

RFC 2964, "*Use of HTTP State Management*"
for guidelines on using Cookies.

RFC 2965, "*HTTP State Management Mechanism*"
for the latest IETF standard.

HTTP Cookies: Standards, Privacy, and Politics
(http://arxiv.org/abs/cs.SE/0105018)
by David M. Kristol for an excellent overview of the issues surrounding standardization of Cookies.

4.7.1 Classes

class Cookie(*name, value*[, *attributes*])

This class is used to construct a single cookie named *name* and having *value* as the value. Additionally, any of the attributes defined in the Netscape specification and RFC2109 can by supplied as keyword arguments.

The attributes of the class represent cookie attributes, and their string representations become part of the string representation of the cookie. The `Cookie` class restricts attribute names to only valid values, specifically, only the following attributes are allowed: `name, value, version, path, domain, secure, comment, expires, max_age, commentURL, discard, port, __data__`.

The `__data__` attribute is a general-purpose dictionary that can be used for storing arbitrary values, when necessary (This is useful when subclassing `Cookie`).

The `expires` attribute is a property whose value is checked upon setting to be in format '`Wdy, DD-Mon-YYYY HH:MM:SS GMT`' (as dictated per Netscape cookie specification), or a numeric value representing time in seconds since beginning of epoch (which will be automatically correctly converted to GMT time string). An invalid `expires` value will raise `ValueError`.

When converted to a string, a `Cookie` will be in correct format usable as value in a '`Cookie`' or '`Set-Cookie`' header.

Note: Unlike the Python Standard Library Cookie classes, this class represents a single cookie (referred to as *Morsel* in Python Standard Library).

`parse`(*string*)

This is a class method that can be used to create a `Cookie` instance from a cookie string *string* as passed in a header value. During parsing, attribute names are converted to lower case.

Because this is a class method, it must be called explicitly specifying the class.

This method returns a dictionary of `Cookie` instances, not a single `Cookie` instance.

Here is an example of getting a single `Cookie` instance:

```
mycookies = Cookie.parse("spam=eggs; expires=Sat, 14-Jun-2003 02:42:36 GMT")
spamcookie = mycookies["spam"]
```

Note: Because this method uses a dictionary, it is not possible to have duplicate cookies. If you would like to have more than one value in a single cookie, consider using a `MarshalCookie`.

class `SignedCookie`(*name, value, secret*[, *attributes*])

This is a subclass of `Cookie`. This class creates cookies whose name and value are automatically signed using HMAC (md5) with a provided secret *secret*, which must be a non-empty string.

`parse`(*string, secret*)

This method acts the same way as `Cookie.parse()`, but also verifies that the cookie is correctly signed. If the signature cannot be verified, the object returned will be of class `Cookie`.

Note: Always check the types of objects returned by `SignedCookie.parse()`.If it is an instance of `Cookie` (as opposed to `SignedCookie`), the signature verification has failed:

```
# assume spam is supposed to be a signed cookie
if type(spam) is not Cookie.SignedCookie:
    # do something that indicates cookie isn't signed correctly
```

class `MarshalCookie`(*name, value, secret*[, *attributes*])

This is a subclass of `SignedCookie`. It allows for *value* to be any marshallable objects. Core Python types such as string, integer, list, etc. are all marshallable object. For a complete list see *marshal* module documentation.

When parsing, the signature is checked first, so incorrectly signed cookies will not be unmarshalled.

4.7.2 Functions

`add_cookie`(*req, cookie*[, *value, attributes*])

This is a convenience function for setting a cookie in request headers. *req* is a mod_python `Request` object. If *cookie* is an instance of `Cookie` (or subclass thereof), then the cookie is set, otherwise, *cookie* must be a string,

in which case a `Cookie` is constructed using *cookie* as name, *value* as the value, along with any valid `Cookie` attributes specified as keyword arguments.

This function will also set '`Cache-Control: no-cache="set-cookie"`' header to inform caches that the cookie value should not be cached.

Here is one way to use this function:

```
c = Cookie.Cookie('spam', 'eggs', expires=time.time()+300)
Cookie.add_cookie(req, c)
```

Here is another:

```
Cookie.add_cookie(req, 'spam', 'eggs', expires=time.time()+300)
```

`get_cookies`(*req* [, *Class*, *data*])

This is a convenience function for retrieving cookies from incoming headers. *req* is a mod_python `Request` object. *Class* is a class whose `parse()` method will be used to parse the cookies, it defaults to `Cookie`. *Data* can be any number of keyword arguments which, will be passed to `parse()` (This is useful for `signedCookie` and `MarshalCookie` which require `secret` as an additional argument to `parse`).

4.7.3 Examples

This example sets a simple cookie which expires in 300 seconds:

```
from mod_python import Cookie, apache
import time

def handler(req):

    cookie = Cookie.Cookie('eggs', 'spam')
    cookie.expires = time.time() + 300
    Cookie.add_cookie(req, cookie)

    req.write('This response contains a cookie!\n')
    return apache.OK
```

This example checks for incoming marshal cookie and displays it to the client. If no incoming cookie is present a new marshal cookie is set. This example uses '`secret007`' as the secret for HMAC signature.

```
from mod_python import apache, Cookie

def handler(req):

    cookies = Cookie.get_cookie(req, Cookie.MarshalCookie,
                                secret='secret007')
    if cookies.has_key('spam'):
        spamcookie = cookies['spam']

        req.write('Great, a spam cookie was found: %s\n' \
                                  % str(spamcookie))
        if type(spamcookie) is Cookie.MarshalCookie:
            req.write('Here is what it looks like decoded: %s=%s\n'
```

```
                    % (spamcookie.name, spamcookie.value))
        else:
            req.write('WARNING: The cookie found is not a \
                       MarshalCookie, it may have been tapered with!')

    else:

        # MarshaCookie allows value to be any marshallable object
        value = {'egg_count': 32, 'color': 'white'}
        Cookie.add_cookie(req, Cookie.MarshalCookie('spam', value, \
                          'secret007'))
        req.write('Spam cookie not found, but we just set one!\n')

    return apache.OK
```

4.8 Session – Session Management

The Session module provides objects for maintaining persistent sessions across requests.

The module contains a BaseSession class, which is not meant to be used directly (it provides no means of storing a session), and a DbmSession class, which uses a dbm to store sessions.

The BaseSession class also provides session locking, both across processes and threads. For locking it uses APR global_mutexes (a number of them is pre-created at startup) The mutex number is computed by using modulus of the session id hash(). (Therefore it's possible that different session id's will have the same hash, but the only implication is that those two sessions cannot be locked at the same time resulting in a slight delay.)

4.8.1 Classes

Session(*req*[, *sid, secret, timeout, lock, lockfile*])

This function queries the MPM and based on that returns either a new instance of DbmSession or MemorySession. It takes same arguments as BaseSession.

MemorySession will be used if the MPM is threaded and not forked (such is the case on Windows), or if it threaded, forked, but only one process is allowed (the worker MPM can be configured to run this way). In all other cases DbmSession is used.

class BaseSession(*req*[, *sid, secret, timeout, lock, lockfile*])

This class is meant to be used as a base class for other classes that implement a session storage mechanism. *req* is a required reference to a mod_python request object.

BaseSession is a subclass of dict. Data can be stored and retrieved from the session by using it as a dictionary.

sid is an optional session id; if provided, such a session must already exist, otherwise it is ignored and a new session with a new sid is created. If *sid* is not provided, the object will attempt to look at cookies for session id. If a sid is found in cookies, but it is not previously known or the session has expired, then a new sid is created. Whether a session is "new" can be determined by calling the is_new() method.

Cookies generated by sessions will have a path attribute which is calculated by comparing the server DocumentRoot and the directory in which the PythonHandler directive currently in effect was specified. E.g. if document root is '/a/b/c' and PythonHandler was specified in '/a/b/c/d/e', the path will be set to '/d/e'. You can force a specific path by using ApplicationPath option ('PythonOption ApplicationPath /my/path' in server configuration).

When a *secret* is provided, BaseSession will use SignedCookie when generating cookies thereby making the session id almost impossible to fake. The default is to use plain Cookie (though even if not signed, the session id is generated to be very difficult to guess).

A session will timeout if it has not been accessed for more than *timeout*, which defaults to 30 minutes. An attempt to load an expired session will result in a "new" session.

The *lock* argument (defaults to 1) indicates whether locking should be used. When locking is on, only one session object with a particular session id can be instantiated at a time. *lockfile* is the name of a file to be used for inter-process locks.

A session is in "new" state when the session id was just generated, as opposed to being passed in via cookies or the *sid* argument.

is_new()
Returns 1 if this session is new. A session will also be "new" after an attempt to instantiate an expired or non-existent session. It is important to use this method to test whether an attempt to instantiate a session has succeeded, e.g.:

```
sess = Session(req)
if sess.is_new():
    # redirect to login
    util.redirect(req, 'http://www.mysite.com/login')
```

id()
Returns the session id.

created()
Returns the session creation time in seconds since beginning of epoch.

last_accessed()
Returns last access time in seconds since beginning of epoch.

timeout()
Returns session timeout interval in seconds.

set_timeout(*secs***)**
Set timeout to *secs*.

invalidate()
This method will remove the session from the persistent store and also place a header in outgoing headers to invalidate the session id cookie.

load()
Load the session values from storage.

save()
This method writes session values to storage.

delete()
Remove the session from storage.

init_lock()
This method initializes the session lock. There is no need to ever call this method, it is intended for subclasses that wish to use an alternative locking mechanism.

lock()
Locks this session. If the session is already locked by another thread/process, wait until that lock is released. There is no need to call this method if locking is handled automatically (default).

This method registeres a cleanup which always unlocks the session at the end of the request processing.

unlock()
Unlocks this session. (Same as `lock()` - when locking is handled automatically (default), there is no need to call this method).

cleanup()
This method is for subclasses to implement session storage cleaning mechanism (i.e. deleting expired sessions, etc.). It will be called at random, the chance of it being called is controlled by `CLEANUP_CHANCE`

`Session` module variable (default 1000). This means that cleanups will be ordered at random and there is 1 in 1000 chance of it happening. Subclasses implementing this method should not perform the (potentially time consuming) cleanup operation in this method, but should instead use `req.register_cleanup` to register a cleanup which will be executed after the request has been processed.

class `MemorySession`(*req*, [*, sid, secret, dbmtype, timeout, lock*])

This class provides session storage using a global dictionary. This class provides by far the best performance, but cannot be used in a multi-process configuration, and also consumes memory for every active session.

Note that using this class directly is not cross-platform. For best compatibility across platforms, always use the `Session()` function to create sessions.

class `DbmSession`(*req*, [*, dbm, sid, secret, dbmtype, timeout, lock*])

This class provides session storage using a dbm file. Generally, dbm access is very fast, and most dbm implementations memory-map files for faster access, which makes their performance nearly as fast as direct shared memory access.

dbm is the name of the dbm file (the file must be writable by the httpd process). This file is not deleted when the server process is stopped (a nice side benefit of this is that sessions can survive server restarts). By default the session information is stored in a dbmfile named '`mp_sess.dbm`' and stored in a temporary directory returned by `tempfile.gettempdir()` standard library function. It can be overridden by setting `PythonOption SessionDbm filename`.

The implementation uses Python `anydbm` module, which will default to `dbhash` on most systems. If you need to use a specific dbm implementation (e.g. `gdbm`), you can pass that module as *dbmtype*.

Note that using this class directly is not cross-platform. For best compatibility across platforms, always use the `Session()` function to create sessions.

4.9 `psp` – Python Server Pages

The `psp` module provides a way to convert text documents (including, but not limited to HTML documents) containing Python code embedded in special brackets into pure Python code suitable for execution within a mod_python handler, thereby providing a versatile mechanism for delivering dynamic content in a style similar to ASP, JSP and others.

The parser used by `psp` is written in C (generated using flex) and is therefore very fast.

See 6.2 "PSP Handler" for additional PSP information.

Inside the document, Python *code* needs to be surrounded by '`<%`' and '`%>`'. Python *expressions* are enclosed in '`<%=`' and '`%>`'. A *directive* can be enclosed in '`<%@`' and '`%>`'. A comment (which will never be part of the resulting code) can be enclosed in '`<%--`' and '`--%>`'

Here is a primitive PSP page that demonstrated use of both code and expression embedded in an HTML document:

```
<html>
<%
import time
%>
Hello world, the time is: <%=time.strftime("%Y-%m-%d, %H:%M:%S")%>
</html>
```

Internally, the PSP parser would translate the above page into the following Python code:

```
req.write("""<html>
""")
import time
req.write("""
```

```
Hello world, the time is: """); req.write(str(time.strftime("%Y-%m-%d, %H:%M:%S"))); req.writ
</html>
""")
```

This code, when executed inside a handler would result in a page displaying words '`Hello world, the time is: `' followed by current time.

Python code can be used to output parts of the page conditionally or in loops. Blocks are denoted from within Python code by indentation. The last indentation in Python code (even if it is a comment) will persist through the document until either end of document or more Python code.

Here is an example:

```
<html>
<%
for n in range(3):
    # This indent will persist
%>
<p>This paragraph will be
repeated 3 times.</p>
<%
# This line will cause the block to end
%>
This line will only be shown once.<br>
</html>
```

The above will be internally translated to the following Python code:

```
req.write("""<html>
""")
for n in range(3):
    # This indent will persist
    req.write("""
<p>This paragraph will be
repeated 3 times.</p>
""")
# This line will cause the block to end
req.write("""
This line will only be shown once.<br>
</html>
""")
```

The parser is also smart enough to figure out the indent if the last line of Python ends with '`:`' (colon). Considering this, and that the indent is reset when a newline is encountered inside '`<%%>`', the above page can be written as:

```
<html>
<%
for n in range(3):
%>
<p>This paragraph will be
repeated 3 times.</p>
<%
%>
This line will only be shown once.<br>
</html>
```

However, the above code can be confusing, thus having descriptive comments denoting blocks is highly recommended as a good practice.

The only directive supported at this time is `include`, here is how it can be used:

```
<%@ include file="/file/to/include"%>
```

If the `parse()` function was called with the *dir* argument, then the file can be specified as a relative path, otherwise it has to be absolute.

class PSP(*req*, [, *filename*, *string*, *vars*])

This class represents a PSP object.

req is a request object; *filename* and *string* are optional keyword arguments which indicate the source of the PSP code. Only one of these can be specified. If neither is specified, `req.filename` is used as *filename*.

vars is a dictionary of global variables. Vars passed in the `run()` method will override vars passed in here.

This class is used internally by the PSP handler, but can also be used as a general purpose templating tool.

When a file is used as the source, the code object resulting from the specified file is stored in a memory cache keyed on file name and file modification time. The cache is global to the Python interpreter. Therefore, unless the file modification time changes, the file is parsed and resulting code is compiled only once per interpreter.

The cache is limited to 512 pages, which depending on the size of the pages could potentially occupy a significant amount of memory. If memory is of concern, then you can switch to dbm file caching. Our simple tests showed only 20% slower performance using bsd db. You will need to check which implementation `anydbm` defaults to on your system as some dbm libraries impose a limit on the size of the entry making them unsuitable. Dbm caching can be enabled via `PSPDbmCache` Python option, e.g.:

```
PythonOption PSPDbmCache ''/tmp/pspcache.dbm''
```

Note that the dbm cache file is not deleted when the server restarts.

Unlike with files, the code objects resulting from a string are cached in memory only. There is no option to cache in a dbm file at this time.

run([*vars*])

This method will execute the code (produced at object initialization time by parsing and compiling the PSP source). Optional argument *vars* is a dictionary keyed by strings that will be passed in as global variables.

Additionally, the PSP code will be given global variables `req`, `psp`, `session` and `form`. A session will be created and assigned to `session` variable only if `session` is referenced in the code (the PSP handler examines `co_names` of the code object to make that determination). Remember that a mere mention of `session` will generate cookies and turn on session locking, which may or may not be what you want. Similarly, a mod_python `FieldStorage` object will be instantiated if `form` is referenced in the code.

The object passed in `psp` is an instance of `PSPInstance`.

display_code()

Returns an HTML-formatted string representing a side-by-side listing of the original PSP code and resulting Python code produced by the PSP parser.

Here is an example of how `PSP` can be used as a templating mechanism:

The template file:

```
<html>
  <!-- This is a simple psp template called template.html -->
  <h1>Hello, <%=what%>!</h1>
</html>
```

The handler code:

```
from mod_python import apache, psp
```

```
def handler(req):
    template = psp.PSP(req, filename='template.html')
    template.run({'what':'world'})
    return apache.OK
```

class PSPInstance()

An object of this class is passed as a global variable `psp` to the PSP code. Objects of this class are instantiated internally and the interface to `__init__` is purposely undocumented.

set_error_page(*filename*)

Used to set a psp page to be processed when an exception occurs. If the path is absolute, it will be appended to document root, otherwise the file is assumed to exist in the same directory as the current page. The error page will receive one additional variable, `exception`, which is a 3-tuple returned by `sys.exc_info()`.

apply_data(*object*[, ***kw*])

This method will call the callable object *object*, passing form data as keyword arguments, and return the result.

redirect(*location*[, *permanent=0*])

This method will redirect the browser to location *location*. If *permanent* is true, then `MOVED_PERMANENTLY` will be sent (as opposed to `MOVED_TEMPORARILY`).

Note: Redirection can only happen before any data is sent to the client, therefore the Python code block calling this method must be at the very beginning of the page. Otherwise an `IOError` exception will be raised.

Example:

```
<%

# note that the '<' above is the first byte of the page!
psp.redirect('http://www.modpython.org')
%>
```

Additionally, the `psp` module provides the following low level functions:

parse(*filename*[, *dir*])

This function will open file named *filename*, read and parse its content and return a string of resulting Python code.

If *dir* is specified, then the ultimate filename to be parsed is constructed by concatenating *dir* and *filename*, and the argument to `include` directive can be specified as a relative path. (Note that this is a simple concatenation, no path separator will be inserted if *dir* does not end with one).

parsestring(*string*)

This function will parse contents of *string* and return a string of resulting Python code.

CHAPTER

FIVE

Apache Configuration Directives

5.1 Request Handlers

5.1.1 Python*Handler Directive Syntax

All request handler directives have the following syntax:

`Python*Handler` *`handler [handler ...]  [ | .ext [.ext ...]  ]`*

Where *handler* is a callable object that accepts a single argument - request object, and *.ext* is a file extension.

Multiple handlers can be specified on a single line, in which case they will be called sequentially, from left to right. Same handler directives can be specified multiple times as well, with the same result - all handlers listed will be executed sequentially, from first to last. If any handler in the sequence returns a value other than `apache.OK`, then execution of all subsequent handlers is aborted.

The list of handlers can optionally be followed by a `|` followed by one or more file extensions. This would restrict the execution of the handler to those file extensions only. This feature only works for handlers executed after the trans phase.

A *handler* has the following syntax:

`module[::object]`

Where *module* can be a full module name (package dot notation is accepted), and the optional *object* is the name of an object inside the module.

Object can also contain dots, in which case it will be resolved from left to right. During resolution, if mod_python encounters an object of type `<class>`, it will try instantiating it passing it a single argument, a request object.

If no object is specified, then it will default to the directive of the handler, all lower case, with the word '`python`' removed. E.g. the default object for PythonAuthenHandler would be authenhandler.

Example:

```
PythonAuthzHandler mypackage.mymodule::checkallowed
```

For more information on handlers, see Overview of a Handler.

Side note: The '`::`' was chosen for performance reasons. In order for Python to use objects inside modules, the modules first need to be imported. Having the separator as simply a '`.`', would considerably complicate process of sequentially evaluating every word to determine whether it is a package, module, class etc. Using the (admittedly un-Python-like) '`::`' takes the time consuming work of figuring out where the module part ends and the object inside of it begins away from mod_python resulting in a modest performance gain.

5.1.2 PythonPostReadRequestHandler

Syntax: *Python*Handler Syntax*
Context: server config, virtual host
Override: not None
Module: mod_python.c

This handler is called after the request has been read but before any other phases have been processed. This is useful to make decisions based upon the input header fields.

Note: When this phase of the request is processed, the URI has not yet been translated into a path name, therefore this directive could never be executed by Apache if it could specified within `<Directory>`, `<Location>`, `<File>` directives or in an '.htaccess' file. The only place this directive is allowed is the main configuration file, and the code for it will execute in the main interpreter. And because this phase happens before any identification of the type of content being requested is done (i.e. is this a python program or a gif?), the python routine specified with this handler will be called for *ALL* requests on this server (not just python programs), which is an important consideration if performance is a priority.

The handlers below are documented in order in which phases are processed by Apache.

5.1.3 PythonTransHandler

Syntax: *Python*Handler Syntax*
Context: server config, virtual host
Override: not None
Module: mod_python.c

This handler gives allows for an opportunity to translate the URI into an actual filename, before the server's default rules (Alias directives and the like) are followed.

Note: At the time when this phase of the request is being processed, the URI has not been translated into a path name, therefore this directive will never be executed by Apache if specified within `<Directory>`, `<Location>`, `<File>` directives or in an '.htaccess' file. The only place this can be specified is the main configuration file, and the code for it will execute in the main interpreter.

5.1.4 PythonHeaderParserHandler

Syntax: *Python*Handler Syntax*
Context: server config, virtual host, directory, htaccess
Override: not None
Module: mod_python.c

This handler is called to give the module a chance to look at the request headers and take any appropriate specific actions early in the processing sequence.

5.1.5 PythonInitHandler

Syntax: *Python*Handler Syntax*
Context: server config, virtual host, directory, htaccess
Override: not None
Module: mod_python.c

This handler is the first handler called in the request processing phases that is allowed both inside and outside '.htaccess' and directory.

This handler is actually an alias to two different handlers. When specified in the main config file outside any directory tags, it is an alias to `PostReadRequestHandler`. When specified inside directory (where `PostReadRequestHandler` is not allowed), it aliases to `PythonHeaderParserHandler`.

(This idea was borrowed from mod_perl)

5.1.6 PythonAccessHandler

Syntax: *Python*Handler Syntax*
Context: server config, virtual host, directory, htaccess
Override: not None
Module: mod_python.c

This routine is called to check for any module-specific restrictions placed upon the requested resource.

For example, this can be used to restrict access by IP number. To do so, you would return `HTTP_FORBIDDEN` or some such to indicate that access is not allowed.

5.1.7 PythonAuthenHandler

Syntax: *Python*Handler Syntax*
Context: server config, virtual host, directory, htaccess
Override: not None
Module: mod_python.c

This routine is called to check the authentication information sent with the request (such as looking up the user in a database and verifying that the [encrypted] password sent matches the one in the database).

To obtain the username, use `req.user`. To obtain the password entered by the user, use the `req.get_basic_auth_pw()` function.

A return of `apache.OK` means the authentication succeeded. A return of `apache.HTTP_UNAUTHORIZED` with most browser will bring up the password dialog box again. A return of `apache.HTTP_FORBIDDEN` will usually show the error on the browser and not bring up the password dialog `again.` `HTTP_FORBIDDEN` should be used when authentication succeeded, but the user is not permitted to access a particular URL.

An example authentication handler might look like this:

```
def authenhandler(req):

    pw = req.get_basic_auth_pw()
    user = req.user
    if user == "spam" and pw == "eggs":
        return apache.OK
    else:
        return apache.HTTP_UNAUTHORIZED
```

Note: `req.get_basic_auth_pw()` must be called prior to using the `req.user` value. Apache makes no attempt to decode the authentication information unless `req.get_basic_auth_pw()` is called.

5.1.8 PythonAuthzHandler

Syntax: *Python*Handler Syntax*
Context: server config, virtual host, directory, htaccess
Override: not None
Module: mod_python.c

This handler runs after AuthenHandler and is intended for checking whether a user is allowed to access a particular resource. But more often than not it is done right in the AuthenHandler.

5.1.9 PythonTypeHandler

Syntax: *Python*Handler Syntax*
Context: server config, virtual host, directory, htaccess
Override: not None
Module: mod_python.c

This routine is called to determine and/or set the various document type information bits, like Content-type (via `r->content_type`), language, et cetera.

5.1.10 PythonFixupHandler

Syntax: *Python*Handler Syntax*
Context: server config, virtual host, directory, htaccess
Override: not None
Module: mod_python.c

This routine is called to perform any module-specific fixing of header fields, et cetera. It is invoked just before any content-handler.

5.1.11 PythonHandler

Syntax: *Python*Handler Syntax*
Context: server config, virtual host, directory, htaccess
Override: not None
Module: mod_python.c

This is the main request handler. Many applications will only provide this one handler.

5.1.12 PythonLogHandler

Syntax: *Python*Handler Syntax*
Context: server config, virtual host, directory, htaccess
Override: not None
Module: mod_python.c

This routine is called to perform any module-specific logging activities.

5.1.13 PythonCleanupHandler

Syntax: *Python*Handler Syntax*
Context: server config, virtual host, directory, htaccess
Override: not None
Module: mod_python.c

This is the very last handler, called just before the request object is destroyed by Apache.

Unlike all the other handlers, the return value of this handler is ignored. Any errors will be logged to the error log, but will not be sent to the client, even if PythonDebug is On.

This handler is not a valid argument to the `rec.add_handler()` function. For dynamic clean up registration, use `req.register_cleanup()`.

Once cleanups have started, it is not possible to register more of them. Therefore, `req.register_cleanup()` has no effect within this handler.

Cleanups registered with this directive will execute *after* cleanups registered with `req.register_cleanup()`.

5.2 Filters

5.2.1 PythonInputFilter

Syntax: PythonInputFilter handler name
Context: server config
Module: mod_python.c

Registers an input filter *handler* under name *name*. *Handler* is a module name optionally followed `::` and a callable object name. If callable object name is omitted, it will default to '`inputfilter`'. *Name* is the name under which the filter is registered, by convention filter names are usually in all caps.

To activate the filter, use the `AddInputFilter` directive.

5.2.2 PythonOutputFilter

Syntax: PythonOutputFilter handler name
Context: server config
Module: mod_python.c

Registers an output filter *handler* under name *name*. *Handler* is a module name optionally followed `::` and a callable object name. If callable object name is omitted, it will default to '`outputfilter`'. *Name* is the name under which the filter is registered, by convention filter names are usually in all caps.

To activate the filter, use the `AddOutputFilter` directive.

5.3 Connection Handler

5.3.1 PythonConnectionHandler

Syntax: PythonConnectionHandler handler
Context: server config
Module: mod_python.c

Specifies that the connection should be handled with *handler* connection handler. *Handler* will be passed a single argument - the connection object.

Handler is a module name optionally followed `::` and a callable object name. If callable object name is omitted, it will default to '`connectionhandler`'.

5.4 Other Directives

5.4.1 PythonEnablePdb

Syntax: PythonEnablePdb {On, Off}
Default: PythonEnablePdb Off
Context: server config, virtual host, directory, htaccess
Override: not None
Module: mod_python.c

When On, mod_python will execute the handler functions within the Python debugger pdb using the `pdb.runcall()` function.

Because pdb is an interactive tool, start httpd from the command line with the -DONE_PROCESS option when using this directive. As soon as your handler code is entered, you will see a Pdb prompt allowing you to step through the code and examine variables.

5.4.2 PythonDebug

Syntax: PythonDebug {On, Off}
Default: PythonDebug Off
Context: server config, virtual host, directory, htaccess
Override: not None
Module: mod_python.c

Normally, the traceback output resulting from uncaught Python errors is sent to the error log. With PythonDebug On directive specified, the output will be sent to the client (as well as the log), except when the error is `IOError` while writing, in which case it will go to the error log.

This directive is very useful during the development process. It is recommended that you do not use it production environment as it may reveal to the client unintended, possibly sensitive security information.

5.4.3 PythonImport

Syntax: PythonImport *module interpreter_name*
Context: server config
Module: mod_python.c

Tells the server to import the Python module module at process startup under the specified interpreter name. This is useful for initialization tasks that could be time consuming and should not be done at the request processing time, e.g. initializing a database connection.

The import takes place at child process initialization, so the module will actually be imported once for every child process spawned.

Note: At the time when the import takes place, the configuration is not completely read yet, so all other directives, including PythonInterpreter have no effect on the behavior of modules imported by this directive. Because of this limitation, the interpreter must be specified explicitly, and must match the name under which subsequent requests relying on this operation will execute. If you are not sure under what interpreter name a request is running, examine the `interpreter` member of the request object.

See also Multiple Interpreters.

5.4.4 PythonInterpPerDirectory

Syntax: PythonInterpPerDirectory {On, Off}
Default: PythonInterpPerDirectory Off
Context: server config, virtual host, directory, htaccess
Override: not None
Module: mod_python.c

Instructs mod_python to name subinterpreters using the directory of the file in the request (`req.filename`) rather than the the server name. This means that scripts in different directories will execute in different subinterpreters as opposed to the default policy where scripts in the same virtual server execute in the same subinterpreter, even if they are in different directories.

For example, assume there is a '/directory/subdirectory'. '/directory' has an .htaccess file with a PythonHandler directive. '/directory/subdirectory' doesn't have an .htaccess. By default, scripts in /directory and '/directory/subdirectory' would execute in the same interpreter assuming both directories are accessed via the same virtual server. With PythonInterpPerDirectory, there would be two different interpreters, one for each directory.

Note: In early phases of the request prior to the URI translation (PostReadRequestHandler and TransHandler) the path is not yet known because the URI has not been translated. During those phases and with PythonInterpPerDirectory on, all python code gets executed in the main interpreter. This may not be exactly what you want, but unfortunately there is no way around this.

See Also:

Section 4.1 Multiple Interpreters
(pyapi-interps.html)
for more information

5.4.5 PythonInterpPerDirective

Syntax: PythonInterpPerDirective {On, Off}
Default: PythonInterpPerDirective Off
Context: server config, virtual host, directory, htaccess
Override: not None
Module: mod_python.c

Instructs mod_python to name subinterpreters using the directory in which the Python*Handler directive currently in effect was encountered.

For example, assume there is a '/directory/subdirectory'. '/directory' has an .htaccess file with a PythonHandler directive. '/directory/subdirectory' has another '.htaccess' file with another PythonHandler. By default, scripts in '/directory' and '/directory/subdirectory' would execute in the same interpreter assuming both directories are in the same virtual server. With PythonInterpPerDirective, there would be two different interpreters, one for each directive.

See Also:

Section 4.1 Multiple Interpreters
(pyapi-interps.html)
for more information

5.4.6 PythonInterpreter

Syntax: PythonInterpreter name
Context: server config, virtual host, directory, htaccess
Override: not None
Module: mod_python.c

Forces mod_python to use interpreter named *name*, overriding the default behaviour or behaviour dictated by `PythonInterpPerDirectory` or `PythonInterpPerDirective` directive.

This directive can be used to force execution that would normally occur in different subinterpreters to run in the same one. When specified in the DocumentRoot, it forces the whole server to run in one subinterpreter.

See Also:

Section 4.1 Multiple Interpreters
(pyapi-interps.html)
for more information

5.4.7 PythonHandlerModule

Syntax: PythonHandlerModule module
Context: server config, virtual host, directory, htaccess
Override: not None
Module: mod_python.c

PythonHandlerModule can be used an alternative to Python*Handler directives. The module specified in this handler will be searched for existence of functions matching the default handler function names, and if a function is found, it will be executed.

For example, instead of:

```
PythonAutenHandler mymodule
PythonHandler mymodule
PythonLogHandler mymodule
```

one can simply say

```
PythonHandlerModule mymodule
```

5.4.8 PythonAutoReload

Syntax: PythonAutoReload {On, Off}
Default: PythonAutoReload On
Context: server config, virtual host, directory, htaccess
Override: not None
Module: mod_python.c

If set to Off, instructs mod_python not to check the modification date of the module file.

By default, mod_python checks the time-stamp of the file and reloads the module if the module's file modification date is later than the last import or reload. This way changed modules get automatically reimported, eliminating the need to restart the server for every change.

Disabling autoreload is useful in production environment where the modules do not change; it will save some processing time and give a small performance gain.

5.4.9 PythonOptimize

Syntax: PythonOptimize {On, Off}
Default: PythonOptimize Off

Context: server config
Module: mod_python.c

Enables Python optimization. Same as the Python **-O** option.

5.4.10 PythonOption

Syntax: PythonOption key value
Context: server config, virtual host, directory, htaccess
Override: not None
Module: mod_python.c

Assigns a key value pair to a table that can be later retrieved by the `req.get_options()` function. This is useful to pass information between the apache configuration files ('httpd.conf', '.htaccess', etc) and the Python programs.

5.4.11 PythonPath

Syntax: PythonPath *path*
Context: server config, virtual host, directory, htaccess
Override: not None
Module: mod_python.c

PythonPath directive sets the PythonPath. The path must be specified in Python list notation, e.g.

```
PythonPath "['/usr/local/lib/python2.0', '/usr/local/lib/site_python', '/some/other/place']"
```

The path specified in this directive will replace the path, not add to it. However, because the value of the directive is evaled, to append a directory to the path, one can specify something like

```
PythonPath "sys.path+['/mydir']"
```

Mod_python tries to minimize the number of evals associated with the PythonPath directive because evals are slow and can negatively impact performance, especially when the directive is specified in an '.htaccess' file which gets parsed at every hit. Mod_python will remember the arguments to the PythonPath directive in the un-evaled form, and before evaling the value it will compare it to the remembered value. If the value is the same, no action is taken. Because of this, you should not rely on the directive as a way to restore the pythonpath to some value if your code changes it.

Note: This directive should not be used as a security measure since the Python path is easily manipulated from within the scripts.

CHAPTER

SIX

Standard Handlers

6.1 Publisher Handler

The `publisher` handler is a good way to avoid writing your own handlers and focus on rapid application development. It was inspired by *Zope* ZPublisher.

6.1.1 Introduction

To use the handler, you need the following lines in your configuration

```
<Directory /some/path>
  SetHandler mod_python
  PythonHandler mod_python.publisher
  </Directory>
```

This handler allows access to functions and variables within a module via URL's. For example, if you have the following module, called 'hello.py':

```
""" Publisher example """

def say(req, what="NOTHING"):
return "I am saying %s" % what
```

A URL `http://www.mysite.com/hello.py/say` would return '`I am saying NOTHING`'. A URL `http://www.mysite.com/hello.py/say?what=hello` would return '`I am saying hello`'.

6.1.2 The Publishing Algorithm

The Publisher handler maps a URI directly to a Python variable or callable object, then, respectively, returns it's string representation or calls it returning the string representation of the return value.

Traversal

The Publisher handler locates and imports the module specified in the URI. The module location is determined from the `req.filename` attribute. Before importing, the file extension, if any, is discarded.

If `req.filename` is empty, the module name defaults to '`index`'.

Once module is imported, the remaining part of the URI up to the beginning of any query data (a.k.a. PATH_INFO) is used to find an object within the module. The Publisher handler *traverses* the path, one element at a time from left to right, mapping the elements to Python object within the module.

If no path_info was given in the URL, the Publisher handler will use the default value of '`index`'. If the last element is an object inside a module, and the one immediately preceding it is a directory (i.e. no module name is given), then the module name will also default to '`index`'.

The traversal will stop and `HTTP_NOT_FOUND` will be returned to the client if:

- Any of the traversed object's names begin with an underscore ('_'). Use underscores to protect objects that should not be accessible from the web.
- A module is encountered. Published objects cannot be modules for security reasons.

If an object in the path could not be found, `HTTP_NOT_FOUND` is returned to the client.

For example, given the following configuration:

```
DocumentRoot /some/dir

<Directory /some/dir>
SetHandler mod_python
PythonHandler mod_python.publisher
</Directory>
```

And the following '/some/dir/index.py' file:

```
def index(req):

return "We are in index()"

def hello(req):

return "We are in hello()"
```

Then:

http://www.somehost/index/index will return '`We are in index()`'

http://www.somehost/index/ will return '`We are in index()`'

http://www.somehost/index/hello will return '`We are in hello()`'

http://www.somehost/hello will return '`We are in hello()`'

http://www.somehost/spam will return '`404 Not Found`'

Argument Matching and Invocation

Once the destination object is found, if it is callable and not a class, the Publisher handler will get a list of arguments that the object expects. This list is compared with names of fields from HTML form data submitted by the client via `POST` or `GET`. Values of fields whose names match the names of callable object arguments will be passed as strings. Any fields whose names do not match the names of callable argument objects will be silently dropped, unless the destination callable object has a `**kwargs` style argument, in which case fields with unmatched names will be passed in the `**kwargs` argument.

If the destination is not callable or is a class, then its string representation is returned to the client.

Authentication

The publisher handler provides simple ways to control access to modules and functions.

At every traversal step, the Publisher handler checks for presence of `__auth__` and `__access__` attributes (in this order), as well as `__auth_realm__` attribute.

If `__auth__` is found and it is callable, it will be called with three arguments: the `Request` object, a string containing the user name and a string containing the password. If the return value of `__auth__` is false, then `HTTP_UNAUTHORIZED` is returned to the client (which will usually cause a password dialog box to appear).

If `__auth__` is a dictionary, then the user name will be matched against the key and the password against the value associated with this key. If the key and password do not match, `HTTP_UNAUTHORIZED` is returned. Note that this requires storing passwords as clear text in source code, which is not very secure.

`__auth__` can also be a constant. In this case, if it is false (i.e. `None`, `0`, `""`, etc.), then `HTTP_UNAUTHORIZED` is returned.

If there exists an `__auth_realm__` string, it will be sent to the client as Authorization Realm (this is the text that usually appears at the top of the password dialog box).

If `__access__` is found and it is callable, it will be called with two arguments: the `Request` object and a string containing the user name. If the return value of `__access__` is false, then `HTTP_FORBIDDEN` is returned to the client.

If `__access__` is a list, then the user name will be matched against the list elements. If the user name is not in the list, `HTTP_FORBIDDEN` is returned.

Similarly to `__auth__`, `__access__` can be a constant.

In the example below, only user '`eggs`' with password '`spam`' can access the `hello` function:

```
__auth_realm__ = "Members only"

def __auth__(req, user, passwd):

if user == "eggs" and passwd == "spam" or \
user == "joe" and passwd == "eoj":
return 1
else:
return 0

def __access__(req, user):
if user == "eggs":
return 1
else:
return 0

def hello(req):
return "hello"
```

Here is the same functionality, but using an alternative technique:

```
__auth_realm__ = "Members only"
__auth__ = {"eggs":"spam", "joe":"eoj"}
__access__ = ["eggs"]
```

```
def hello(req):
return "hello"
```

Since functions cannot be assigned attributes, to protect a function, an `__auth__` or `__access__` function can be defined within the function, e.g.:

```
def sensitive(req):

def __auth__(req, user, password):
if user == 'spam' and password == 'eggs':
# let them in
return 1
else:
# no access
return 0

# something involving sensitive information
return 'sensitive information'
```

Note that this technique will also work if `__auth__` or `__access__` is a constant, but will not work is they are a dictionary or a list.

The `__auth__` and `__access__` mechanisms exist independently of the standard *PythonAuthenHandler*. It is possible to use, for example, the handler to authenticate, then the `__access__` list to verify that the authenticated user is allowed to a particular function.

Note: In order for mod_python to access `__auth__`, the module containing it must first be imported. Therefore, any module-level code will get executed during the import even if `__auth__` is false. To truly protect a module from being accessed, use other authentication mechanisms, e.g. the Apache `mod_auth` or with a mod_python *PythonAuthenHandler* handler.

6.1.3 Form Data

In the process of matching arguments, the Publisher handler creates an instance of *FieldStorage* class. A reference to this instance is stored in an attribute `form` of the `Request` object.

Since a `FieldStorage` can only be instantiated once per request, one must not attempt to instantiate `FieldStorage` when using the Publisher handler and should use `Request.form` instead.

6.2 PSP Handler

PSP handler is a handler that processes documents using the `PSP` class in `mod_python.psp` module.

To use it, simply add this to your httpd configuration:

```
AddHandler mod_python .psp
PythonHandler mod_python.psp
```

For more details on the PSP syntax, see Section 4.9.

If `PythonDebug` server configuration is `On`, then by appending an underscore ('_') to the end of the url you can get a nice side-by-side listing of original PSP code and resulting Python code generated by the `psp` module. This is very useful for debugging.

Note: Leaving debug on in a production environment will allow remote users to display source code of your PSP pages!

6.3 CGI Handler

CGI handler is a handler that emulates the CGI environment under mod_python.

Note that this is not a 'true' CGI environment in that it is emulated at the Python level. stdin and stdout are provided by substituting sys.stdin and sys.stdout, and the environment is replaced by a dictionary. The implication is that any outside programs called from within this environment via os.system, etc. will not see the environment available to the Python program, nor will they be able to read/write from standard input/output with the results expected in a 'true' CGI environment.

The handler is provided as a stepping stone for the migration of legacy code away from CGI. It is not recommended that you settle on using this handler as the preferred way to use mod_python for the long term. This is because the CGI environment was not intended for execution within threads (e.g. requires changing of current directory with is inherently not thread-safe, so to overcome this cgihandler maintains a thread lock which forces it to process one request at a time in a multi-threaded server) and therefore can only be implemented in a way that defeats many of the advantages of using mod_python in the first place.

To use it, simply add this to your '.htaccess' file:

```
SetHandler mod_python
PythonHandler mod_python.cgihandler
```

As of version 2.7, the cgihandler will properly reload even indirectly imported module. This is done by saving a list of loaded modules (sys.modules) prior to executing a CGI script, and then comparing it with a list of imported modules after the CGI script is done. Modules (except for whose whose __file__ attribute points to the standard Python library location) will be deleted from sys.modules thereby forcing Python to load them again next time the CGI script imports them.

If you do not want the above behavior, edit the 'cgihandler.py' file and comment out the code delimited by ###.

Tests show the cgihandler leaking some memory when processing a lot of file uploads. It is still not clear what causes this. The way to work around this is to set the Apache MaxRequestsPerChild to a non-zero value.

APPENDIX

A

Changes from Previous Major Version (2.x)

- Mod_python 3.0 no longer works with Apache 1.3, only Apache 2.x is supported.
- Mod_python no longer works with Python versions less than 2.2.1
- Mod_python now supports Apache filters.
- Mod_python now supports Apache connection handlers.
- Request object supports internal_redirect().
- Connection object has read(), readline() and write().
- Server object has get_config().
- Httpdapi handler has been deprecated.
- Zpublisher handler has been deprecated.
- Username is now in req.user instead of req.connection.user

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z