

# Web Development with CherryPy

## *Presented to the Calgary Linux Users Group*

Jeffrey Clement

`jclement@calgary.linux.ca`

# Overview

- What is CherryPy?
- Design Philosophy
- Installation
- Classes, Masks, Views, Functions and URLs
- "Hello World!"
- Compiling
- The HTTP Server
- "Hello World! v2"
- Standard Cherry Classes
- Example: Guestbook
- Why CherryPy?
- References

# What is CherryPy?

- A Python based web application development toolkit: libraries, and compiler
- Allows us to build web applications in Python in much the same way we would build an other Python program
- CherryPy applications are compiled into a standalone Python app complete with all your program logic and a web server (really, really easy deployment)
- My favorite web development tool ever! (not that I'm biased :)

# Design Philosophy

- Speed - compiled into one app, runs from memory
- Scalability - simply start more processes on one or more machines and add a load balancer
- Web app is just a normal Python app - cross platform deployment
- Developers can use OOP
- Templating Language (putting HTML in code is annoying) - allows web developers to maintain look and feel without seeing (much) code.

# Noteworthy Features

- Good documentation and examples
- Standard library provides some very helpful things like authentication, database connectivity, etc
- Supports SSL using PyOpenSSL
- Supports XML-RPC calls to server (really easy way to setup an XMLRPC server!)
- CherryPy applications are a simple standalone Python script that runs anywhere Python does
- Simple. Learning and using CherryPy is very quick and intuitive.

# Installation

Debian (I love Debian):

- (root) `apt-get install cherrippy` (it's in testing)
- Then to invoke the compiler just call `cherrippy`. It's in the path.

Other Unix / Win:

- (user) `tar -xzf cherrippy-XXX.tgz`
- Just put your project under the `cherrippy` directory and invoke the compiler with `../cherrippy.py`.

# Classes, Views, Masks and Functions

- CherryPy adds a few new constructs to the Python language:
  - CherryClasses: A Class construct for web apps. Requests are mapped to method calls on CherryClasses
  - Views: Pure python code that can be called by browser. This is just a normal Python method that returns a string of HTML
  - Masks: Method which consists of HTML marked up with templating code. Easier than embedding HTML code in Python code such as we would do in a View. Again can called by browser.
  - Functions: Pure python code that can not be called by browser but can be used by views and masks.

# URL Mapping

- CherryPy takes requests from the browser and translates them to method calls on instantiated CherryClasses
- If the browser sends a request that looks like `/a/b` then CherryPy executes `a.b()`
- If the browser sends a request that looks like `/a` then CherryPy executes the first of `root.a()` or `a.index()`
- If the browsers sends a request like `/` then CherryPy executes `root.index()`
- So it's really easy to make method calls to CherryClasses
- Even more exciting is passing data to the methods. Form data is automatically passed as named arguments to the method. ie) `/a/b?p1=abc&p2=def` invokes `a.b(p1='abc',p2='def')`.
- This means we layout our application into logical sections (CherryClasses) and add methods to that class that behave just like normal python methods. No complicated `request.getField()` syntax or anything like that.



# "Hello World!"

- The default CherryClass should be called "Root". Within a CherryClass the default method is called "index".
- The Root/index method is called if the user doesn't specify a class/method to invoke.

\_\_\_\_\_ Hello.cpy \_\_\_\_\_

```
1 CherryClass Root:
2     mask:
3         def index(self):
4             <h1>Hello World</h1>
```

# "Compiling and Testing"

- To try out our first application we run it through the CherryPy compiler which produces the standalone Python script:  

```
$ cherrypy Hello.cpy
```
- This creates a python script called `HelloServer.py` which contains the webserver and all your content. Simply ship this script to your server machine and run it.
- By default CherryPy apps bind to port 8000. You can override this by creating a config file called "HelloServer.cfg"
- After we startup the server we just point our web browser at "http://127.0.0.1:8000/" and voila!
- We can even take our script and run it through something like py2exe to convert it to a standalone executable for Windows users.

# The HTTP Server

CherryPy "executables" include a fairly complete HTTP server that can be run in several ways:

- single threaded - Server consists of only one process which handles all requests
  - Very fast. Minimal overhead per request.
  - Not appropriate for use on it's own since only one request is processed at a time. A slow web client holds up everyone.
  - Usually run behind Apache using mod\_proxy or mod\_rewrite (easy to do)
  - Can easily run on multiple machines and load balance between them to scale up
- forking - starts a fixed number of forked servers
  - Useful for multiprocessor servers.
  - Should still be run behind Apache. Since we just need one slow request per forked copy still holds up the whole system
- multithreaded - each request is processed in separate thread
  - A bit more overhead per request runs well standalone.
  - One/Many slow client(s) will not hold up others

# "Hello World!" v2

- Python maps URLs to method calls: ie)  
http://server:8000/ invokes Root.index  
http://server:8000/hello invokes Root.hello or ...  
http://server:8000/root/hello invokes Root.hello
- Query parameters are automatically parsed and passed as named parameters to the method call. ie)  
http://server:8000/hello?name=Jeff invokes Root.hello(name='jeff')
- This example collects the users name and then uses the HTML templating to insert that into a webpage.

\_\_\_\_\_ Hello2.cpy \_\_\_\_\_

```
1 CherryClass Root:
2     mask:
3         def index(self):
4             <h1>Hello World</h1>
5             <form action="/hello">
6                 Name: <input type="text" name="name">
7             </form>
8         def hello(self, name):
9             <h1>Hello <py-eval="name"></h1>
```

# Example: Guest Book

- We are going to make three CherryClasses:
  1. MyPage: Base class for all our pages. Includes "header" and "footer" methods to add common look and feel for us.
  2. Root: Has our main homepage which links to our guest book
  3. Guestbook: Contains all the methods related to the guestbook including "view" and "add"
- We have a prebuilt normal Python class for handling the guest book (Our business object) – To keep the code simple the guest book is stored in memory only.
- So we'll have four files: guestbook.py, Root.cpy, Guestbook.cpy and MyPage.cpy
- Oh... And maybe RootServer.cfg if we want to enable SSL or change the server port

# Example: 'guestbook.py'

```
_____ guestbook.py _____
1  import time
2  class Guestbook:
3      def __init__(self):
4          self._list = []
5      def add(self, name, comment):
6          """ Add given name and comment to the guestbook """
7          newRecord = {} # new guestbook entry - it's a dictionary
8          newRecord['name'] = name
9          newRecord['comment'] = comment
10         newRecord['date'] = time.ctime()
11         # add new record onto tail of guestbook
12         self._list.append(newRecord)
13     def list(self):
14         """ Returns array containing all guest book entries """
15         return self._list
```

- This is just a normal python class I'm going to use to build my web app

# Example: "MyPage.cpy"

```
_____ MyPage.cpy _____
1  CherryClass MyPage abstract:  # abstract class isn't instantiated
2  function:
3      def redirect(self, url):
4          # handy little function to send a redirect
5          response.headerMap['status']=302
6          response.headerMap['location']=url
7          return 'Moved <a href="%s">here</a>' % url
8  mask:
9      def header(self, title):
10         <html>
11         <head><title><py-eval="title"></title></head>
12         <body bgcolor="lightblue">
13         <h2><py-eval="title"></h2>
14         <hr>
15     def footer(self):
16         <hr>
17         &copy; 2003 Someone or Something.
18         </body>
19         </html>
```

# Example: "Root.cpy"

```
Root.cpy
1 use MyPage
2
3 CherryClass Root(MyPage):
4     mask:
5         def index(self):
6             <py-eval="self.header('Homepage')">
7             <p>Welcome to my homepage
8             <p>Click <a href="/guestbook/index"> here to view
9                 my guestbook
10            <py-eval="self.footer()">
```

- Remember: the Root class is the default class called if the browser doesn't specifically name one.



# Example: "Guestbook.cpy (1/2)"

```

1  use MyPage
2
3  import guestbook
4  import cgi
5
6  CherryClass Guestbook(MyPage):
7  variable:
8      data = guestbook.Guestbook()    # new instance of guestbook
9  function:
10     def viewGuestbook(self):
11         out = ''
12         for record in self.data.list():
13             out += "<B>%s - %s</b><br>%s<br><br>" % (
14                 cgi.escape(record['name']),      # escape any HTML
15                 cgi.escape(record['date']),
16                 cgi.escape(record['comment']))
17     return out
```

# Example: "Guestbook.cpy (2/2)"

```

1  view:
2      def addFormSubmit(self, name, comment):
3          self.data.add(name, comment)
4          return self.redirect('index')
5  mask:
6      def index(self):
7          <py-eval="self.header('View Guestbook')">
8          <py-eval="self.viewGuestbook()">
9          <center><a href="addForm">Add New Record</a></center>
10         <py-eval="self.footer()">
11     def addForm(self):
12         <py-eval="self.header('Add to Guestbook')">
13         <form action="addFormSubmit" method="post">
14         Name: <input type="text" name="name"><br>
15         Comment:
16             <textarea rows=5 cols=40 name="comment"></textarea><br>
17             <input type="submit">
18         </form>
19         <py-eval="self.footer()">

```

# Example: Compiling and Running

```
1 jsc@mico:~/sample$ cherrypy Root.cpy Guestbook.cpy
2 jsc@mico:~/sample$ python RootServer.py
3 Reading parameters from RootServer.cfg ...
4 Server parameters:
5   logToScreen: 1
6   logFile:
7   socketPort: 0
8   socketFile:
9   fixedNumberOfProcesses: 1
10  threading: 0
11  forking: 0
12  sslKeyFile:
13  sslCertificateFile:
14  xmlRpc: 0
15  flushCacheDelay: 0 min
16  staticContent: []
17 Calling initServer() ...
18 Serving HTTP on socket port: 8000
```

Now point your web browser to "http://127.0.0.1:8000" and voila!

# Why CherryPy? (1/2)

- It's Open Source (GPL'd) – The GPL is only applied to the Cherry compiler. The output of the compiler, your website, is your property.
- Python based – Python is a very good tool for rapidly developing software and it shows here. We have access to all the standard python libraries.
- Again... Fantastic documentation.
- Minimal Dependencies / Simple Installation – Server only requires Python to run. Plugins into Apache using standard modules.
- Cross platform – Runs on any platform that supports Python (Win, Lin, Max, Jython (JAVA)).
- Applications can run behind other webservers or standalone.
- Simple – Very logical and pythonic. Easy to get started.

# Why CherryPy? (2/2)

- Gives an easy way for hobby users to setup a website/app without the effort of installing Apache, PHP, ...
- Great way to build web apps for a third party. Installation is always easy and very few dependencies.
- Easy debugging: Errors just throw standard Python exceptions with traceback.
- Very fast, stable, and scales well. <http://www.waypath.com/> runs CP with about 10k hits/day with no problems.
- The underlying Python HTTPd is quite stable and has a good security track record.

# References

- <http://cherrypy.org>  
CherryPy homepage
- <http://tinyurl.com/fc6w>  
Introduction to CherryPy
- <http://www.freecherrypy.org/>  
Free CherryPy application hosting for personal use
- <http://www.python-hosting.com/>  
Commercial hosting
- <http://jclement.ca/clug/cherrypy-clug>  
This presentation in PDF/TEX formats as well as sample code
- <http://python.org>  
Homepage of the Python programming language
- <http://jclement.ca/software/jcard2/>  
JCard2 is a little web contact manager I wrote in CherryPy. Might be handy for a bigger example.