mod_python documentation

Release 3.4.1-3.4.1

Gregory Trubetskoy

October 22, 2013

CONTENTS

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 for information on the Python language

Apache HTTP Server Project Web Site for information on the Apache server

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 Apache HTTP Server API

Apache processes requests in *phases* (i.e. read the request, parse headers, check access, etc.). Phases are 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 Developer Documentation, as well as the Mod_python - Integrating Python with Apache paper.

Currently only a subset of the Apache HTTP Server API is accessible via mod_python. It was never the goal of the project to provide a 100% coverage of the API. Rather, mod_python is focused on the most useful parts of the API and on providing the most "Pythonic" ways of using it.

1.3 Other Features

Mod_python also provides a number features that fall in the category of web development, e.g. a parser for embedding Python in HTML (*psp – Python Server Pager*), a handler that maps URL space into modules and functions (*Publisher Handler*), support for session (*Session – Session Management*) and cookie (*Cookie – HTTP State Management*) handling.

See Also:

Apache HTTP Server Developer Documentation for HTTP developer information

Mod_python - Integrating Python with Apache for details on how mod_python interfaces with Apache HTTP Server

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 or visit the mod_python mailing list page

2.1 Prerequisites

In the ideal case your Operating System provides a pre-packaged version of mod_python. If not, you will need to compile it yourself. This version of mod_python requires:

- Python 2.6 or 2.7. (earlier versions might work too).
- Apache 2.2 or later. Apache 2.4 is highly recommended over 2.2.

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 pre-packaged software then you may need to install the "development" packages which contain the include files and libraries necessary to compile mod_python. Please check your OS documentation for specifics. (Hint: look for packages named python-devel or python-dev and apache-devel or apache-dev or httpd-dev, etc.).

2.2 Compiling

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 required for compilation.

  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 first 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` option, e.g.:

```
$ ./configure --with-python=/usr/local/bin/python2.3
```

- Sets the directory for the apache mutex locks (if the mutex mechanism chosen by APR requires one).

  Note: mutex locks are used only by *mod_python Sessions* and *PSP* (which maintains a Session implicitly). If you're not using mod_python Sessions or PSP, then this setting should not matter.

  Default is `/tmp`. The directory must exist and be writable by the owner of the apache process.

  Use *with-mutex-dir* option, e.g:

  ```
  $ ./configure --with-mutex-dir=/var/run/mod_python
  ```

  The mutex directory can also be specified at run time using *PythonOption* `mod_python.mutex_directory`. See *Configuring Apache*.

  New in version 3.3.0

- Sets the maximum number of mutex locks reserved by mod_python.

  Note: mutex locks are used only by *mod_python Sessions* and *PSP* (which maintains a Session implicitly). If you're not using mod_python Sessions or PSP, then this setting should not matter.

  The mutexes used for locking are a limited resource on some systems. Increasing the maximum number of locks may increase performance when using session locking. The default is 8. A reasonable number for higher performance would be 32. Use *with-max-locks* option, e.g:

  ```
  $ ./configure --with-max-locks=32
  ```

  The number of locks can also be specified at run time using *PythonOption* `mod_python.mutex_locks`. See *Configuring Apache*.

  New in version 3.2.0

- Attempts to locate **flex** and determine its version. If **flex** cannot be found in your `PATH` **configure** will fail. If the wrong version is found **configure** will generate a warning. You can generally ignore this warning unless you need to re-create `src/psp_parser.c`.

  The parser used by psp (See *psp – Python Server Pager*) is written in C generated using **flex**. (This requires a reentrant version of **flex**, 2.5.31 or later).

  If the first flex binary in the path is not suitable or not the one desired you can specify an alternative location with the option:with-flex: option, e.g:

  ```
  $ ./configure --with-flex=/usr/local/bin/flex
  ```

  New in version 3.2.0

2.2.2 Running `make`

- To start the build process, simply run:

  ```
  $ make
  ```

2.3 Installing

Running `make install`

- This part of the installation in most cases needs to be done as root:

  ```
  $ sudo make install
  ```

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

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

2.4 Configuring Apache

- *LoadModule*

  You need to configure 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.

- See *Testing* below for more basic configuration parameters.

2.5 Testing

1. Make a directory that would be visible on your web site, e.g. `htdocs/test`.
2. Add the following configuration directives to the main server config file:

   ```
   <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 Server-Root)

   This configuration can also be specified in an `.htaccess` file. Note that `.htaccess` configuration is typically disabled by default, to enable it in a directory specify `AllowOverride` with at least `FileInfo`.
3. This causes all requests for URLs ending in `.py` to be processed by mod_python. Upon being handed a request, mod_python looks for the appropriate *python handler* to handle it. Here, there is a single `PythonHandler` directive defining module `mptest` as the python handler to use. We'll see next how this python handler is defined.
4. 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.
5. 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.content_type = 'text/plain'
    req.write("Hello World!")
    return apache.OK
```

6. 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.

7. Note that according to the configuration written above, you can point your browser to *any* URL ending in .py in the test directory. Therefore pointing your browser to `/test/foobar.py` will be handled exactly the same way by `mptest.py`. This is because the code in the `handler` function does not bother examining the URL and always acts the same way no matter what the URL is.

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

2.6 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.

- Beginning with mod_python 3.2.0, you can use the mod_python.testhandler to diagnose your configuration. Add this to your `httpd.conf` file:

  ```
  <Location /mpinfo>
    SetHandler mod_python
    PythonHandler mod_python.testhandler
  </Location>
  ```

  Now point your browser to the `/mpinfo` URL (e.g. `http://localhost/mpinfo`) and note down the information given. This will help you reporting your problem to the mod_python list.

- Ask on the mod_python list. Please 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) the section *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 demonstrates a simple feedback form. The form asks for a name, e-mail address and a comment which are then used to construct and send a message to the webmaster. 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>
```

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`.

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 *Publisher Handler* for more information on the publisher handler.

3.2 Quick Overview of how Apache Handles Requests

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 provide this one handler. To see all the possible handlers, refer to Section *Apache Configuration 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>
```

Note: `/mywebdir` is an absolute physical path in this case.

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

```python
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:

- If not already done, prepend the directory in which the `PythonHandler` directive was found to `sys.path`.
- 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'`.)
- Look for a function called `handler` in module `myscript`.
- Call the function, passing it a request object. (More on what a request object is later).
- At this point we're inside the script, let's examine it line-by-line:
  - `from mod_python import apache`

    This imports the apache module which provides 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 because our handler does not produce any html, `'text/plain'` is more appropriate. You should always make sure this is set *before* any call to `'req.write'`. When you first call `'req.write'`, the response HTTP header is sent to the client and all subsequent changes to the content type (or other HTTP headers) have no effect.
  - `req.write("Hello World!")`

    This writes the `'Hello World!'` string to the client.
  - `return apache.OK`

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

Note: It is important to understand that in order for the handler code to be executed, the URL needs not refer specficially to `myscript.py`. The only requirement is that it refers to a `.py` file. This is because the `AddHandler mod_python .py` directive assignes mod_python to be a handler for a file *type* (based on extention `.py`), not a specific file. Therefore the name in the URL does not matter, in fact the file referred to in the URL doesn't event have to exist. Given the above configuration, 'http://myserver/mywebdir/myscript.py' and 'http://myserver/mywebdir/montypython.py' would yield the exact same result.

3.4 Now something More Complicated - Authentication

Now that you know how to write a basic 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>
```

Note that depending on which version of Apache is being used, you may need to set either the code{AuthAuthoritative} or AuthBasicAuthoritative directive to Off to tell Apache that you want allow the task of performing basic authentication to fall through to your handler.

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.

3.5 Your Own 404 Handler

In some cases, you may wish to return a 404 (`HTTP_NOT_FOUND`) or other non-200 result from your handler. There is a trick here. if you return `HTTP_NOT_FOUND` from your handler, Apache will handle rendering an error page. This can be problematic if you wish your handler to render it's own error page.

In this case, you need to set `req.status = apache.HTTP_NOT_FOUND`, render your page, and then `return(apache.OK)`:

```
from mod_python import apache

def handler(req):
   if req.filename[-17:] == 'apache-error.html':
      #  make Apache report an error and render the error page
      return(apache.HTTP_NOT_FOUND)
   if req.filename[-18:] == 'handler-error.html':
      #  use our own error page
      req.status = apache.HTTP_NOT_FOUND
      pagebuffer = 'Page not here.  Page left, not know where gone.'
   else:
      #  use the contents of a file
      pagebuffer = open(req.filename, 'r').read()

   #  fall through from the latter two above
   req.write(pagebuffer)
   return(apache.OK)
```

Note that if wishing to returning an error page from a handler phase other than the response handler, the value `apache.DONE` must be returned instead of `apache.OK`. If this is not done, subsequent handler phases will still be run. The value of `apache.DONE` indicates that processing of the request should be stopped immediately. If using stacked response handlers, then `apache.DONE` should also be returned in that situation to prevent subsequent handlers registered for that phase being run if appropriate.

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. (In fact, 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 the *main interpeter*. 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 behavior 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 ends.

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

Note: If any module is being used which has a C code component that uses the simplified API for access to the Global Interpreter Lock (GIL) for Python extension modules, then the interpreter name must be forcibly set to be `'main_interpreter'`. This is necessary as such a module will only work correctly if run within the context of the first Python interpreter created by the process. If not forced to run under the `'main_interpreter'`, a range of Python errors can arise, each typically referring to code being run in *restricted mode*.

See Also:

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

http://www.python.org/peps/pep-0311.html PEP 0311 - Simplified Global Interpreter Lock Acquisition for Extensions

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 `request.write()` method.

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

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 `request.main` value.

For more information on filters, see http://httpd.apache.org/docs-2.4/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

apache.**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 // DEPRECATED
```

server is a reference to a `request.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 `request.log_error()` instead, it will prepend request-specific information such as the source IP of the request to the log entry.

apache.**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:

```python
from mod_python import apache
module = apache.import_module('module_name', log=1)
```

apache.**allow_methods**([**args*])

A convenience function to set values in `request.allowed()`. `request.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
```

`apache.`**`exists_config`**`(`*name*`)`

This function returns True if the Apache server was launched with the definition with the given *name*. This means that you can test whether Apache was launched with the `-DFOOBAR` parameter by calling `apache.exists_config_define('FOOBAR')`.

`apache.`**`stat`**`(`*fname*, *wanted*`)`

This function returns an instance of an `mp_finfo` object describing information related to the file with name `fname`. The `wanted` argument describes the minimum attributes which should be filled out. The resultant object can be assigned to the `request.finfo` attribute.

`apache.`**`register_cleanup`**`(`*callable*[, *data*]`)`

Registers a cleanup that will be performed at child shutdown time. Equivalent to `server.register_cleanup()`, except that a request object is not required. *Warning:* do not pass directly or indirectly a request object in the data parameter. Since the callable will be called at server shutdown time, the request object won't exist anymore and any manipulation of it in the handler will give undefined behaviour.

`apache.`**`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.

`apache.`**`server_root`**`()`

Returns the value of ServerRoot.

`apache.`**`make_table`**`()`

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

`apache.`**`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 Attributes

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

apache.**main_server**
A `server` object for the main server. *(Read-Only)*

apache.**MODULE_MAGIC_NUMBER_MAJOR**
Integer. An internal to Apache version number useful to determine whether certain features should be available. See `MODULE_MAGIC_NUMBER_MINOR`.

Major API changes that could cause compatibility problems for older modules such as structure size changes. No binary compatibility is possible across a change in the major version.

(Read-Only)

apache.**MODULE_MAGIC_NUMBER_MINOR**
Integer. An internal to Apache version number useful to determine whether certain features should be available. See `MODULE_MAGIC_NUMBER_MAJOR`.

Minor API changes that do not cause binary compatibility problems.

(Read-Only)

4.5.3 Table Object (mp_table)

class apache.**table**([*mapping-or-sequence*])
Returns a new empty object of type `mp_table`. See Section *Table Object (mp_table)* 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 `table.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, `request.headers_in()` and `request.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*)
Allows for creating duplicate keys, which is useful when multiple headers, such as *Set-Cookie:* are required.

4.5.4 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. For brevity, we often refer to it here and throughout the code as `req`.

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

Request Methods

`request.`**add_cgi_vars**()
Calls Apache function `ap_add_common_vars()` followed some code very similar to Apache `ap_add_cgi_vars()` with the exception of calculating `PATH_TRANSLATED` value, thereby avoiding subrequests and filesystem access used in the `ap_add_cgi_vars()` implementation.

`request.`**add_common_vars**()
Use of this method is discouraged, use `request.add_cgi_vars()` instead.

Calls the Apache `ap_add_common_vars()` function. After a call to this method, `request.subprocess_env` will contain *some* CGI information.

`request.`**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: If you pass this function an invalid handler, an exception will be generated at the time an attempt is made to find the handler.

request.**add_input_filter**(*filter_name*)
Adds the named filter into the input filter chain for the current request. The filter should be added before the first attempt to read any data from the request.

request.**add_output_filter**(*filter_name*)
Adds the named filter into the output filter chain for the current request. The filter should be added before the first attempt to write any data for the response.

Provided that all data written is being buffered and not flushed, this could be used to add the "CONTENT_LENGTH" filter into the chain of output filters. The purpose of the "CONTENT_LENGTH" filter is to add a `Content-Length:` header to the response.:

```
req.add_output_filter("CONTENT_LENGTH")
req.write("content",0)
```

request.**allow_methods**(*methods*[, *reset*])
Adds methods to the `request.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.

request.**auth_name**()
Returns AuthName setting.

request.**auth_type**()
Returns AuthType setting.

request.**construct_url**(*uri*)
This function returns a fully qualified URI string from the path specified by uri, using the information stored in the request to determine the scheme, server name and port. The port number is not included in the string if it is the same as the default port 80.

For example, imagine that the current request is directed to the virtual server www.modpython.org at port 80. Then supplying `'/index.html'` will yield the string `'http://www.modpython.org/index.html'`.

request.**discard_request_body**()
Tests for and reads any message body in the request, simply discarding whatever it receives.

request.**document_root**()
Returns DocumentRoot setting.

request.**get_basic_auth_pw**()
Returns a string containing the password when Basic authentication is used.

request.**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 `request.get_options()`. The table has directives as keys, and their values, if any, as values.

request.**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.

`request.`**`get_options`**`()`

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

`request.`**`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, `request.prev()` will contain a reference to a request object from which it was redirected.

`request.`**`is_https`**`()`

Returns non-zero if the connection is using SSL/TLS. Will always return zero if the mod_ssl Apache module is not loaded.

You can use this method during any request phase, unlike looking for the `HTTPS` variable in the `request.subprocess_env` member dictionary. This makes it possible to write an authentication or access handler that makes decisions based upon whether SSL is being used.

Note that this method will not determine the quality of the encryption being used. For that you should call the *ssl_var_lookup* method to get one of the *SSL_CIPHER** variables.

`request.`**`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.

`request.`**`meets_conditions`**`()`

Calls the Apache `ap_meets_conditions()` function which returns a status code. If *status* is `apache.OK`, generate the content of the response normally. If not, simply return *status*. Note that *mtime* (and possibly the ETag header) should be set as appropriate prior to calling this function. The same goes for `request.status()` if the status differs from `apache.OK`.

Example:

```
# ...
r.headers_out['ETag'] = '"1130794f-3774-4584-a4ea-0ab19e684268"'
r.headers_out['Expires'] = 'Mon, 18 Apr 2005 17:30:00 GMT'
r.update_mtime(1000000000)
r.set_last_modified()

status = r.meets_conditions()
if status != apache.OK:
   return status

# ... do expensive generation of the response content ...
```

request.**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
```

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

request.**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.

request.**readline**([*len*])

Like `request.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'`.

request.**readlines**([*sizehint*])

Reads all lines using `request.readline()` and returns a list of the lines read. If the optional *sizehint* parameter is given in, the method will read at least *sizehint* bytes of data, up to the completion of the line in which the *sizehint* bytes limit is reached.

request.**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.

request.**register_input_filter**(*filter_name*, *filter*[, *dir*])

Allows dynamic registration of mod_python input filters. *filter_name* is a string which would then subsequently be used to identify the filter. *filter* is a string containing the name of the module and the filter function. Optional *dir* is a string containing the name of the directory to be added to the pythonpath. 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).

The registration of the filter this way only persists for the life of the request. To actually add the filter into the chain of input filters for the current request `request.add_input_filter()` would be used.

request.**register_output_filter**(*filter_name*, *filter*[, *dir*])

Allows dynamic registration of mod_python output filters. *filter_name* is a string which would then subsequently be used to identify the filter. *filter* is a string containing the name of the module and the filter function. Optional *dir* is a string containing the name of the directory to be added to the pythonpath. 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).

The registration of the filter this way only persists for the life of the request. To actually add the filter into the chain of output filters for the current request `request.add_output_filter()` would be used.

request.**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).

Returns the number of bytes sent, or raises an IOError exception on failure.

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

request.**set_etag**()

Sets the outgoing `'ETag'` header.

request.**set_last_modified**()

Sets the outgoing `Last-Modified` header based on value of `mtime` attribute.

request.**ssl_var_lookup**(*var_name*)

Looks up the value of the named SSL variable. This method queries the mod_ssl Apache module directly, and may therefore be used in early request phases (unlike using the `request.subprocess_env` member.

If the mod_ssl Apache module is not loaded or the variable is not found then `None` is returned.

If you just want to know if a SSL or TLS connection is being used, you may consider calling the `is_https` method instead.

It is unfortunately not possible to get a list of all available variables with the current mod_ssl implementation, so you must know the name of the variable you want. Some of the potentially useful ssl variables are listed below. For a complete list of variables and a description of their values see the mod_ssl documentation.:

```
SSL_CIPHER
SSL_CLIENT_CERT
SSL_CLIENT_VERIFY
SSL_PROTOCOL
SSL_SESSION_ID
```

Note: Not all SSL variables are defined or have useful values in every request phase. Also use caution when relying on these values for security purposes, as SSL or TLS protocol parameters can often be renegotiated at any time during a request.

---

request.**update_mtime**(*dependency_mtime*)

If *ependency_mtime* is later than the value in the `mtime` attribute, sets the attribute to the new value.

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

`request.`**`flush`**()
Flushes the output buffer.

`request.`**`set_content_length`**(*len*)
Sets the value of `request.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 `request.write()`), calling the method is meaningless.

Request Members

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

`request.`**`server`**
A server object associated with this request. See *Server Object (mp_server)* for more details. *(Read-Only)*

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

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

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

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

`request.`**`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 figured 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.

`request.`**`proxyreq`**
A proxy request: one of `apache.PROXYREQ_*` values.

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

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

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

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

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

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

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

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

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

request.**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 request.allow_methods() method, described in section *Request Methods*. *(Read-Only)*

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

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

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

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

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

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

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

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

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

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

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

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

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

request.**headers_in**
A table object containing headers sent by the client.

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

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

request.**subprocess_env**

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

request.**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 request.notes.

request.**phase**

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

request.**interpreter**

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

request.**content_type**

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

request.**content_languages**

Tuple. List of strings representing the content languages.

request.**handler**

The symbolic name of the content handler (as in module, not mod_python handler) that will service the request during the response phase. When the SetHandler/AddHandler directives are used to trigger mod_python, this will be set to 'mod_python' by mod_mime. A mod_python handler executing prior to the response phase may also set this to 'mod_python' along with calling request.add_handler() to register a mod_python handler for the response phase:

```
def typehandler(req):
   if os.path.splitext(req.filename)[1] == ".py":
      req.handler = "mod_python"
     req.add_handler("PythonHandler", "mod_python.publisher")
     return apache.OK
   return apache.DECLINED
```

request.**content_encoding**

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

request.**vlist_validator**

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

request.**user**

If an authentication check is made, this will hold the user name. Same as CGI REMOTE_USER.

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

request.**ap_auth_type**

Authentication type. Same as CGI AUTH_TYPE.

request.**no_cache**

Boolean. This response cannot be cached.

request.**no_local_copy**

Boolean. No local copy exists.

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

request.**uri**
The path portion of the URI.

request.**filename**
String. File name being requested.

request.**canonical_filename**
String. The true filename (request.filename is canonicalized if they don't match).

request.**path_info**
String. What follows after the file name, but is before query args, if anything. Same as CGI PATH_INFO.

request.**args**
String. Same as CGI QUERY_ARGS.

request.**finfo**
A file information object with type mp_finfo, analogous to the result of the POSIX stat function, describing the file pointed to by the URI. The object provides the attributes fname, filetype, valid, protection, user, group, size, inode, device, nlink, atime, mtime, ctime and name.

The attribute may be assigned to using the result of apache.stat(). For example:

```
if req.finfo.filetype == apache.APR_DIR:
  req.filename = posixpath.join(req.filename, 'index.html')
  req.finfo = apache.stat(req.filename, apache.APR_FINFO_MIN)
```

For backward compatibility, the object can also be accessed as if it were a tuple. The apache module defines a set of FINFO_* constants that should be used to access elements of this tuple.:

```
user = req.finfo[apache.FINFO_USER]
```

request.**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)

request.**used_path_info**
Flag to accept or reject path_info on current request.

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

request.**useragent_addr**
Apache 2.4 only

The (address, port) tuple for the user agent.

This attribute should reflect the address of the user agent and not necessarily the other end of the TCP connection, for which there is connection.client_addr. *(Read-Only)*

request.**useragent_ip**
Apache 2.4 only

String with the IP of the user agent. Same as CGI REMOTE_ADDR.

This attribute should reflect the address of the user agent and not necessarily the other end of the TCP connection, for which there is connection.client_ip. *(Read-Only)*

4.5.5 Connection Object (mp_conn)

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

Connection Methods

`connection.`**`log_error`**`(`*message*[, *level*]`)`

An interface to the Apache `ap_log_cerror` 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 connection or
request object, use the :func:`apache.log_error` function.
```

`connection.`**`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 behavior 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.

`connection.`**`readline`**`(`[*length*]`)`

Reads a line from the connection or up to *length* bytes.

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

`connection.`**`write`**`(`*string*`)`

Writes *string* to the client.

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

Connection Members

`connection.`**`base_server`**

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

`connection.`**`local_addr`**

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

`connection.`**`remote_addr`**

Deprecated in Apache 2.4, use client_addr. (Aliased to client_addr for backward compatibility)

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

connection.**client_addr**
Apache 2.4 only

The (address, port) tuple for the client.

This attribute reflects the other end of the TCP connection, which may not always be the address of the user agent. A more accurate source of the user agent address is `request.useragent_addr`. *(Read-Only)*

connection.**remote_ip**
Deprecated in Apache 2.4, use client_ip. (Aliased to client_ip for backward compatibility)

String with the IP of the client. In Apache 2.2 same as CGI `REMOTE_ADDR`. *(Read-Only)*

connection.**client_ip**
Apache 2.4 only

String with the IP of the client.

This attribute reflects the other end of the TCP connection, which may not always be the address of the user agent. A more accurate source of the user agent address is `request.useragent_ip`.

(Read-Only)

connection.**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)*

connection.**remote_logname**
Remote name if using **RFC 1413** (ident). Same as CGI

`REMOTE_IDENT`. *(Read-Only)*

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

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

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

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

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

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

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

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

4.5.6 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

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

`filter.`**`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 *length* argument is negative or omitted, reads all data currently available.

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

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

`filte.`**`flush`**`()`
: Flushes the output by sending a FLUSH bucket.

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

`filter.`**`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

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

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

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

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

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

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

`server.get_config()`
Similar to `request.get_config()`, but returns a table object holding only the mod_python configuration defined at global scope within the Apache configuration. That is, outside of the context of any VirtualHost, Location, Directory or Files directives.

`server.get_options()`
Similar to `request.get_options()`, but returns a table object holding only the mod_python options defined at global scope within the Apache configuration. That is, outside of the context of any VirtualHost, Location, Directory or Files directives.

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

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

`server.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 the request object be supplied to infer the interpreter name. If you don't have any request object at hand, then you must use the `apache.register_cleanup()` variant.

Note: *Warning:* do not pass directly or indirectly a request object in the data parameter. Since the callable will be called at server shutdown time, the request object won't exist anymore and any manipulation of it in the callable will give undefined behaviour.

Server Members

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

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

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

`server.server_hostname`
Value of the `ServerName` directive. Same as CGI

`SERVER_NAME`. *(Read-Only)*

`server.names`
Tuple. List of normal server names specified in the `ServerAlias` directive. This list does not include wildcarded names, which are listed separately in `wild_names`. *(Read-Only)*

server.**wild_names**
Tuple. List of wildcarded server names specified in the `ServerAlias` directive. *(Read-Only)*

server.**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)*

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

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

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

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

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

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

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

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

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

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

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

server.**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:

RFC 3875 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 `util.`**`FieldStorage`**(*req*[, *keep_blank_values*[, *strict_parsing*[, *file_callback*[, *field_callback*]]]])

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.

The optional argument *file_callback* allows the application to override both file creation/deletion semantics and location. See *FieldStorage Examples* for additional information. *New in version 3.2*

The optional argument *field_callback* allows the application to override both the creation/deletion semantics and behavior. *New in version 3.2*

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`. A suggested strategy for dealing with this is that any handler should first check for the existence of a `form` attribute within the request object. If this exists, it should be taken to be an existing instance of the `FieldStorage` class and that should be used. If the attribute does not exist and needs to be created, it should be cached as the `form` attribute of the request object so later handler code can use it.

When the `FieldStorage` class instance is created, 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 `Field.file` attribute of a `Field` object.

The `FieldStorage` class has a mapping object interface, i.e. it can be treated like a dictionary in most instances, but is not strictly compatible as is it missing some methods provided by dictionaries and some methods don't behave entirely like their counterparts, especially when there is more than one value associated with a form field. 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 `StringField.value` attribute for compatibility with standard library `cgi` module.
- An instance 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 `StringFile.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

`util.add_field(name, value)`
: Adds an additional form field with *name* and *value*. If a form field already exists with *name*, the *value* will be added to the list of existing values for the form field. This method should be used for adding additional fields in preference to adding new fields direct to the list of fields.

If the value associated with a field should be replaced when it already exists, rather than an additional value being associated with the field, the dictionary like subscript operator should be used to set the value, or the existing field deleted altogether first using the `del` operator.

`util.clear()`
: Removes all form fields. Individual form fields can be deleted using the `del` operator.

`util.get(name, default)`
: If there is only one value associated with form field *name*, that single value will be returned. If there are multiple values, a list is returned holding all values. If no such form field or value exists then the method returns the value specified by the parameter *default*. A subscript operator is also available which yields the same result except that an exception will be raised where the form field *name* does not exist.

`util.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.

`util.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.

`util.has_key(name)`
: Returns `True` if *name* is a valid form field. The `in` operator is also supported and will call this method.

`util.items()`
: Returns a list consisting of tuples for each combination of form field name and value.

`util.keys()`
: This method returns the names of the form fields. The `len` operator is also supported and will return the number of names which would be returned by this method.

4.6.2 FieldStorage Examples

The following examples demonstrate how to use the *file_callback* parameter of the `FieldStorage` constructor to control file object creation. The `Storage` classes created in both examples derive from FileType, thereby providing extended file functionality.

These examples are provided for demonstration purposes only. The issue of temporary file location and security must be considered when providing such overrides with mod_python in production use.

Simple file control using class constructor

This example uses the `FieldStorage` class constructor to create the file object, allowing simple control. It is not advisable to add class variables to this if serving multiple sites from apache. In that case use the factory method instead:

```
class Storage(file):

   def __init__(self, advisory_filename):
       self.advisory_filename = advisory_filename
       self.delete_on_close = True
       self.already_deleted = False
       self.real_filename = '/someTempDir/thingy-unique-thingy'
       super(Storage, self).__init__(self.real_filename, 'w+b')

   def close(self):
       if self.already_deleted:
           return
       super(Storage, self).close()
       if self.delete_on_close:
           self.already_deleted = True
           os.remove(self.real_filename)

   request_data = util.FieldStorage(request, keep_blank_values=True, file_callback=Storage)
```

Advanced file control using object factory

Using a object factory can provide greater control over the constructor parameters:

```
import os

class Storage(file):

    def __init__(self, directory, advisory_filename):
        self.advisory_filename = advisory_filename
        self.delete_on_close = True
        self.already_deleted = False
        self.real_filename = directory + '/thingy-unique-thingy'
        super(Storage, self).__init__(self.real_filename, 'w+b')

    def close(self):
        if self.already_deleted:
            return
        super(Storage, self).close()
        if self.delete_on_close:
            self.already_deleted = True
            os.remove(self.real_filename)

class StorageFactory:

    def __init__(self, directory):
        self.dir = directory

    def create(self, advisory_filename):
        return Storage(self.dir, advisory_filename)

file_factory = StorageFactory(someDirectory)
# [...sometime later...]
request_data = util.FieldStorage(request, keep_blank_values=True,
                                 file_callback=file_factory.create)
```

4.6.3 Field class

class util.**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-like object. For file uploads it points to a TemporaryFile instance. (For more information see the TemporaryFile class in the standard python tempfile module.

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

4.6.4 Other functions

util.**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.

`util.`**`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.

`util.`**`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 with a value of `apache.DONE` to ensuring that any later phases or stacked handlers do not run. 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.

More specifically, the biggest difference between Netscape and RFC cookies is that RFC cookies are sent from the browser to the server along with their attributes (like Path or Domain). The `Cookie` module ignore those incoming attributes, so all incoming cookies end up as Netscape-style cookies, without any of their attributes defined.

See Also:

Persistent Client State - HTTP Cookies for the original Netscape specification.

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

RFC 2694 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 by David M. Kristol for an excellent overview of the issues surrounding standardization of Cookies.

4.7.1 Classes

class `Cookie.`**`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, httponly, __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 `Cookie.`**`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 :meth: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 `Cookie.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 marchal module documentation.

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

Functions

`Cookie.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)

`Cookie.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()`). The set of cookies found is returned as a dictionary.

`Cookie.get_cookie`(*req*, *name*[, *Class*[, *data*]])

This is a convenience function for retrieving a single named cookie from incoming headers. *req* is a mod_python `Request` object. *name* is the name of the cookie. *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()`). The cookie if found is returned, otherwise `None` is returned.

4.7.2 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_cookies(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), `DbmSession` class, which uses a dbm to store sessions, and `FileSession` class, which uses individual files 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.`**`Session`**(*req*[, *sid*[, *secret*[, *timeout*[, *lock*]]]])

`Session()` takes the same arguments as `BaseSession`.

This function returns a instance of the default session class. The session class to be used can be specified using `PythonOption mod_python.session.session_type value`, where *value* is one of `DbmSession`, `MemorySession` or `FileSession`. Specifying custom session classes using `PythonOption` session is not yet supported.

If session type option is not found, the function queries the MPM and based on that returns either a new instance of `DbmSession` or `MemorySession`. `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.

Note that on Windows if you are using multiple Python interpreter instances and you need sessions to be shared between applications running within the context of the distinct Python interpreter instances, you must specifically indicate that `DbmSession` should be used, as `MemorySession` will only allow a session to be valid within the context of the same Python interpreter instance.

Also note that the option name `mod_python.session.session_type` only started to be used from mod_python 3.3 onwards. If you need to retain compatibility with older versions of mod_python, you should use the now obsolete `session` option instead.

class `Session.`**`BaseSession`**(*req*[, *sid*[, *secret*[, *timeout*[, *lock*]]]])

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 the directory `PythonHandler` was specified was `/a/b/c/d/e`, the path will be set to `/d/e`.

The deduction of the path in this way will only work though where the `Directory` directive is used and the directory is actually within the document root. If the `Location` directive is used or the directory is outside of the document root, the path will be set to `/`. You can force a specific path by setting the `mod_python.session.application_path` option (`'PythonOption mod_python.session.application_path /my/path'` in server configuration).

Note that prior to mod_python 3.3, the option was `ApplicationPath`. If your system needs to be compatible with older versions of mod_python, you should continue to use the now obsolete option name.

The domain of a cookie is by default not set for a session and as such the session is only valid for the host which generated it. In order to have a session which spans across common sub domains, you can specify the parent domain using the `mod_python.session.application_domain` option (`'PythonOption mod_python.session.application_domain mod_python.org'` 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 and a save performed, within the *timeout* period. Upon a save occurring the time of last access is updated and the period until the session will timeout be reset. The default *timeout* period is 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.

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 :meth:req.register_cleanup‘ to register a cleanup which will be executed after the request has been processed.

class Session.**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. An alternative directory can be specified using `PythonOption mod_python.dbm_session.database_directory /path/to/directory`. The path and filename can can be overridden by setting `PythonOption mod_python.dbm_session.database_filename filename`.

Note that the above names for the `PythonOption` settings were changed to these values in mod_python 3.3. If you need to retain compatibility with older versions of mod_python, you should continue to use the now obsolete `session_directory` and `session_dbm` options.

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.

class `Session.`**`FileSession`**(*req*[, *sid*[, *secret*[, *timeout*[, *lock*[, *fast_cleanup*[, *verify_cleanup*]]]]]])

New in version 3.2.0.

This class provides session storage using a separate file for each session. It is a subclass of `BaseSession`.

Session data is stored in a separate file for each session. These files are not deleted when the server process is stopped, so sessions are persistent across server restarts. The session files are saved in a directory named mp_sess in the temporary directory returned by the `tempfile.gettempdir()` standard library function. An alternate path can be set using `PythonOption mod_python.file_session.database_directory /path/to/directory`. This directory must exist and be readable and writeable by the apache process.

Note that the above name for the `PythonOption` setting was changed to these values in mod_python 3.3. If you need to retain compatibility with older versions of mod_python, you should continue to use the now obsolete `session_directory` option.

Expired session files are periodically removed by the cleanup mechanism. The behaviour of the cleanup can be controlled using the *fast_cleanup* and *verify_cleanup* parameters, as well as `PythonOption mod_python.file_session.cleanup_time_limit` and `PythonOption mod_python.file_session.cleanup_grace_period`.

- *fast_cleanup*

  A boolean value used to turn on FileSession cleanup optimization. Default is *True* and will result in reduced cleanup time when there are a large number of session files.

  When *fast_cleanup* is True, the modification time for the session file is used to determine if it is a candidate for deletion. If `(current_time - file_modification_time) > (timeout + grace_period)`, the file will be a candidate for deletion. If *verify_cleanup* is False, no futher checks will be made and the file will be deleted.

  If *fast_cleanup* is False, the session file will unpickled and it's timeout value used to determine if the session is a candidate for deletion. *fast_cleanup* = `False` implies *verify_cleanup* = `True`.

  The timeout used in the fast_cleanup calculation is same as the timeout for the session in the current request running the filesession_cleanup. If your session objects are not using the same timeout, or you are manually setting the timeout for a particular session with `set_timeout()`, you will need to set *verify_cleanup* = `True`.

  The value of *fast_cleanup* can also be set using `PythonOption mod_python.file_session.enable_fast_cleanup`.

- *verify_cleanup*

Boolean value used to optimize the FileSession cleanup process. Default is `True`.

If *verify_cleanup* is True, the session file which is being considered for deletion will be unpickled and its timeout value will be used to decide if the file should be deleted.

When *verify_cleanup* is False, the timeout value for the current session will be used in to determine if the session has expired. In this case, the session data will not be read from disk, which can lead to a substantial performance improvement when there are a large number of session files, or where each session is saving a large amount of data. However this may result in valid sessions being deleted if all the sessions are not using a the same timeout value.

The value of *verify_cleanup* can also be set using `PythonOption mod_python.file_session.verify_session_timeout`.

- **`PythonOption mod_python.file_session.cleanup_time_limit [value]`** Integer value in seconds. Default is 2 seconds.

  Session cleanup could potentially take a long time and be both cpu and disk intensive, depending on the number of session files and if each file needs to be read to verify the timeout value. To avoid overloading the server, each time filesession_cleanup is called it will run for a maximum of *session_cleanup_time_limit* seconds. Each cleanup call will resume from where the previous call left off so all session files will eventually be checked.

  Setting *session_cleanup_time_limit* to 0 will disable this feature and filesession_cleanup will run to completion each time it is called.

- `PythonOption mod_python.file_session.cleanup_grace_period [value]` Integer value in seconds. Default is 240 seconds. This value is added to the session timeout in determining if a session file should be deleted.

  There is a small chance that a the cleanup for a given session file may occur at the exact time that the session is being accessed by another request. It is possible under certain circumstances for that session file to be saved in the other request only to be immediately deleted by the cleanup. To avoid this race condition, a session is allowed a *grace_period* before it is considered for deletion by the cleanup. As long as the grace_period is longer that the time it takes to complete the request (which should normally be less than 1 second), the session will not be mistakenly deleted by the cleanup.

  The default value should be sufficient for most applications.

class `Session.`**`MemorySession`**(*req*[, *sid*[, *secret*[, *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. It also cannot be used where multiple Python interpreters are used within the one Apache process and it is necessary to share sessions between applications running in the distinct interpreters.

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

4.8.2 Examples

The following example demonstrates a simple hit counter.:

```
from mod_python import Session

def handler(req):
    session = Session.Session(req)

    try:
        session['hits'] += 1
```

```
    except:
        session['hits'] = 1

    session.save()

    req.content_type = 'text/plain'
    req.write('Hits: %d\n' % session['hits'])
    return apache.OK
```

4.9 psp – Python Server Pager

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 *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.write("""
</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 `mod_python.psp.cache_database_filename` Python option, e.g.:

```
PythonOption mod_python.psp.cache_database_filename "/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.

Note that the above name for the option setting was only changed to this value in mod_python 3.3. If you need to retain backward compatibility with older versions of mod_python use the `PSPDbmCache` option instead.

`PSP.`**`run`**`(`[*vars*[, *flush*]]`)`

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. Optional argument *flush* is a boolean flag indicating whether output should be flushed. The default is not to flush output.

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 `PSPInterface`.

`PSP.`**`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 `psp.`**`PSPInterface`**

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:

psp.**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).

psp.**parsestring** (*string*)
This function will parse contents of *string* and return a string of resulting Python code.

4.10 `httpdconf` – HTTPd Configuration

The `httpdconf` module provides a simple framework for generating Apache HTTP Server configuration in Python. It was inspired by HTMLgen by Robin Friedrich. `httpdconf` appeared in 2002 as part of the mod_python test framework and its use has been primarily limited to mod_python tests. This latest version of mod_python includes many improvements to `httpdconf` and makes it part of the Python API.

The basic idea is that an Apache configuration directive can be specified as Python code, e.g.:

```
>>> from mod_python.httpdconf import *
>>> conf = DocumentRoot('/path/to/htdocs')
```

The resulting object renders itself as a valid Apache directive when converted to string:

```
>>> print conf
DocumentRoot /path/to/htdocs
```

While the `__repr__` method of the object returns the string of Python code used to construct it in the first place:

```
>>> print `conf`
DocumentRoot('/path/to/htdocs')
```

4.10.1 Classes for Directive types

httpdconf separates all Apache directives into the following classes.

class `httpdconf.`**`Directive`**(*name*, *value*[, *flipslash=1*])

This is a simple directive. Its syntax is the directive *name* followed by a string *value*. Even though the Apache directives can be followed by multiple arguments, httpdconf views it as just a single string, e.g. `CustomLog('logs/access_log combined')`.

class `httpdconf.`**`Container`**(**args*[, *only_if=None*])

A Container groups directives specified as *args* into a single object. args can include other containers as well. The optional *only_if* argument is a string of Python that is evaled at directive render time. The directive is rendered only if the eval return a true value.

```
>>> c = Container(CustomLog('logs/access_log combined'), ErrorLog('logs/error_log'))
>>> print c
CustomLog logs/access_log combined
ErrorLog logs/error_log

>>> print `c`
Container(
    CustomLog('logs/access_log combined'),
    ErrorLog('logs/error_log'),
    only_if='True')
)
```

Note how elements within a Container are properly indented when rendered as Python code. A more practical example of only_if may be `only_if="mod_python.version.HTTPD_VERSION[0:3] == '2.4'"`.

`append`(*value*)

Appends an object to a container. There is no difference between specifying contained object at creation time or appending elements to a container later.

`ContainerTag(tag, attr, args[, flipslash=1)]`

A ContainerTag is a tag that contains other tags, e.g. `Directory` or `Location`.

class `httpdconf.`**`Comment`**(*comment*)

A Comment renders itself as an Apache configuration comment. There is no need to include `#` as part of the *comment* string. Multi-line comments can be specified by a newline charater. Example:

```
>>> c = Comment("\nThis is\na comment\n")
>>> print c
#
# This is
# a comment

>>> print `c`
Comment('\n'
        'This is\n'
        'a comment\n')
```

httpdconf includes a basic set of Apache configuration directives (see code for which ones), but any Apache configuration directive can be trivially created by sub-classing one of the above classes:

```
>>> from mod_python.httpdconf import *
>>> class MyDirective(Directive):
...     def __init__(self, val):
...         Directive.__init__(self, self.__class__.__name__, val)
...
>>> c = MyDirective('foo')
>>> print c
MyDirective foo
```

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 an optional 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` or `apache.DECLINED`, then execution of all subsequent handlers for that phase is aborted. What happens when either `apache.OK` or `apache.DECLINED` is returned is dependent on which phase is executing.

Note that prior to mod_python 3.3, if any handler in the sequence, no matter which phase was executing, returned a value other than `apache.OK`, then execution of all subsequent handlers for that phase was 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) or an actual path to a module code file. The module is loaded using the mod_python module importer as implemented by the `apache.import_module()` function. Reference should be made to the documentation of that function for further details of how module importing is managed.

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 Request Handler*.

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) `'::'` removes the time-consuming work of figuring out where the module part ends and the object inside of it begins, resulting in a modest performance gain.

The handlers in this document are listed in order in which phases are processed by Apache.

5.1.2 Python*Handlers and Python path

If a `Python*Handler` directive is specified in a *directory section* (i.e. inside a `<Directory></Directory>` or `<DirectoryMatch></DirectoryMatch>` or in an `.htaccess` file), then this directory is automatically prepended to the Python path (`sys.path`) *unless* Python path is specified explicitely with the `PythonPath` directive.

If a `Python*Handler` directive is specified in a *location section* (i.e. inside `<Location></Location>` or `<LocationMatch></LocationMatch>`), then no path modification is done automatically and in most cases a `PythonPath` directive is required to augment the path so that the handler module can be imported.

Also for `Python*Handlers` inside a location section mod_python disables the phase of the request that maps the URI to a file on the filesystem (`ap_hook_map_to_storage`). This is because there is usually no link between path specified in `<Location>` and the filesystem, while attempting to map to a filesystem location results in unnecessary and expensive filesystem calls. Note that an important side-effect of this is that once a request URI has been matched to a `<Location>` containing a mod_python handler, all `<Directory>` and `<DirectoryMatch>` directives and their contents are ignored for this request.

5.1.3 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.

Where multiple handlers are specified, if any handler in the sequence returns a value other than `apache.OK` or `apache.DECLINED`, then execution of all subsequent handlers for this phase is aborted.

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.

5.1.4 PythonTransHandler

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

This handler 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.

Where multiple handlers are specified, if any handler in the sequence returns a value other than `apache.DECLINED`, then execution of all subsequent handlers for this phase is aborted.

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.

5.1.5 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.

Where multiple handlers are specified, if any handler in the sequence returns a value other than `apache.OK` or `apache.DECLINED`, then execution of all subsequent handlers for this phase is aborted.

5.1.6 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 :file'.htaccess' and directory.

Where multiple handlers are specified, if any handler in the sequence returns a value other than `apache.OK` or `apache.DECLINED`, then execution of all subsequent handlers for this phase is aborted.

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)

5.1.7 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.

Where multiple handlers are specified, if any handler in the sequence returns a value other than `apache.OK` or `apache.DECLINED`, then execution of all subsequent handlers for this phase is aborted.

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.

5.1.8 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).

Where multiple handlers are specified, if any handler in the sequence returns a value other than `apache.DECLINED`, then execution of all subsequent handlers for this phase is aborted.

To obtain the username, use `req.user`. To obtain the password entered by the user, use the `request.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: `request.get_basic_auth_pw()` must be called prior to using the `request.user` value. Apache makes no attempt to decode the authentication information unless `request.get_basic_auth_pw()` is called.

5.1.9 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.

Where multiple handlers are specified, if any handler in the sequence returns a value other than `apache.DECLINED`, then execution of all subsequent handlers for this phase is aborted.

5.1.10 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.

Where multiple handlers are specified, if any handler in the sequence returns a value other than `apache.DECLINED`, then execution of all subsequent handlers for this phase is aborted.

5.1.11 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.

Where multiple handlers are specified, if any handler in the sequence returns a value other than `apache.OK` or `apache.DECLINED`, then execution of all subsequent handlers for this phase is aborted.

5.1.12 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.

Where multiple handlers are specified, if any handler in the sequence returns a status value other than `apache.OK` or `apache.DECLINED`, then execution of subsequent handlers for the phase are skipped and the return status becomes that for the whole content handler phase. If all handlers are run, the return status of the final handler is what becomes the return status of the whole content handler phase. Where that final status is `apache.DECLINED`, Apache will fall back to using the `default-handler` and attempt to serve up the target as a static file.

5.1.13 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.

Where multiple handlers are specified, if any handler in the sequence returns a value other than `apache.OK` or `apache.DECLINED`, then execution of all subsequent handlers for this phase is aborted.

5.1.14 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.

The *module* referred to by the handler can be a full module name (package dot notation is accepted) or an actual path to a module code file. The module is loaded using the mod_python module importer as implemented by the `apache.import_module()` function. Reference should be made to the documentation of that function for further details of how module importing is managed.

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.

The *module* referred to by the handler can be a full module name (package dot notation is accepted) or an actual path to a module code file. The module is loaded using the mod_python module importer as implemented by the `apache.import_module()` function. Reference should be made to the documentation of that function for further details of how module importing is managed.

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'`.

The *module* can be a full module name (package dot notation is accepted) or an absolute path to a module code file. The module is loaded using the mod_python module importer as implemented by the `apache.import_module()` function. Reference should be made to the documentation of that function for further details of how module importing is managed.

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

The *module* can be a full module name (package dot notation is accepted) or an absolute path to a module code file. The module is loaded using the mod_python module importer as implemented by the `apache.import_module()` function. Reference should be made to the documentation of that function for further details of how module importing is managed.

The `PythonImport` directive is useful for initialization tasks that could be time consuming and should not be done at the time of processing a request, e.g. initializing a database connection. Where such initialization code could fail and cause the importing of the module to fail, it should be placed in its own function and the alternate syntax used:

`PythonImport *module::function* *interpreter_name*`

The named function will be called only after the module has been imported successfully. The function will be called with no arguments.

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 `request.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:

Multiple Interpreters 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:

Multiple Interpreters 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 a *PythonInterpPerDirectory* or *PythonInterpPerDirective* direcive.

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:

Multiple Interpreters 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:

```
PythonAuthenHandler mymodule
PythonHandler mymodule
PythonLogHandler mymodule
```

one can simply use:

```
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 `request.get_options()` function. This is useful to pass information between the apache configuration files (`httpd.conf`, `.htaccess`, etc) and the Python programs. If the value is omitted or empty (`""`), then the key is removed from the local configuration.

Reserved PythonOption Keywords

Some `PythonOption` keywords are used for configuring various aspects of mod_python. Any keyword starting with mod_python.* should be considered as reserved for internal mod_python use.

Users are encouraged to use their own namespace qualifiers when creating add-on modules, and not pollute the global namespace.

The following PythonOption keys are currently used by mod_python.

mod_python.mutex_directory
mod_python.mutex_locks
mod_python.psp.cache_database_filename
mod_python.session.session_type
mod_python.session.cookie_name
mod_python.session.application_domain
mod_python.session.application_path
mod_python.session.database_directory
mod_python.dbm_session.database_filename
mod_python.dbm_session.database_directory
mod_python.file_session.enable_fast_cleanup
mod_python.file_session.verify_session_timeout
mod_python.file_session.cleanup_grace_period
mod_python.file_session.cleanup_time_limit
mod_python.file_session.database_directory
mod_python.wsgi.application
mod_python.wsgi.base_uri

session *Deprecated in 3.3, use mod_python.session.session_type*
ApplicationPath *Deprecated in 3.3, use mod_python.session.application_path*
session_cookie_name *Deprecated in 3.3, use mod_python.session.cookie_name*
session_directory *Deprecated in 3.3, use mod_python.session.database_directory*
session_dbm *Deprecated in 3.3, use mod_python.dbm_session.database_filename*
session_cleanup_time_limit *Deprecated in 3.3, use mod_python.file_session.cleanup_time_limit*
session_fast_cleanup *Deprecated in 3.3, use mod_python.file_session.enable_fast_cleanup*
session_grace_period *Deprecated in 3.3, use mod_python.file_session.cleanup_grace_period*
session_verify_cleanup *Deprecated in 3.3, use mod_python.file_session.cleanup_session_timeout*
PSPDbmCache *Deprecated in 3.3, use mod_python.psp.cache_database_filename*

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.

When multiple PythonPath directives are specified, the effect is not cumulative, last directive will override all previous ones.

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`::

```python
""" 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 `request.filename` attribute. Before importing, the file extension, if any, is discarded.

If `request.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::

```python
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*.

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 member{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 WSGI Handler

WSGI handler can run WSGI applications as described in **PEP 333**.

Assuming there exists the following minimal WSGI app residing in a file named `mysite/wsgi.py` in directory `/path/to/mysite` (so that the full path to `wsgi.py` is `/path/to/mysite/mysite/wsgi.py`):

```
def application(environ, start_response):
    status = '200 OK'
    output = 'Hello World!'

    response_headers = [('Content-type', 'text/plain'),
                        ('Content-Length', str(len(output)))]
    start_response(status, response_headers)

    return [output]
```

It can be executed using the WSGI handler by adding the following to the Apache configuration:

```
PythonHandler mod_python.wsgi
PythonOption mod_python.wsgi.application mysite.wsgi
PythonPath "sys.path+['/path/to/mysite']"
```

The above configuration will import a module named `mysite.wsgi` and will look for an `application` callable in the module.

An alternative name for the callable can be specified by appending it to the module name separated by `'::'`, e.g.:

```
PythonOption mod_python.wsgi.application mysite.wsgi::my_application
```

If you would like your application to appear under a base URI, it can be specified by wrapping your configuration in a `<Location>` block. It can also be specified via the `mod_python.wsgi.base_uri` option, but the `<Location>` method is recommended, also because it has a side-benefit of informing mod_python to skip the map-to-storage processing phase and thereby improving performance.

For example, if you would like the above application to appear under `'/wsgiapps'`, you could specify:

```
<Location /wsgiapps>
   PythonHandler mod_python.wsgi
   PythonOption mod_python.wsgi.application mysite.wsgi
   PythonPath "sys.path+['/path/to/mysite']"
</Location>
```

With the above configuration, content formerly under `http://example.com/hello` becomes available under `http://example.com/wsgiapps/hello`.

If both `<Location>` and `mod_python.wsgi.base_uri` exist, then `mod_python.wsgi.base_uri` takes precedence. `mod_python.wsgi.base_uri` cannot be `'/'` or end with a `'/'`. “Root” (or no base_uri) is a blank string, which is the default. (Note that it is allowed for `<Location>` path to be `"/"` or have a trailing slash, it will automatically be removed by mod_python before computing `PATH_INFO`).

Note:

PEP 333 describes `SCRIPT_NAME` and `PATH_INFO` environment variables which are core to the specification. Most WSGI-supporting frameworks currently in existence use the value of `PATH_INFO` as the request URI.

The two variable’s name and function originate in CGI (**RFC 3875**), which describes an environment wherein a script (or any executable’s) output could be passed on by the web server as content. A typical CGI script resides somewhere on the filesystem to which the request URI maps. As part of serving the request the server traverses the URI mapping each element to an element of the filesystem path to locate the script. Once the script is found, the portion of the URI used thus far is assigned to the `SCRIPT_NAME` variable, while the remainder of the URI gets assigned to `PATH_INFO`.

Because the relationship between Python modules and files on disk is largely tangential, it is not very clear what exactly `PATH_INFO` and `SCRIPT_NAME` ought to be. Even though Python modules are most often files on disk located somewhere in the Python path, they don’t have to be (they could be code objects constructed on-the-fly), and their location in the filesystem has no relationship to the URL structure at all.

The mismatch between CGI and WSGI results in an ambiguity which requires that the split between the two variables be explicitely specified, which is why `mod_python.wsgi.base_uri` exists. In essence `mod_python.wsgi.base_uri` (or the path in surrounding `<Location>`) is the `SCRIPT_NAME` portion of the URI and defaults to `''`.

An important detail is that `SCRIPT_NAME + PATH_INFO` should result in the original URI (encoding issues aside). Since `SCRIPT_NAME` (in its original CGI definition) referrs to an actual file, its name never ends with a slash. The slash, if any, always ends up in `PATH_INFO`. E.g. `/path/to/myscrip/foo/bar` splits into `/path/to/myscript` and `/foo/bar`. If the whole site is served by an app or a script, then `SCRIPT_NAME` is a blank string `''`, not a `'/'`.

6.3 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 *psp – Python Server Pager*.

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. You'll need to adjust your httpd configuration::

```
AddHandler mod_python .psp .psp_
PythonHandler mod_python.psp
PythonDebug On
```

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

6.4 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.

CHAPTER

SEVEN

COMMAND LINE TOOL - MOD_PYTHON

7.1 Overview of mod_python command

mod_python includes a command-line tool named `mod_python`. The `mod_python` command exists to facilitate tasks related to configuration and management of mod_python.

The general syntax for the command is `mod_python <subcommand> <arguments>` where `<subcommand>` is a separate tool with its own argument requirements.

7.2 mod_python command line tool sub-commands

7.2.1 create

`create` sub-command creates a simple Apache configuration and a skeleton directory structure necessary for placement of configuration, logs and content. It is meant to be executed only once per lifetime of a project.

The configuration generated by `create` consists of an `httpdconf` based version (in Python) which can then be used to generate an actual Apache configuration (by using the `genconfig` subcommand or simply executing the config files itself). The idea is that the Apache configuration is always generated and the Python version is the one meant for editing/adjustments.

The `create` subcommand will create the necessary files and directories if they do not exist, but will not overwrite any existing files or directories only producing a warning when a file or directory already exists. It will abort if the Python version of the configuration file already exists.

`create` requires a single argument: the distination directory, Apache `ServerRoot`.

`create` has the following command options:

-listen
: A string describing the port and optional IP address on which the server is to listen for incoming requests in the form `[ip_address:]port` The argument will be applied to the Apache `Listen` directive as is and therefore must be syntactically compatible with it.

-pythonpath
: A colon (`":"`) separate list of paths to be applied to the *PythonPath* directive.

-pythonhandler
: The name of the Python handler to use. Applied to the *PythonHandler* directive.

-pythonoption
: An option to be specified in the configuration. Multiple options are alowed. Applied to the *PythonOption* directive.

7.2.2 genconfig

This sub-command exists to facilitate re-generation of an Apache configuration from a Python-based one. All it does is run the script, but its use is recommended because the mod_python command will execute the correct version of Python under which mod_python was initially compiled. Example:

```
mod_python genconfig /path/to/server_root/httpd_conf.py > /path/to/server_root/httpd.conf
```

7.2.3 start

Starts an Apache instance. Requires a single argument, the path to Apache configuration file.

7.2.4 stop

Stops an Apache instance (using graceful-stop). Requires a single argument, the path to Apache configuration file.

7.2.5 restart

Stops an Apache instance (using graceful). Requires a single argument, the path to Apache configuration file.

7.2.6 version

This sub-command prints out version and location information about this mod_python installation, the Apache HTTP Server and Python used when building this mod_python instance.

7.2.7 Example

To create an Apache instance with all the required directories for a WSGI application which is located in `/path/to/myapp` and defined in `/path/to/myapp/myapp/myapp/wsgi.py`, run the following:

```
mod_python create /path/to/new/server_root \
    --pythonpath=/path/to/my/app \
    --pythonhandler=mod_python.wsgi \
    --pythonoption="mod_python.wsgi.application myapp.wsgi::application"
```

The above example will create a Python-based configuration in `/path/to/new/server_root/conf/http_conf.py` which is a simple Pythong script. When executed, the output of the script becomes an Apache configuration (`create` will take care of generating the first Apache config for you).

You should be able to run this Apache instance by executing:

```
mod_python start /path/to/new/server_root/conf/httpd.conf
```

CHAPTER

EIGHT

SERVER SIDE INCLUDES

8.1 Overview of SSI

SSI (Server Side Includes) are directives that are placed in HTML pages, and evaluated on the server while the pages are being served. They let you add dynamically generated content to an existing HTML page, without having to serve the entire page via a CGI program, or other dynamic technology such as a mod_python handler.

SSI directives have the following syntax::

```
<!--#element attribute=value attribute=value ... -->
```

It is formatted like an HTML comment, so if you don't have SSI correctly enabled, the browser will ignore it, but it will still be visible in the HTML source. If you have SSI correctly configured, the directive will be replaced with its results.

For a more thorough description of the SSI mechanism and how to enable it, see the SSI tutorial provided with the Apache documentation.

Version 3.3 of mod_python introduces support for using Python code within SSI files. Note that mod_python honours the intent of the Apache `IncludesNOEXEC` option to the `Options` directive. That is, if `IncludesNOEXEC` is enabled, then Python code within a SSI file will not be executed.

8.2 Using Python Code

The extensions to mod_python to allow Python code to be used in conjunction with SSI introduces the new SSI directive called `'python'`. This directive can be used in two forms, these being `'eval'` and `'exec'` modes. In the case of `'eval'`, a Python expression is used and it is the result of that expression which is substituted in place into the page.:

```
<!--#python eval="10*'HELLO '" -->
<!--#python eval="len('HELLO')" -->
```

Where the result of the expression is not a string, the value will be automatically converted to a string by applying `'str()'` to the value.

In the case of `'exec'` a block of Python code may be included. For any output from this code to appear in the page, it must be written back explicitly as being part of the response. As SSI are processed by an Apache output filter, this is done by using an instance of the mod_python `filter` object which is pushed into the global data set available to the code.:

```
<!--#python exec="
filter.write(10*'HELLO ')
filter.write(str(len('HELLO')))
" -->
```

Any Python code within the 'exec' block must have a zero first level indent. You cannot start the code block with an arbitrary level of indent such that it lines up with any indenting used for surrounding HTML elements.

Although the mod_python filter object is not a true file object, that it provides the write() method is sufficient to allow the print statement to be used on it directly. This will avoid the need to explicitly convert non string objects to a string before being output.:

```
<!--#python exec="
print >> filter, len('HELLO')
" -->
```

8.3 Scope of Global Data

Multiple instances of 'eval' or 'exec' code blocks may be used within the one page. Any changes to or creation of global data which is performed within one code block will be reflected in any following code blocks. Global data constitutes variables as well as module imports, function and class definitions.:

```
<!--#python exec="
import cgi, time, os
def _escape(object):
    return cgi.escape(str(object))
now = time.time()
" -->
<html>
<body>
<pre>
<!--#python eval="_escape(time.asctime(time.localtime(now)))"-->

<!--#python exec="
keys = os.environ.keys()
keys.sort()
for key in keys:
    print >> filter, _escape(key),
    print >> filter, '=',
    print >> filter, _escape(repr(os.environ.get(key)))
" -->
</pre>
</body>
</html>
```

The lifetime of any global data is for the current request only. If data must persist between requests, it must reside in external modules and as necessary be protected against multithreaded access in the event that a multithreaded Apache MPM is used.

8.4 Pre-populating Globals

Any Python code which appears within the page has to be compiled each time the page is accessed before it is executed. In other words, the compiled code is not cached between requests. To limit such recompilation and to avoid duplication of common code amongst many pages, it is preferable to pre-populate the global data from within a mod_python handler prior to the page being processed.

In most cases, a fixup handler would be used for this purpose. For this to work, first need to configure Apache so that it knows to call the fixup handler.:

```
PythonFixupHandler _handlers
```

The implementation of the fixup handler contained in `_handlers.py` then needs to create an instance of a Python dictionary, store that in the mod_python request object as `ssi_globals` and then populate that dictionary with any data to be available to the Python code executing within the page.:

```
from mod_python import apache

import cgi, time

def _escape(object):
   return cgi.escape(str(object))

def _header(filter):
   print >> filter, '...'

def _footer(filter):
   print >> filter, '...'

def fixuphandler(req):
   req.ssi_globals = {}
   req.ssi_globals['time'] = time
   req.ssi_globals['_escape'] = _escape
   req.ssi_globals['_header'] = _header
   req.ssi_globals['_footer'] = _footer
   return apache.OK
```

This is most useful where it is necessary to insert common information such as headers, footers or menu panes which are dynamically generated into many pages.:

```
<!--#python exec="
now = time.time()
" -->
<html>
<body>
<!--#python exec="_header(filter)" -->
<pre>
<!--#python eval="_escape(time.asctime(time.localtime(now)))"-->
</pre>
<!--#python exec="_footer(filter)" -->
</body>
</html>
```

8.5 Conditional Expressions

SSI allows for some programmability in its own right through the use of conditional expressions. The structure of this conditional construct is::

```
<!--#if expr="test_condition" -->
<!--#elif expr="test_condition" -->
```

```
<!--#else -->
<!--#endif -->
```

A test condition can be any sort of logical comparison, either comparing values to one another, or testing the 'truth' of a particular value.

The source of variables used in conditional expressions is distinct from the set of global data used by the Python code executed within a page. Instead, the variables are sourced from the `subprocess_env` table object contained within the request object. The values of these variables can be set from within a page using the SSI `'set'` directive, or by a range of other Apache directives within the Apache configuration files such as `BrowserMatchNoCase` and `SetEnvIf`.

To set these variables from within a mod_python handler, the `subprocess_env` table object would be manipulated directly through the request object.:

```
from mod_python import apache

def fixuphandler(req):
   debug = req.get_config().get('PythonDebug', '0')
   req.subprocess_env['DEBUG'] = debug
   return apache.OK
```

If being done from within Python code contained within the page itself, the request object would first have to be accessed via the filter object.:

```
<!--#python exec="
debug = filter.req.get_config().get('PythonDebug', '0')
filter.req.subprocess_env['DEBUG'] = debug
" -->
<html>
<body>
<!--#if expr="${DEBUG} != 0" -->
DEBUG ENABLED
<!--#else -->
DEBUG DISABLED
<!--#endif -->
</body>
</html>
```

8.6 Enabling INCLUDES Filter

SSI is traditionally enabled using the `AddOutputFilter` directive in the Apache configuration files. Normally this would be where the request mapped to a static file.:

```
AddOutputFilter INCLUDES .shtml
```

When mod_python is being used, the ability to dynamically enable output filters for the current request can instead be used. This could be done just for where the request maps to a static file, but may just as easily be carried out where the content of a response is generated dynamically. In either case, to enable SSI for the current request, the `request.add_output_filter()` method of the mod_python request object would be used.:

```
from mod_python import apache

def fixuphandler(req):
   req.add_output_filter('INCLUDES')
   return apache.OK
```

CHAPTER

NINE

CHANGES

9.1 Changes from version 3.3.1

9.1.1 New Features

- Create the mod_python command-line tool to report version, manage Apache configuration and instances.
- Make httpdconf directives render themselves as Python, add the only_if conditional and comments.
- Expose and document httpdconf, make mod_python importable outside of Apache.
- Provide a WSGI handler.
- Change the Copyright to reflect the new status.
- Add support for Apache HTTP Server 2.4.
- Add support for Python 2.7.

9.1.2 Improvements

- Improve WSGI and Python path documentation.
- Change WSGI handler to use Location path as SCRIPT_NAME.
- Add is_location to hlist object, skip the map_to_storage for Location-wrapped Python*Handlers.
- Some optimizations to Python code to make it run faster.
- Add Mutex to Apache 2.4 tests.
- Provide and internal add_cgi_vars() implementation which does not use sub-requests.
- Many documentation clarifications and improvements.
- Add a test to ensure that req.write() and req.flush() do not leak memory (2.4 only).
- Many new tests and test framework improvements.
- Added a curl hint to the tests for easier stagin/debugging.
- Get rid of the ancient memberlist and PyMember_Get/Set calls.
- Add support for the c.remote_ip/addr to c.client_ip/addr change in 2.4. Add req.useragent_addr (also new in 2.4).
- Always check C version against Py version and warn.

- Remove APLOG_NOERRNO references.
- A more unified and cleaned up method of keeping version information.
- Convert documentation to the new reStructuredText format.
- Revert to using the old importer from 3.2.
- Replace README with README.md
- (MODPYTHON-238) Make req.chunked and req.connection.keepalive writable. Being able to set these allows chunking to be turned off when HTTP/1.1 is used but no content length supplied in response.
- (MODPYTHON-226) Make req.status_line writable.

9.1.3 Bug Fixes

- Make PythonCleanupHandler run again.
- Use PCapsule API instead of PyCObject for Python 2.7+.
- Fix SCRIPT_NAME and PATH_INFO inconsistencies so that the WSGI handler behaves correctly.
- Remove with-python-src configure option as it is no longer used to build the docs.
- (MODPYTHON-243) Fixed format string error.
- (MODPYTHON-250) Fixed MacOS X (10.5) Leopard 64 bit architecture problems.
- (MODPYTHON-249) Fixed incorrect use of APR bucket brigades shown up by APR 1.3.2.
- (MODPYTHON-245) Fix prototype of optional exported function mp_release_interpreter().
- (MODPYTHON-220) Fix 'import' from same directory as PSP file.

9.2 Changes from version 3.2.10

9.2.1 New Features

- (MODPYTHON-103) New req.add_output_filter(), req.add_input_filter(), req.register_output_fiter(), req.register_input_filter() methods. These allows the dynamic registration of filters and the attaching of filters to the current request.
- (MODPYTHON-104) Support added for using Python in content being passed through "INCLUDES" output filter, or as more commonly referred to server side include (SSI) mechanism.
- (MODPYTHON-108) Added support to cookies for httponly attribute, an extension originally created by Microsoft, but now getting more widespread use in the battle against cross site-scripting attacks.
- (MODPYTHON-118) Now possible using the PythonImport directive to specify the name of a function contained in the module to be called once the designated module has been imported.
- (MODPYTHON-124) New req.auth_name() and req.auth_type() methods. These return the values associated with the AuthName and AuthType directives respectively. The req.ap_auth_type has now also been made writable so that it can be set by an authentication handler.
- (MODPYTHON-130) Added req.set_etag(), req.set_last_modified() and req.update_mtime() functions as wrappers for similar functions provided by Apache C API. These are required to effectively use the req.meets_condition() function. The documentation for req.meets_condition() has also been updated as what it previously described probably wouldn't actually work.

- (MODPYTHON-132) New req.construct_url() method. Used to construct a fully qualified URI string incorporating correct scheme, server and port.
- (MODPYTHON-144) The "apache.interpreter" and "apache.main_server" attributes have been made publically available. These were previously private and not part of the public API.
- (MODPYTHON-149) Added support for session objects that span domains.
- (MODPYTHON-153) Added req.discard_request_body() function as wrapper for similar function provided by Apache C API. The function tests for and reads any message body in the request, simply discarding whatever it receives.
- (MODPYTHON-164) The req.add_handler(), req.register_input_filter() and req.register_output_filter() methods can now take a direct reference to a callable object as well a string which refers to a module or module::function combination by name.
- (MODPYTHON-165) Exported functions from mod_python module to be used in other third party modules for Apache. The purpose of these functions is to allow those other modules to access the mechanics of how mod_python creates interpreters, thereby allowing other modules to also embed Python and for there not to be a conflict with mod_python.
- (MODPYTHON-170) Added req._request_rec, server._server_rec and conn._conn_rec semi private members for getting accessing to underlying Apache struct as a Python CObject. These can be used for use in implementing SWIG bindings for lower level APIs of Apache. These members should be regarded as experimental and there are no guarantees that they will remain present in this specific form in the future.
- (MODPYTHON-193) Added new attribute available as req.hlist.location. For a handler executed directly as the result of a handler directive within a Location directive, this will be set to the value of the Location directive. If LocationMatch, or wildcards or regular expressions are used with Location, the value will be the matched value in the URL and not the pattern.

9.2.2 Improvements

- (MODPYTHON-27) When using mod_python.publisher, the __auth__() and __access__() functions and the __auth_realm__ string can now be nested within a class method as a well a normal function.
- (MODPYTHON-90) The PythonEnablePdb configuration option will now be ignored if Apache hasn't been started up in single process mode.
- (MODPYTHON-91) If running Apache in single process mode with PDB enabled and the "quit" command is used to exit that debug session, an exception indicating that the PDB session has been aborted is raised rather than None being returned with a subsequent error complaining about the handler returning an invalid value.
- (MODPYTHON-93) Improved util.FieldStorage efficiency and made the interface more dictionary like.
- (MODPYTHON-101) Force an exception when handler evaluates to something other than None but is otherwise not callable. Previously an exception would not be generated if the handler evaluated to False.
- (MODPYTHON-107) Neither mod_python.publisher nor mod_python.psp explicitly flush output after writing the content of the response back to the request object. By not flushing output it is now possible to use the "CONTENT_LENGTH" output filter to add a "Content-Length" header.
- (MODPYTHON-111) Note made in session documentation that a save is required to avoid session timeouts.
- (MODPYTHON-125) The req.handler attribute is now writable. This allows a handler executing in a phase prior to the response phase to specify which Apache module will be responsible for generating the content.
- (MODPYTHON-128) Made the req.canonical_filename attribute writable. Changed the req.finfo attribute from being a tuple to an actual object. For backwards compatibility the attributes of the object can still be accessed as if they were a tuple. New code however should access the attributes as member data. The req.finfo attribute is

also now writable and can be assigned to using the result of calling the new function apache.stat(). This function is a wrapper for apr_stat().

- (MODPYTHON-129) When specifying multiple handlers for a phase, the status returned by each handler is now treated the same as how Apache would treat the status if the handler was registered using the low level C API. What this means is that whereas stacked handlers of any phase would in turn previously be executed as long as they returned apache.OK, this is no longer the case and what happens is dependent on the phase. Specifically, a handler returning apache.DECLINED no longer causes the execution of subsequent handlers for the phase to be skipped. Instead, it will move to the next of the stacked handlers. In the case of PythonTransHandler, PythonAuthenHandler, PythonAuthzHandler and PythonTypeHandler, as soon as apache.OK is returned, subsequent handlers for the phase will be skipped, as the result indicates that any processing pertinent to that phase has been completed. For other phases, stacked handlers will continue to be executed if apache.OK is returned as well as when apache.DECLINED is returned. This new interpretation of the status returned also applies to stacked content handlers listed against the PythonHandler directive even though Apache notionally only ever calls at most one content handler. Where all stacked content handlers in that phase run, the status returned from the last handler becomes the overall status from the content phase.
- (MODPYTHON-141) The req.proxyreq and req.uri attributes are now writable. This allows a handler to setup these values and trigger proxying of the current request to a remote server.
- (MODPYTHON-142) The req.no_cache and req.no_local_copy attributes are now writable.
- (MODPYTHON-143) Completely reimplemented the module importer. This is now used whenever modules are imported corresponding to any of the Python*Handler, Python*Filter and PythonImport directives. The module importer is still able to be used directly using the apache.import_module() function. The new module importer no longer supports automatic reloading of packages/modules that appear on the standard Python module search path as defined by the PythonPath directive or within an application by direct changes to sys.path. Automatic module reloading is however still performed on file based modules (not packages) which are located within the document tree where handlers are located. Locations within the document tree are however no longer added to the standard Python module search path automatically as they are maintained within a distinct importer search path. The PythonPath directive MUST not be used to point at directories within the document tree. To have additional directories be searched by the module importer, they should be listed in the mod_python.importer.path option using the PythonOption directive. This is a path similar to how PythonPath argument is supplied, but MUST not reference sys.path nor contain any directories also listed in the standard Python module search path. If an application does not appear to work under the module importer, the old module importer can be reenabled by setting the mod_python.legacy.importer option using the PythonOption directive to the value '*'. This option must be set in the global Apache configuration.
- (MODPYTHON-152) When in a sub request, when a request is the result of an internal redirect, or when when returning from such a request, the req.main, req.prev and req.next members now correctly return a reference to the original Python request object wrapper first created for the specific request_rec instance rather than creating a new distinct Python request object. This means that any data added explicitly to a request object can be passed between such requests.
- (MODPYTHON-178) When using mod_python.psp, if the PSP file which is the target of the request doesn't actually exist, an apache.HTTP_NOT_FOUND server error is now returned to the client rather than raising a ValueError exception which results in a 500 internal server error. Note that if using SetHandler and the request is against the directory and no DirectoryIndex directive is specified which lists a valid PSP index file, then the same apache.HTTP_NOT_FOUND server error is returned to the client.
- (MODPYTHON-196) For completeness, added req.server.log_error() and req.connection.log_error(). The latter wraps ap_log_cerror() (when available), allowing client information to be logged along with message from a connection handler.
- (MODPYTHON-206) The attribute req.used_path_info is now modifiable and can be set from within handlers. This is equivalent to having used the AcceptPathInfo directive.
- (MODPYTHON-207) The attribute req.args is now modifiable and can be set from within handlers.

9.2.3 Bug Fixes

- (MODPYTHON-38) Fixed issue when using PSP pages in conjunction with publisher handler or where a PSP error page was being triggered, that form parameters coming from content of a POST request weren't available or only available using a workaround. Specifically, the PSP page will now use any FieldStorage object instance cached as req.form left there by preceding code.
- (MODPYTHON-43) Nested __auth__() functions in mod_python.publisher now execute in context of globals from the file the function is in and not that of mod_python.publisher itself.
- (MODPYTHON-47) Fixed mod_python.publisher so it will not return a HTTP Bad Request response when mod_auth is being used to provide Digest authentication.
- (MODPYTHON-63) When handler directives are used within Directory or DirectoryMatch directives where wildcards or regular expressions are used, the handler directory will be set to the shortest directory matched by the directory pattern. Handler directives can now also be used within Files and FilesMatch directives and the handler directory will correctly resolve to the directory corresponding to the enclosing Directory or DirectoryMatch directive, or the directory the .htaccess file is contained in.
- (MODPYTHON-76) The FilterDispatch callback should not flush the filter if it has already been closed.
- (MODPYTHON-84) The original change to fix the symlink issue for req.sendfile() was causing problems on Win32, plus code needed to be changed to work with APR 1.2.7.
- (MODPYTHON-100) When using stacked handlers and a SERVER_RETURN exception was used to return an OK status for that handler, any following handlers weren't being run if appropriate for the phase.
- (MODPYTHON-109) The Py_Finalize() function was being called on child process shutdown. This was being done though from within the context of a signal handler, which is generally unsafe and would cause the process to lock up. This function is no longer called on child process shutdown.
- (MODPYTHON-112) The req.phase attribute is no longer overwritten by an input or output filter. The filter.is_input member should be used to determine if a filter is an input or output filter.
- (MODPYTHON-113) The PythonImport directive now uses the apache.import_module() function to import modules to avoid reloading problems when same module is imported from a handler.
- (MODPYTHON-114) Fixed race conditions on setting sys.path when the PythonPath directive is being used as well as problems with infinite extension of path.
- (MODPYTHON-120) (MODPYTHON-121) Fixes to test suite so it will work on virtual hosting environments where localhost doesn't resolve to 127.0.0.1 but the actual IP address of the host.
- (MODPYTHON-126) When Python*Handler or Python*Filter directive is used inside of a Files directive container, the handler/filter directory value will now correctly resolve to the directory corresponding to any parent Directory directive or the location of the .htaccess file the Files directive is contained in.
- (MODPYTHON-133) The table object returned by req.server.get_config() was not being populated correctly to be the state of directives set at global scope for the server.
- (MODPYTHON-134) Setting PythonDebug to Off, wasn't overriding On setting in parent scope.
- (MODPYTHON-140) The util.redirect() function should be returning server status of apache.DONE and not apache.OK otherwise it will not give desired result if used in non content handler phase or where there are stacked content handlers.
- (MODPYTHON-147) Stopped directories being added to sys.path multiple times when PythonImport and PythonPath directive used.
- (MODPYTHON-148) Added missing Apache contants apache.PROXYREQ_RESPONSE and apache.HTTP_UPGRADE_REQUIRED. Also added new constants for Apache magic mime types and values for interpreting the req.connection.keepalive and req.read_body members.

- (MODPYTHON-150) In a multithread MPM, the apache.init() function could be called more than once for a specific interpreter instance whereas it should only be called once.
- (MODPYTHON-151) Debug error page returned to client when an exception in a handler occurred wasn't escaping special HTML characters in the traceback or the details of the exception.
- (MODPYTHON-157) Wrong interpreter name used for fixup handler phase and earlier, when PythonInterpPerDirectory was enabled and request was against a directory but client didn't provide the trailing slash.
- (MODPYTHON-159) Fix FieldStorage class so that it can handle multiline headers.
- (MODPYTHON-160) Using PythonInterpPerDirective when setting content handler to run dynamically with req.add_handler() would cause Apache to crash.
- (MODPYTHON-161) Directory argument supplied to req.add_handler() is canonicalized and a trailing slash added automatically. This is needed to ensure that the directory is always in POSIX path style as used by Apache and that convention where directories associated with directives always have trailing slash is adhered to. If this is not done, a different interpreter can be chosen to that expected when the PythonInterpPerDirective is used.
- (MODPYTHON-166) PythonHandlerModule was not setting up registration of the PythonFixupHandler or PythonAuthenHandler. For the latter this meant that using Require directive with PythonHandlerModule would cause a 500 error and complaint in error log about "No groups file".
- (MODPYTHON-167) When PythonDebug was On and and exception occurred, the response to the client had a status of 200 when it really should have been a 500 error status indicating that an internal error occurred. A 500 error status was correctly being returned when PythonDebug was Off.
- (MODPYTHON-168) Fixed psp_parser error when CR is used as a line terminator in psp code. This may occur with some older editors such as GoLive on Mac OS X.
- (MODPYTHON-175) Fixed problem whereby a main PSP page and an error page triggered from that page both accessing the session object would cause a deadlock.
- (MODPYTHON-176) Fixed issue whereby PSP code would unlock session object which it had inherited from the caller meaning caller could no longer use it safely. PSP code will now only unlock session if it created it in the first place.
- (MODPYTHON-179) Fixed the behaviour of req.readlines() when a size hint was provided. Previously, it would always return a single line when a size hint was provided.
- (MODPYTHON-180) Publisher would wrongly output a warning about nothing to publish if req.write() or req.sendfile() used and data not flushed, and then published function returned None.
- (MODPYTHON-181) Fixed memory leak when mod_python handlers are defined for more than one phase at the same time.
- (MODPYTHON-182) Fixed memory leak in req.readline().
- (MODPYTHON-184) Fix memory leak in apache.make_table(). This was used by util.FieldStorage class so affected all code using forms.
- (MODPYTHON-185) Fixed segfault in psp.parsestring(src_string) when src_string is empty.
- (MODPYTHON-187) Table objects could crash in various ways when the value of an item was NULL. This could occur for SCRIPT_FILENAME when the req.subprocess_env table was accessed in the post read request handler phase.
- (MODPYTHON-189) Fixed representation returned by calling repr() on a table object.
- (MODPYTHON-191) Session class will no longer accept a normal cookie if a signed cookie was expected.
- (MODPYTHON-194) Fixed potential memory leak due to not clearing the state of thread state objects before deleting them.

- (MODPYTHON-195) Fix potential Win32 resource leaks in parent Apache process when process restarts occur.
- (MODPYTHON-198) Python 2.5 broke nested __auth__/__access__/__auth_realm__ in mod_python.publisher.
- (MODPYTHON-200) Fixed problem whereby signed and marshalled cookies could not be used at the same time. When expecting marshalled cookie, any signed, but not marshalled cookies will be returned as normal cookies.

9.3 Changes from version 3.2.8

9.3.1 New Features

- (MODPYTHON-78) Added support for Apache 2.2.
- (MODPYTHON-94) New req.is_https() and req.ssl_var_lookup() methods. These communicate direct with the Apache mod_ssl module, allowing it to be determined if the connection is using SSL/TLS and what the values of internal ssl variables are.
- (MODPYTHON-131) The directory used for mutex locks can now be specified at at compile time using ./configure –with-mutex-dir value or at run time with PythonOption mod_python.mutex_directory value.
- (MODPYTHON-137) New req.server.get_options() method. This returns the subset of Python options set at global scope within the Apache configuration. That is, outside of the context of any VirtualHost, Location, Directory or Files directives.
- (MODPYTHON-145) The number of mutex locks can now be specified at run time with PythonOption mod_python.mutex_locks value.
- (MODPYTHON-172) Fixed three memory leaks that were found in _apachemodule.parse_qsl, req.readlines and util.cfgtree_walk.

9.3.2 Improvements

- (MODPYTHON-77) Third party C modules that use the simplified API for the Global Interpreter Lock (GIL), as described in PEP 311, can now be used. The only requirement is that such modules can only be used in the context of the "main_interpreter".
- (MODPYTHON-119) DbmSession unit test no longer uses the default directory for the dbm file, so the test will not interfer with the user's current apache instance.
- (MODPYTHON-158) Added additional debugging and logging output for where mod_python cannot initialise itself properly due to Python or mod_python version mismatches or missing Python module code files.

9.3.3 Bug Fixes

- (MODPYTHON-84) Fixed request.sendfile() bug for symlinked files on Win32.
- (MODPYTHON-122) Fixed configure problem when using bash 3.1.x.
- (MODPYTHON-173) Fixed DbmSession to create db file with mode 0640.

9.4 Changes from version 3.2.7

9.4.1 Security Fix

- (MODPYTHON-135) Fixed possible directory traversal attack in FileSession. The session id is now checked to ensure it only contains valid characters. This check is performed for all sessions derived from the BaseSession class.

9.5 Changes from version 3.1.4

9.5.1 New Features

- New apache.register_cleanup() method.
- New apache.exists_config_define() method.
- New file-based session manager class.
- Session cookie name can be specified.
- The maximum number of mutexes mod_python uses for session locking can now be specifed at compile time using configure –with-max-locks.
- New a version attribute in mod_python module.
- New test handler testhandler.py has been added.

9.5.2 Improvements

- Autoreload of a module using apache.import_module() now works if modification time for the module is different from the file. Previously, the module was only reloaded if the the modification time of the file was more recent. This allows for a more graceful reload if a file with an older modification time needs to be restored from backup.
- Fixed the publisher traversal security issue
- Objects hierarchy a la CherryPy can now be published.
- mod_python.c now logs reason for a 500 error
- Calls to PyErr_Print in mod_python.c are now followed by fflush()
- Using an empty value with PythonOption will unset a PythonOption key.
- req.path_info is now a read/write member.
- Improvements to FieldStorage allow uploading of large files. Uploaded files are now streamed to disk, not to memory.
- Path to flex is now discovered at configuration time or can be specifed using configure –with-flex=/path/to/flex.
- sys.argv is now initialized to ["mod_python"] so that modules like numarray and pychart can work properly.

9.5.3 Bug Fixes

- Fixed memory leak which resulted from circular references starting from the request object.
- Fixed memory leak resulting from multiple PythonOption directives.
- Fixed Multiple/redundant interpreter creation problem.
- Cookie attributes with attribute names prefixed with $ are now ignored. See Section 4.7 for more information.
- Bug in setting up of config_dir from Handler directives fixed.
- mod_python.publisher will now support modules with the same name but in different directories
- Fixed continual reloading of modules problem
- Fixed big marshalled cookies error.
- Fixed mod_python.publisher extension handling
- mod_python.publisher default index file traversal
- mod_python.publisher loading wrong module and giving no warning/error
- apply_fs_data() now works with "new style" objects
- File descriptor fd closed after ap_send_fd() in req_sendfile()
- Bug in mem_cleanup in MemorySession fixed.
- Fixed bug in _apache._global_lock() which could cause a segfault if the lock index parameter is greater number of mutexes created at mod_python startup.
- Fixed bug where local_ip and local_host in connection object were returning remote_ip and remote_host instead
- Fixed install_dso Makefile rule so it only installs the dso, not the python files
- Potential deadlock in psp cache handling fixed
- Fixed bug where sessions are used outside <Directory> directive.
- Fixed compile problem on IRIX. ln -s requires both TARGET and LINK_NAME on IRIX. ie. ln -s TARGET LINK_NAME
- Fixed ./configure problem on SuSE Linux 9.2 (x86-64). Python libraries are in lib64/ for this platform.
- Fixed req.sendfile() problem where sendfile(filename) sends the incorrect number of bytes when filename is a symlink.
- Fixed problem where util.FieldStorage was not correctly checking the mime types of POSTed entities
- Fixed conn.local_addr and conn.remote_addr for a better IPv6 support.
- Fixed psp_parser.l to properly escape backslash-n, backslash-t and backslash-r character sequences.
- Fixed segfault bug when accessing some request object members (allowed_methods, allowed_xmethods, content_languages) and some server object members (names, wild_names).
- Fixed request.add_handler() segfault bug when adding a handler to an empty handler list.
- Fixed PythonAutoReload directive so that AutoReload can be turned off.
- Fixed connection object read() bug on FreeBSD.
- Fixed potential buffer corruption bug in connection object read().

9.6 Changes from 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

CHAPTER

TEN

HISTORY AND LICENSE

10.1 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)).

Nsapy itself was based on an original concept and first code by Aaron Watters from "Internet Programming with Python" by Aaron Watters, Guido Van Rossum and James C. Ahlstrom, ISBN 1-55851-484-8.

Without Aaron's inspiration, there would be no mod_python. Quoting from the Httpdapy README file:

```
Although Nsapy only worked with Netscape servers, it was very
generic in its design and was based on some brilliant ideas that
weren't necessarily Netscape specific. Its design is a combination
of extensibility, simplicity and efficiency that takes advantage of
many of the key benefits of Python and is totally in the spirit of
Python.
```

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.

10.2 License

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

> http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

Apache License:

```
                              Apache License
                        Version 2.0, January 2004
                     http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

   "License" shall mean the terms and conditions for use, reproduction,
   and distribution as defined by Sections 1 through 9 of this document.

   "Licensor" shall mean the copyright owner or entity authorized by
   the copyright owner that is granting the License.

   "Legal Entity" shall mean the union of the acting entity and all
   other entities that control, are controlled by, or are under common
   control with that entity. For the purposes of this definition,
   "control" means (i) the power, direct or indirect, to cause the
   direction or management of such entity, whether by contract or
   otherwise, or (ii) ownership of fifty percent (50%) or more of the
   outstanding shares, or (iii) beneficial ownership of such entity.

   "You" (or "Your") shall mean an individual or Legal Entity
   exercising permissions granted by this License.

   "Source" form shall mean the preferred form for making modifications,
   including but not limited to software source code, documentation
```

```
   source, and configuration files.

   "Object" form shall mean any form resulting from mechanical
   transformation or translation of a Source form, including but
   not limited to compiled object code, generated documentation,
   and conversions to other media types.

   "Work" shall mean the work of authorship, whether in Source or
   Object form, made available under the License, as indicated by a
   copyright notice that is included in or attached to the work
   (an example is provided in the Appendix below).

   "Derivative Works" shall mean any work, whether in Source or Object
   form, that is based on (or derived from) the Work and for which the
   editorial revisions, annotations, elaborations, or other modifications
   represent, as a whole, an original work of authorship. For the purposes
   of this License, Derivative Works shall not include works that remain
   separable from, or merely link (or bind by name) to the interfaces of,
   the Work and Derivative Works thereof.

   "Contribution" shall mean any work of authorship, including
   the original version of the Work and any modifications or additions
   to that Work or Derivative Works thereof, that is intentionally
   submitted to Licensor for inclusion in the Work by the copyright owner
   or by an individual or Legal Entity authorized to submit on behalf of
   the copyright owner. For the purposes of this definition, "submitted"
   means any form of electronic, verbal, or written communication sent
   to the Licensor or its representatives, including but not limited to
   communication on electronic mailing lists, source code control systems,
   and issue tracking systems that are managed by, or on behalf of, the
   Licensor for the purpose of discussing and improving the Work, but
   excluding communication that is conspicuously marked or otherwise
   designated in writing by the copyright owner as "Not a Contribution."

   "Contributor" shall mean Licensor and any individual or Legal Entity
   on behalf of whom a Contribution has been received by Licensor and
   subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   copyright license to reproduce, prepare Derivative Works of,
   publicly display, publicly perform, sublicense, and distribute the
   Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   (except as stated in this section) patent license to make, have made,
   use, offer to sell, sell, import, and otherwise transfer the Work,
   where such license applies only to those patent claims licensable
   by such Contributor that are necessarily infringed by their
   Contribution(s) alone or by combination of their Contribution(s)
   with the Work to which such Contribution(s) was submitted. If You
   institute patent litigation against any entity (including a
   cross-claim or counterclaim in a lawsuit) alleging that the Work
   or a Contribution incorporated within the Work constitutes direct
   or contributory patent infringement, then any patent licenses
```

```
   granted to You under this License for that Work shall terminate
   as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the
   Work or Derivative Works thereof in any medium, with or without
   modifications, and in Source or Object form, provided that You
   meet the following conditions:

   (a) You must give any other recipients of the Work or
       Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices
       stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works
       that You distribute, all copyright, patent, trademark, and
       attribution notices from the Source form of the Work,
       excluding those notices that do not pertain to any part of
       the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its
       distribution, then any Derivative Works that You distribute must
       include a readable copy of the attribution notices contained
       within such NOTICE file, excluding those notices that do not
       pertain to any part of the Derivative Works, in at least one
       of the following places: within a NOTICE text file distributed
       as part of the Derivative Works; within the Source form or
       documentation, if provided along with the Derivative Works; or,
       within a display generated by the Derivative Works, if and
       wherever such third-party notices normally appear. The contents
       of the NOTICE file are for informational purposes only and
       do not modify the License. You may add Your own attribution
       notices within Derivative Works that You distribute, alongside
       or as an addendum to the NOTICE text from the Work, provided
       that such additional attribution notices cannot be construed
       as modifying the License.

   You may add Your own copyright statement to Your modifications and
   may provide additional or different license terms and conditions
   for use, reproduction, or distribution of Your modifications, or
   for any such Derivative Works as a whole, provided Your use,
   reproduction, and distribution of the Work otherwise complies with
   the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise,
   any Contribution intentionally submitted for inclusion in the Work
   by You to the Licensor shall be under the terms and conditions of
   this License, without any additional terms or conditions.
   Notwithstanding the above, nothing herein shall supersede or modify
   the terms of any separate license agreement you may have executed
   with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade
   names, trademarks, service marks, or product names of the Licensor,
   except as required for reasonable and customary use in describing the
   origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or
```

```
   agreed to in writing, Licensor provides the Work (and each
   Contributor provides its Contributions) on an "AS IS" BASIS,
   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
   implied, including, without limitation, any warranties or conditions
   of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A
   PARTICULAR PURPOSE. You are solely responsible for determining the
   appropriateness of using or redistributing the Work and assume any
   risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory,
   whether in tort (including negligence), contract, or otherwise,
   unless required by applicable law (such as deliberate and grossly
   negligent acts) or agreed to in writing, shall any Contributor be
   liable to You for damages, including any direct, indirect, special,
   incidental, or consequential damages of any character arising as a
   result of this License or out of the use or inability to use the
   Work (including but not limited to damages for loss of goodwill,
   work stoppage, computer failure or malfunction, or any and all
   other commercial damages or losses), even if such Contributor
   has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing
   the Work or Derivative Works thereof, You may choose to offer,
   and charge a fee for, acceptance of support, warranty, indemnity,
   or other liability obligations and/or rights consistent with this
   License. However, in accepting such obligations, You may act only
   on Your own behalf and on Your sole responsibility, not on behalf
   of any other Contributor, and only if You agree to indemnify,
   defend, and hold each Contributor harmless for any liability
   incurred by, or claims asserted against, such Contributor by reason
   of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

   To apply the Apache License to your work, attach the following
   boilerplate notice, with the fields enclosed by brackets "[]"
   replaced with your own identifying information. (Don't include
   the brackets!)  The text should be enclosed in the appropriate
   comment syntax for the file format. We also recommend that a
   file or class name and description of purpose be included on the
   same "printed page" as the copyright notice for easier
   identification within third-party archives.

Copyright [yyyy] [name of copyright owner]

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
```

CHAPTER

ELEVEN

ABOUT THESE DOCUMENTS

These documents are generated from reStructuredText sources by Sphinx, a document processor specifically written for the Python documentation.

CHAPTER

TWELVE

COPYRIGHT

See *History and License* for complete license and permissions information.

APPENDIX

A

ABOUT THESE DOCUMENTS

These documents are generated from reStructuredText sources by Sphinx, a document processor specifically written for the Python documentation.

APPENDIX

B

HISTORY AND LICENSE

B.1 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)).

Nsapy itself was based on an original concept and first code by Aaron Watters from "Internet Programming with Python" by Aaron Watters, Guido Van Rossum and James C. Ahlstrom, ISBN 1-55851-484-8.

Without Aaron's inspiration, there would be no mod_python. Quoting from the Httpdapy README file:

```
Although Nsapy only worked with Netscape servers, it was very
generic in its design and was based on some brilliant ideas that
weren't necessarily Netscape specific. Its design is a combination
of extensibility, simplicity and efficiency that takes advantage of
many of the key benefits of Python and is totally in the spirit of
Python.
```

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.

B.2 License

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

> http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

Apache License:

```
                              Apache License
                        Version 2.0, January 2004
                     http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

   "License" shall mean the terms and conditions for use, reproduction,
   and distribution as defined by Sections 1 through 9 of this document.

   "Licensor" shall mean the copyright owner or entity authorized by
   the copyright owner that is granting the License.

   "Legal Entity" shall mean the union of the acting entity and all
   other entities that control, are controlled by, or are under common
   control with that entity. For the purposes of this definition,
   "control" means (i) the power, direct or indirect, to cause the
   direction or management of such entity, whether by contract or
   otherwise, or (ii) ownership of fifty percent (50%) or more of the
   outstanding shares, or (iii) beneficial ownership of such entity.

   "You" (or "Your") shall mean an individual or Legal Entity
   exercising permissions granted by this License.

   "Source" form shall mean the preferred form for making modifications,
   including but not limited to software source code, documentation
```

```
   source, and configuration files.

   "Object" form shall mean any form resulting from mechanical
   transformation or translation of a Source form, including but
   not limited to compiled object code, generated documentation,
   and conversions to other media types.

   "Work" shall mean the work of authorship, whether in Source or
   Object form, made available under the License, as indicated by a
   copyright notice that is included in or attached to the work
   (an example is provided in the Appendix below).

   "Derivative Works" shall mean any work, whether in Source or Object
   form, that is based on (or derived from) the Work and for which the
   editorial revisions, annotations, elaborations, or other modifications
   represent, as a whole, an original work of authorship. For the purposes
   of this License, Derivative Works shall not include works that remain
   separable from, or merely link (or bind by name) to the interfaces of,
   the Work and Derivative Works thereof.

   "Contribution" shall mean any work of authorship, including
   the original version of the Work and any modifications or additions
   to that Work or Derivative Works thereof, that is intentionally
   submitted to Licensor for inclusion in the Work by the copyright owner
   or by an individual or Legal Entity authorized to submit on behalf of
   the copyright owner. For the purposes of this definition, "submitted"
   means any form of electronic, verbal, or written communication sent
   to the Licensor or its representatives, including but not limited to
   communication on electronic mailing lists, source code control systems,
   and issue tracking systems that are managed by, or on behalf of, the
   Licensor for the purpose of discussing and improving the Work, but
   excluding communication that is conspicuously marked or otherwise
   designated in writing by the copyright owner as "Not a Contribution."

   "Contributor" shall mean Licensor and any individual or Legal Entity
   on behalf of whom a Contribution has been received by Licensor and
   subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   copyright license to reproduce, prepare Derivative Works of,
   publicly display, publicly perform, sublicense, and distribute the
   Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   (except as stated in this section) patent license to make, have made,
   use, offer to sell, sell, import, and otherwise transfer the Work,
   where such license applies only to those patent claims licensable
   by such Contributor that are necessarily infringed by their
   Contribution(s) alone or by combination of their Contribution(s)
   with the Work to which such Contribution(s) was submitted. If You
   institute patent litigation against any entity (including a
   cross-claim or counterclaim in a lawsuit) alleging that the Work
   or a Contribution incorporated within the Work constitutes direct
   or contributory patent infringement, then any patent licenses
```

```
   granted to You under this License for that Work shall terminate
   as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the
   Work or Derivative Works thereof in any medium, with or without
   modifications, and in Source or Object form, provided that You
   meet the following conditions:

   (a) You must give any other recipients of the Work or
       Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices
       stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works
       that You distribute, all copyright, patent, trademark, and
       attribution notices from the Source form of the Work,
       excluding those notices that do not pertain to any part of
       the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its
       distribution, then any Derivative Works that You distribute must
       include a readable copy of the attribution notices contained
       within such NOTICE file, excluding those notices that do not
       pertain to any part of the Derivative Works, in at least one
       of the following places: within a NOTICE text file distributed
       as part of the Derivative Works; within the Source form or
       documentation, if provided along with the Derivative Works; or,
       within a display generated by the Derivative Works, if and
       wherever such third-party notices normally appear. The contents
       of the NOTICE file are for informational purposes only and
       do not modify the License. You may add Your own attribution
       notices within Derivative Works that You distribute, alongside
       or as an addendum to the NOTICE text from the Work, provided
       that such additional attribution notices cannot be construed
       as modifying the License.

   You may add Your own copyright statement to Your modifications and
   may provide additional or different license terms and conditions
   for use, reproduction, or distribution of Your modifications, or
   for any such Derivative Works as a whole, provided Your use,
   reproduction, and distribution of the Work otherwise complies with
   the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise,
   any Contribution intentionally submitted for inclusion in the Work
   by You to the Licensor shall be under the terms and conditions of
   this License, without any additional terms or conditions.
   Notwithstanding the above, nothing herein shall supersede or modify
   the terms of any separate license agreement you may have executed
   with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade
   names, trademarks, service marks, or product names of the Licensor,
   except as required for reasonable and customary use in describing the
   origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or
```

```
   agreed to in writing, Licensor provides the Work (and each
   Contributor provides its Contributions) on an "AS IS" BASIS,
   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
   implied, including, without limitation, any warranties or conditions
   of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A
   PARTICULAR PURPOSE. You are solely responsible for determining the
   appropriateness of using or redistributing the Work and assume any
   risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory,
   whether in tort (including negligence), contract, or otherwise,
   unless required by applicable law (such as deliberate and grossly
   negligent acts) or agreed to in writing, shall any Contributor be
   liable to You for damages, including any direct, indirect, special,
   incidental, or consequential damages of any character arising as a
   result of this License or out of the use or inability to use the
   Work (including but not limited to damages for loss of goodwill,
   work stoppage, computer failure or malfunction, or any and all
   other commercial damages or losses), even if such Contributor
   has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing
   the Work or Derivative Works thereof, You may choose to offer,
   and charge a fee for, acceptance of support, warranty, indemnity,
   or other liability obligations and/or rights consistent with this
   License. However, in accepting such obligations, You may act only
   on Your own behalf and on Your sole responsibility, not on behalf
   of any other Contributor, and only if You agree to indemnify,
   defend, and hold each Contributor harmless for any liability
   incurred by, or claims asserted against, such Contributor by reason
   of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

   To apply the Apache License to your work, attach the following
   boilerplate notice, with the fields enclosed by brackets "[]"
   replaced with your own identifying information. (Don't include
   the brackets!)  The text should be enclosed in the appropriate
   comment syntax for the file format. We also recommend that a
   file or class name and description of purpose be included on the
   same "printed page" as the copyright notice for easier
   identification within third-party archives.

Copyright [yyyy] [name of copyright owner]

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
```

APPENDIX

C

COPYRIGHT

See *History and License* for complete license and permissions information.

PYTHON MODULE INDEX

INDEX

D

E

F

G

H

I

Q

R

S

T

U

V

W