mod_snake: Flexible Apache Modules in Python

by Jon Travis

1 Introduction

In early May of 2000, a solution for truly integrating Python written modules with the Apache webserver was not available. There existed two main projects which provided some interaction between the two, PyApache and Httpdapy. PyApache simply acted as a CGI accelerator by embedding the Python interpreter within the webserver process. Httpdapy took the process one step further, and allowed various Python hooks to be called during processing phases. Due to the lack of integration capabilities in these projects, mod_snake was born.

2 Design Goals

The initial design goals of mod_snake were:

* to give module designers as much power in Python with respect to Apache manipulation as they would have in C,

* to use Object Oriented Programming (OOP) practices in order to decrease the initial learning curve and maintain coherence for long-time Python programmers,

* to provide a framework that would be easily back-portable to Apache 1.3, and

* to abstract enough of the Apache internals that the majority of Python modules could plug seamlessly into either Apache 1.3 or 2.0 servers.

The reasoning behind these goals is fairly basic: Python programmers should be able to write modules using existing Python libraries, and achieve the same functionality that they would have when writing Apache modules in C. Programming in Apache for Python should not have big hurdles, and should be as comfortable as possible.

3 Why Use mod_snake?

There are a number of reasons for web developers to use mod_snake:

- It allows for easy migration of existing Python CGI modules into an accelerated handler, which can then be extended for more functionality.
- By using mod_snake plugin modules, many limitations of server interaction with CGIs, such as URI translation are lifted.
- The modular nature of Python makes it very easy to reuse code.
- Reference counting reduces the amount of programming 'cleanup' that needs to be done, which commonly clutters C code.
- mod_snake abstracts much of the underlying Apache interfaces, and therefore makes it trivial to port modules to future revisions of Apache.

3.1 "But isn't this the same as mod_python?"

Approximately the same time mod_snake was released, there was a similar module released, called mod_python. One of the most commonly asked questions is, "isn't this the same as mod_python?" The short answer is no.

There are a variety of differences between mod_snake and mod_python:

* mod_python is able to run on Windows based platforms
* Both mod_snake and mod_python can accelerate Python written CGI
* Both mod_snake and mod_python allow access to the Apache API and structures
* mod_snake allows for dynamic directive creation
* mod_snake works with both Apache 1.3 and 2.0
* mod_snake provides the power of server-based and directory-based configuration
* mod_snake comes with a module which allows embedding of Python in HTML pages

Analysis of the various features that mod_snake and mod_python provide will need to be done on an independent basis by web developers to determine which provides the exact functionality desired.

4 Installation and Configuration

4.1 Download and Installation

The installation of mod_snake is a very straightforward and simple process for either the Apache 1.3 or 2.0 webserver. The latest release of mod_snake can be obtained from its home site at: http://modsnake.sourceforge.net. For purposes of illustration, the Apache and mod_snake source will be unpacked in **/usr/local/src** and installed in **/usr/local/apache.**

4.1.1 Apache 1.3 Installation

First, download and unpack the latest release of Apache 1.3, available from http://www.apache.org.

* To setup the include files which mod_snake will need to build, configure a default Apache installation by typing:

  cd /usr/local/src/apache_1.3.x
  ./configure

* In Apache 1.3, mod_snake is currently only built as a static module. To configure mod_snake:

  cd /usr/local/src/mod_snake-x.x.x
  ./configure --with-apache=/usr/local/src/apache_1.3.x

* Build and install mod_snake into the Apache source directories:

  make install

* Finally, configure and install Apache with the new mod_snake module:

  cd /usr/local/src/apache_1.3.x
  ./configure --activate-module=src/modules/mod_snake/libmod_snake.a
  make install

4.1.2 Apache 2.0 Installation

First, download and unpack the latest release of Apache 2.0, available from http://www.apache.org.

* Configure Apache to enable DSO support, and install it:

  cd /usr/local/src/apache_2.0.x
  ./configure --enable-so
  make install

* To configure and build mod_snake:

  cd /usr/local/src/mod_snake-x.x.x
  ./configure --with-apache=/usr/local/src/apache_2.0.x
  make

* If everything succeeded, there will be a new library called libmod_snake.so within the mod_snake-x.x.x/src directory. You may leave this file here, or copy it to a more conventional location, such as the installed Apache libexec directory.

4.2 Apache Server Configuration

To enable the mod_snake module for your Apache installation, edit the server configuration file in: /usr/local/apache_x.x.x/conf/httpd.conf

* For Apache 1.3, no LoadModule or AddModule directives will need to be added, however, for Apache 2.0, after the Listen directives, add the line:

    LoadModule libexec/libmod_snake.so

* Add the SnakeLib directory into the mod_snake search path, by adding at the end of the file:

    SnakeModuleDir /usr/local/apache/snake_lib

You will need to copy the SnakeLib files from /usr/local/src/mod_snake-x.x.x/snake_lib to /usr/local/apache/snake_lib. These files are modules which can be used for accelerating Python CGIs, embedding Python in HTML, and other useful features.

5 Inner Workings

5.1 A Quick Module Primer

Apache gains the majority of its power through plugin modules. These modules can do a variety of things such as translate URIs, perform user validation, establish MIME types, generate content, log transactions, and much more.

From the time that the configuration file is read, to after a client has closed its connection and the server shuts down, Apache breaks down the steps that it takes into phases. These phases go in order from post_config, open_logs, post_read_request, translate_name, header_parser, access_checker, check_user_id, auth_checker, type_checker, fixups, content_handler, log_transaction. While this list is not comprehensive, it gives a small overview of the order that the Apache phases go in.

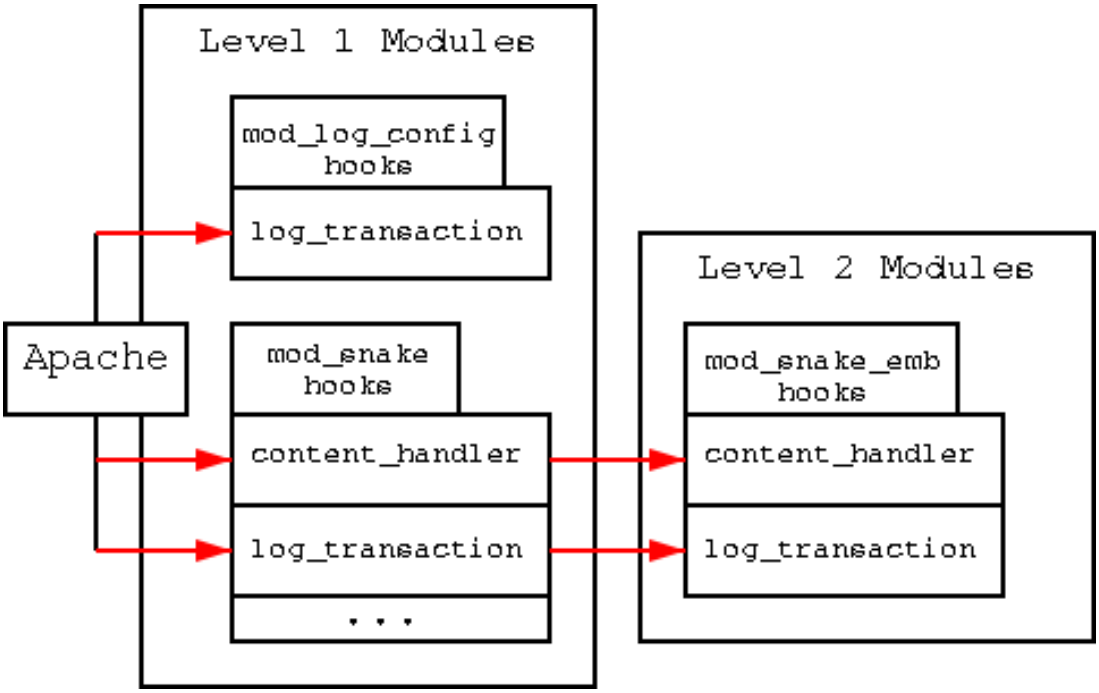
Apache speaks to the plugin modules by way of 'hooks'. These hooks are simply function calls that Apache can call during the cooresponding phase. Modules register which hooks they are using with Apache, and when the appropriate phase occurs in the Apache server, the module's hook function will be called.

For instance, if a web developer wishes to have transactions logged in XML format to a file, he may choose to write a simple module that hooks into the log_transaction Apache phase. When Apache reaches the phase for logging transactions, the hook in the developer's module will be called with information about the request.

Another example might be that the web developer wants to limit access to a certain region of a site, based on a username and password verified against an internal customer database. In this case, the module will be slightly more sophisticated, because it may need to hook into multiple Apache phases, such as check_userid, and auth_checker.

5.2 Second Level Modules

Modules which Apache can make direct calls to can be thought of as first level modules; they are loaded directly by the Apache process, and need no intermediate support. Dynamically Shared Object (DSO) modules are really first level modules, as mod_so only facilitates their loading into the main server.



Mod_Snake is a first level module, as it is loaded directly from the server, and the Apache process calls functions within mod_snake directly. However, mod_snake also acts as a proxy for second level modules. It binds to all of the hooks that Apache makes available and translates these calls to Python calls. This makes all of the Python modules second level modules.

5.3 Pros

There are many pros to having mod_snake directly manage all of the loaded Python modules:

* It provides a centralized location for storing loaded module information
* Communication between loaded Python modules is more easily facilitated
* Functionality common to all Python modules can be included within mod_snake and made available as an imported module
* It is currently the only way to have Python loaded modules

5.4 Cons

Unfortunately, there is also a major con with respect to mod_snake managing the Python modules:

* The Apache API does not make it simple for proxies of second level modules. Therefore much of the work that Apache does before calling a module's phase handler is duplicated in mod_snake.

A good example of this is the process_connection hook. Apache calls this hook when it has accepted a connection on one of its ports. Apache loops through all of the modules binding to this hook until one returns OK. When mod_snake's process_connection hook is called, it must do the same thing ... loop through all of its loaded modules until one returns OK. Though this processing is done very quickly, it still slightly increases the overhead of mod_snake.

6 Baby Steps: Using Python In CGIs

6.1 CGI Introduction

One of the original ways of creating dynamic content on the web came in the form of Common Gateway Interface (CGI) scripts. These scripts generally involve a fork of the server, and an exec of the CGI program or script. When run, the CGI's output is sent to the remote client. CGIs can be written in any number of languages as long as the operating system knows how to execute it.

The traditional means for using Python as a CGI involved writing a script that could be invoked directly from the shell. A small Python CGI might look like:

```
#!/usr/bin/python

print 'Content-type: text/plain\n\n'
print 'Hello world, I'm a Python CGI'
```

This example is not very useful, but does show some of the basic concepts behind dynamic content generation. Namely that the first part of the output must be HTTP headers, and the rest is content, sent directly to the browser.

6.2 A Simple Example

One of the most popular items on personal homepages, second only to the 'Under Construction' logo, is the hit counter. By using a CGI script, the developer can generate content for their page and update the counter at the same time. A start at the script might look like:

joeys_page.py:

```
#!/usr/bin/python
import hitcounter

HITFILE = 'mypages.hits'

print 'Content-type: text/html\n\n'
print """
<HTML><BODY>
  Hello world!  My name is Joey Buttafooko, and this is my page!  It has
  been accessed %d times!  Neat!</BODY></HTML>""" % hitcounter.do_hit(HITFILE)
```

And an associated module which can load the hits from a file, increment the counter, and write out the new value may look like:

hitcounter.py:

```
def do_hit(filename):
        try:    filefd = open(filename, 'r+')
        except: filefd = open(filename, 'w+')

        try:    hits = int(filefd.read())
        except: hits = 0

        hits = hits + 1
        filefd.seek(0)
        filefd.write('%d' % hits)
        return hits
```

This script is now complete, and ready to run. In order to invoke it, the hitcounter.py must be placed in the /usr/local/apache/snake_lib directory, and the joeys_page.py file must be placed in a snakecgi enabled directory. This can be done with mod_snake by editing the httpd.conf

file in /usr/local/apache/conf. The lines at the end of the file should read:

```
SnakeModule mod_snake_cgi.SnakeCGI
Alias /snakecgi/  "/usr/local/apache/snakecgi/"
<Directory "/usr/local/apache/snakecgi">
        SetHandler snakecgi
</Directory>
```

The SnakeModule line tells mod_snake to load the SnakeCGI class from the mod_snake_cgi module. This module can be found in the snake_lib directory which was copied over to /usr/local/apache/snake_lib. The other portion of the configuration tells Apache that everything within the /usr/local/apache/snakecgi directory is to be interpreted as a snakecgi script. The string, 'snakecgi', is a magic constant within mod_snake_cgi.SnakeCGI, similar to 'cgi-script' within the mod_cgi handler.

6.3 Joey's Dilemma

The author of the previous CGI, Joey Buttafooko, gained a lot of popularity on the web, shortly after he put up his page. In fact, he was rated second on list of top five web sites! The first person on the list was a woman by the name of Amy Phishah. This puzzled him greatly, because he knew that he had to be getting more hits than her. Perplexed, he decided to investigate the quality of his hit counter. He determined that he should stress test his page to ensure that every hit was counted.

Fortunately, Apache comes with a useful utility called Apache bench. This can usually be found in the bin directory, along with httpd. Joey decided to simulate numerous concurrent hits to make sure the counter was reporting the correct values. He ran Apache bench for 1000 requests per trial, and a concurrency of 5. Here are the values of the web counter after each test: 13273, 14230, 15234, and 16165. This yielded 957, 1004, and 931 hits, respectively. Joey was shocked! He was getting robbed out of a lot of hits!

After looking back through his CGI, Joey figured out what was going wrong. With so many hits coming in, two requests may be reading or writing to the file at the same time, colliding with each other! He then rewrote it to take advantage of POSIX file locking:

hitcounter.py:

```
import posixfile

def do_hit(filename):
        try:    filefd = posixfile.open(filename, 'r+')
        except: filefd = posixfile.open(filename, 'w+')

        filefd.lock('w|')
        try:    hits = int(filefd.read())
        except: hits = 0

        hits = hits + 1
        filefd.seek(0)
        filefd.write('%d' % hits)
        filefd.lock('u')
        return hits
```

After adding this new locking functionality, Joey tested his counter in the same way, and came up with the following hit counts: 24141, 25145, 26149, and 27153. This yielded 1004, 1004, and 1004 hits, respectively. Though this number was slightly off from the 1000 hits he expected, he was content that the numbers were consistent and chalked up the mistake to an off-by-one bug in the Apache bench utility.

6.4 Expansion via cgi.py

The Python distribution includes a module called cgi.py. This module allows CGI authors to process query strings, forms, and escaped text. This module is useful when doing any kind of web work, whether it be CGI, embedding, or a regular module.

7 Using Embedded Python

In the previous section, a web developer named Joey starting using Python in a CGI environment for generating dynamic content. He quickly figured out that having a hitcounter alone is not enough to satisfy a visitor's curiosity; he needed more content. As he added more and more pages to his site, he became a bit upset about always needing to add his CGIs to a specific directory. He needed a new way to generate content and keep his counters.

Fortunately, mod_snake's embedding module allowed him to be liberated from the CGI placement and put actual Python code directory into HTML files.

7.1 Configuring mod_snake_emb

The snake_lib directory which was copied into /usr/local/apache/snake_lib contains a module for adding embedded Python into HTML. To enable this module in the server, it needs to be loaded via a SnakeModule directive. In /usr/local/apache/conf/httpd.conf, the following line should appear:

SnakeModule mod_snake_emb.SnakeEMB

This will load the module for processing embedded Python, but will not enable it for any files. mod_snake_emb enables use of several directives within the httpd.conf file and other .htaccess files. These directives are:

* **SnakeEMB** <on|off> - Enables or disables embedded Python processing for a given directory
* **SnakeEMBCache** <on|off> - Turns caching of embedded Python files on or off.
* **SnakeEMBErrSend** <on|off> - Directs errors which occur during embedded processing to the remote client, instead of a sending a generic error message.
* **SnakeEMBLogFile** <file> - Specifies a file to write errors to when errors occur during embedded processing. If this directive is absent, log information will be written to the ErrorLog

Since Joey plans on using embedded Python processing in nearly every page on his site, he enables processing for all files with the following httpd.conf lines:

SnakeEMB on
SnakeEMBCache on

7.2 Embedding Delimiters

mod_snake_emb allows developers to add Python code in two different ways. These token delimiters were originally stolen from the EmbPerl project.

* [- code -] - Within the bracket-minus delimiters, the code is executed in the same fashion that ordinary Python code is within the exec() call. No output from contained statements is sent to the remote client.

* [+ code +] - Within the bracket-plus delimiters, code is evaluated via the Python eval() call. The result of the contained expression is sent to the remote client.

A very small embedded Python example might display the current time:

time_show.html:

```
[-import time-]

The current time is: [+time.ctime(time.time())+]
```

7.3 An Embedded Hit Counter

Armed with this new knowledge, Joey decides to use his existing hitcounter.py module which he stowed in his snake_lib directory, and start adding it to his pages. His main page looks like this:

joeys_page.html:

```
[-
import hitcounter
HITFILE = "mypages.hits"
-]
<HTML><BODY>
    Hi, I'm Joey Buttafooko! Welcome to my page. Here are the current places
    you can visit. (UNDER CONSTRUCTION)
    <BR>
    This page has been accessed: <B>[+hitcounter.do_hits(HITFILE)+]</B> times!
</BODY></HTML>
```

When this page gets accessed, all of the Python code is executed, and the resultant HTML that is sent to the client contains the new hit count.

7.4 Page Generation With HTMLgen

A great way to develop a consistent, dynamic webpage, without ever touching an HTML tag is HTMLgen. It is available from http://starship.python.net/crew/friedrich/HTMLgen/html/main.html. The HTMLgen package can either be unpacked in the **/usr/local/apache/snake_lib** directory, or in **/usr/local/apache/HTMLgen**. In the latter situation, an additional SnakeModuleDir directive will need to be added to the httpd.conf file.

A good way to make a website have a consistant look is to centralize creation functions and expand upon them when creating more specific files. A simple Python script to create a page might start out as:

amy_site.py:

```
from HTMLgen import *
from HTMLcolors import *

class AmyNavBar(Container):
    nav_items = (('[Main]', 'index.html'),
                 ('[News]', 'news.html'))
    def __init__(self, current_file):
        Container.__init__(self)
        for title, file in self.nav_items:
            if current_page == file:
                self.append(title)
            else:
                self.append(Href(file, title))

class AmyPage(SimpleDocument):
    def __init__(self, title, file):
```

```
        SimpleDocument.__init__(self, title=title, bgcolor=WHITE)
        self.append(Heading(1, title))
        self.append(AmyNavBar(file), BR(), BR())
```

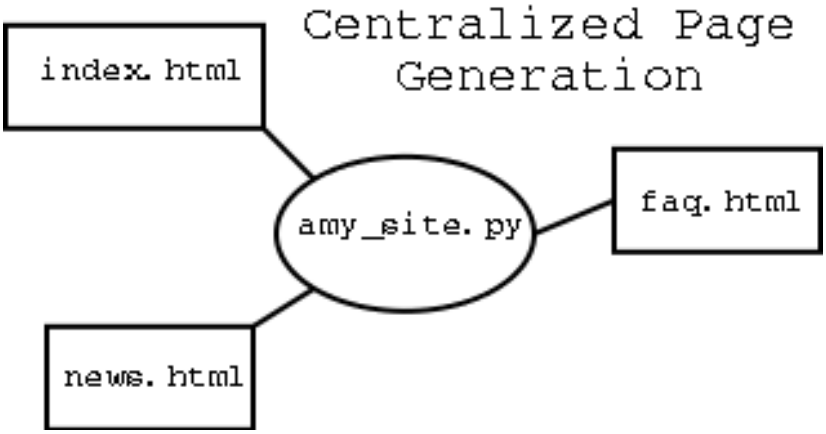
index.html:

```
[-
import amy_site

page = amy_site.AmyPage('Main Home Page', 'index.html')
page.append('Welcome to my page!  It is currently under constuction, so watch out!')
-]

[+'%s' % page+]
```

These few lines may appear to do very little, but in reality these basic tools can create powerful websites with a consistant look and feel. Not only can Amy's website be created much quicker, but she no longer needs to worry about any obscure HTML syntax or bulky headers. In addition, she can add more pages to her navigation bar, or change the entire layout of her site, by simply editing one file.



7.4 More Feats With mod_snake_emb

The mod_snake_emb module allows for a variety of great uses:

* XML is a great way to store information in an easily parsable format, and XSL is an easy and powerful way to render it. ForeThought (www.forethought.com) distributes an XSL module which can be used in embedded Python with only a few l lines of code to create dynamic pages that use style sheets.

* Many news sites keep their stories in mySQL, Postgres, or similar databases. Since Python modules are available for accessing these databases, news references can be made directly from HTML documents

8 Apache + Python Integration = The Big Tamale!

Up until this section, mod_snake has had little to offer for the power programmer. It was shown to be a useful aid for creating dynamic content, but what can it do for site administrators, analyzers, or inebriated football players?

8.1 Hello World - Small Tamales

This first example shows the basic layout of Python modules which can plug into mod_snake. It is a very simple one, which only echos 'Hello World' to the client.

```
import mod_snake

class Hello_World:
    def __init__(self, module):
        module.add_hook('content_handler', self.content_handler)

    def content_handler(self, per_dir, per_svr, req):
        if req.uri == '/hello_world' and req.method == 'GET':
            req.content_encoding = 'text/html'
            req.send_http_header()
            req.rwrite('Hello world!\n')
            return mod_snake.OK

        return mod_snake.DECLINED
```

The breakdown of this module is relatively trivial:

```
class Hello_World:
```

Mod_Snake knows how to access many different 'modules' from one source file. The class name is what differentiates it from other modules within the same source file.

```
def __init__(self, module):
```

The __init__ method is called when an instance of the module is to be created. Passed in is the mod_snake module representation, which can be used for adding hooks and directives.

```
module.add_hook('content_handler', self.content_handler)
```

This line tells mod_snake that the module will be attaching to the 'content_handler' hook. This hook is called to generate content.

```
def content_handler(self, per_dir, per_svr, req):
```

When a request comes in, the content_handler method is called in the Hello_World module object. This method is called with per directory and per server configurations, as well as information about the request itself.

```
if req.uri == '/hello_world' and req.method == 'GET':
```

The content handler must determine whether or not it should generate content for the client. This check against the URI and method ensures we are only generating content when asked.

```
req.content_encoding = 'text/html'
req.send_http_header()
```

The next two lines give Apache information about the content that is about to be sent. This allows Apache to setup appropriate HTTP headers, and then send them.

```
req.rwrite('Hello world!\n')
```

This line does the actual writing of data to the client.

```
return mod_snake.OK
```

Since the Hello_World module has processed the request and sent the headers as well as the content, it should return mod_snake.OK. This tells Apache that the content_handler hook of other modules should not be run.

```
return mod_snake.DECLINED
```

If the Hello_World module did not process the request, then returning DECLINED allows Apache to give other modules a chance to generate content.

To test this module, it must be placed into the snake_lib directory and added into the server's httpd.conf file:

SnakeModule my_first_hello_world.Hello_World

8.2 Customizing Hello_World

Apache allows modules to create per-directory and per-server configuration objects which can be passed back in later. In the hello_world example, it might be convenient to change the name of the hello_world document with a configuration directive. This is all very easy to do with mod_snake. The new, hi-tech hello_world looks looks like this:

```
import mod_snake

class Hello_World:
        def __init__(self, module):
                directives = { "HelloWorldURI" : (mod_snake.RSRC_CONF,
                                                  mod_snake.TAKE1,
                                                  self.cmd_HelloWorldURI) }
                module.add_directives(directives)
                module.add_hook('create_svr_config', self.create_svr_config)
                module.add_hook('content_handler', self.content_handler)

        def create_svr_config(self, svr):
                return { "URI" : "/hello_world" }

        def cmd_HelloWorldURI(self, per_dir, per_svr, take1_arg):
                per_svr['URI'] = take1_arg
                return None

        def content_handler(self, per_dir, per_svr, req):
                if r.uri == per_svr['URI'] and r.method == 'GET':
                        r.content_encoding = 'text/html'
                        r.send_http_header()
                        r.rwrite('Hello World!\n')
                        return mod_snake.OK
                return mod_snake.DECLINED
```

The important additions to this file are the addition of directives and the use of per-server configuration objects.

```
directives = { "HelloWorldURI" : (mod_snake.RSRC_CONF,
                                  mod_snake.TAKE1,
                                  self.cmd_HelloWorldURI) }
module.add_directives(directives)
module.add_hook('create_svr_config', self.create_svr_config)
```

The directives dictionary gives information to Apache about what directives in configuration files are accepted. The 3-tuple value gives information about the directive: in which parts of the configuration it is valid, how many arguments it takes, and which function should be called when the directive is encountered.

The create_svr_config hook is added so that when the cmd_HelloWorldURI hook is called, it will have a place to store the data given by the user. In addition, this configuration object will be passed later to the content_handler method.

```
def create_svr_cfg(self, svr):
        return { 'URI' : '/hello_world' }
```

This method is called when Apache encounters a new server configuration, such as a virtual host. The return value can be of any type, as it is not used by Apache or mod_snake at all.

```
def cmd_HelloWorldURI(self, per_dir, per_svr, take1_arg):
        per_svr['URI'] = take1_arg
        return None
```

The cmd_HelloWorldURI method is called whenever the 'HelloWorldURI' directive is found in the main configuration file. It is passed the server configuration in per_svr, and the value in the configuration file in take1_arg. If an error occurs in this function, it can signal it to mod_snake by returning a string error of what the error was.

8.3 An XML Logger

Often the standard Common Log Format (CLF) does not give enough information to site administrators. When the CLF is deviated from, by adding other fields, or changing the format of the data, log analyzers do not know what to expect. By storing logging data in an XML format, sites which frequently change the data they log can continue to use their log analyzers without worrying about spending a lot of time doing conversions. A basic XML logger might be written as:

```
import mod_snake
import cgi

class XMLLog:
        def __init__(self, module):
                directives = { 'XMLLog' : (mod_snake.RSRC_CONF, mod_snake.TAKE1,
                                           self.cmd) }
                module.add_directives(directives)
                module.add_hook('create_svr_config', self.create_svr_config)
                module.add_hook('open_logs', self.open_logs)
                module.add_hook('log_transaction', self.log_transaction)

        def create_svr_config(self, svr):
                return { 'XMLLog' : mod_snake.ap_server_root_relative('logs/log.xml') }

        def cmd_XMLLog(self, per_dir, per_svr, take1_arg):
                per_svr['XMLLog'] = mod_snake.ap_server_root_relative(take1_arg)

        def open_logs(self, per_svr, module):
                per_svr['logfd'] = open(per_svr['XMLLog'], 'a')
                return mod_snake.DECLINED

        def log_transaction(self, per_dir, per_svr, req):
                str ='<ent><remote_ip>%s</remote_ip><request_file>%s</request_file>'
                str =str + '<method>%s</method><bytes_sent>%d</bytes_sent>'
                str = str + '</ent>'
                per_svr['logfd'].write(str % ( req.connection.remote_ip,
                                               cgi.escape(req.filename), req.method,
                                               req.bytes_sent))
                return mod_snake.DECLINED
```

Notice that this module has maintained its simplicity while taking on a large amount of power. When the server starts up, it calls the 'open_logs' hook, which allows modules to open files that they would not ordinarily have permission to do. Later, if the administrator wishes to add more fields to log entries, he can do so without worrying about converting his old data, or breaking his log analyzer.

8.4 Writing New Protocols

Since mod_snake can also run under Apache 2.0, Python modules have access to some of the new features that it provides, such as connection processing and I/O filtering. The process_connection phase allows modules to intervene before any data is written from the server to the client, and before any data is read by the server. This opens up the ability for modules to do a number of interesting things such as SSL encryption, SMTP handling, FTP serving, finger services, etc. Apache 2.0 comes with a module called mod_echo, which takes input from the client and echos it back. The same module can be written as a mod_snake module, showing just how much power Python modules have access to:

```
import mod_snake

def Mod_Echo:
        def __init__(self, module):
                directives = { "ModEcho" :
(mod_snake.RSRC_CONF,mod_snake.FLAG,
                                                self.cmd_ModEcho)}
                module.add_directives(directives)
                module.add_hook('process_connection', self.process_connection)
                module.add_hook('create_svr_config', self.create_svr_config)

        def create_svr_config(self, svr):
                return { 'enabled' : 0 }

        def cmd_ModEcho(self, dir_cfg, svr_cfg, on_or_off):
                svr_cfg['enabled'] = on_or_off

        def process_connection(self, svr_cfg, conn):
                if not svr_cfg['enabled']:
                        return mod_snake.DECLINED

                conn.write("Hello, I'm mod_echo -- Let's chat!\n")
                conn.flush()

                backfire = ''
                while 1:
                        (status, bytes, data) = conn.read(1)
                        if status or not bytes: break

                        backfire = backfire + data
                        if data[0] == '\n':
                                conn.write(backfire)
                                conn.flush()
                                backfire = ''
                return mod_snake.OK
```

9 The Future

The future of mod_snake is very bright. Apache 2.0 opens up a new world of possibilities, especially with respect to content filtering. As Apache grows more powerful, it is hoped that mod_snake will keep up with the changing technology and continue to make it easy for web developers to efficiently develop new modules to suit their needs. In the very near future, some of the goals of the mod_snake project are:

* to start a repository for modules which are written for mod_snake,
* continue bug fixes and enhancements,
* provide a framework and major components for web application development,
* investigate operation with existing Python web projects such as Zope and WebWare,
* and to accelerate and refine the code.

10 More Information

More information about the topics discussed in this paper, can be found.

* in the book, "Writing Apache Modules in Perl and C",
* in the documents section, at http://modsnake.sourceforge.net,
* and from the Apache website, at http://www.apache.org.