Introduction to Python

Christopher Barker

UW Continuing Education / Isilon

July 25, 2012

Table of Contents

Review of Previous Class

- Keyword arguments/parameters
- Lists
- Dictionaries
- Sets

Homework review

Homework notes

Class Structure

This class is different – more a tutorial than a class: lots of coding.

We're going to run through building a really basic HTTP server from the ground up.

We'll see how far we get.

Note: I'm no expert – I'm learning along with you...

Sockets

"Socket" at either end of a pathway: client and server can be "plugged in" to communicate
Five pieces of data to uniquely identify a connection

- Transport protocol (UDP, TCP) (we'll use TCP)
- remote IP address
- Remote port number
- Local IP address
- Local port number

(use localhost on both ends for this class...)

Python Socket Module

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

`AF_INET` : Internet Family of Protocols
`SOCK_STREAM` : TCP

Set some options:

```
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
```

`SOL_SOCKET` : ???
`SO_REUSEADDR` : re-use the address – so the OS won't reserve it

A Socket Client/Server

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# set an option to tell the OS to re-use the socket
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

# the bind makes it a server
s.bind( (host,port) )
s.listen(backlog)

while True: # keep looking for new connections forever
    client, address = s.accept() # look for a connection
    data = client.recv(size)
    if data: # if the connection was closed there would be
        client.send(data)
        client.close()
```

HTTP

Hyper**T**ext **T**ransfer **P**rotocol

Client-Server:

- requests
- responses

Each has:

- Method specification (request)
- Status line (response)
- Headers (RFC 822-compliant)

(optionally)

- Entity headers and body

(RFC 2616)

HTTP Requests

Request Methods

- GET – Request a URI content
- HEAD – GET headers only
- POST – PUT save URI content
- PUT – POST Request URI content, with entity transfer to server

There are four others – but these are the ones most used

HTTP request

Example HTTP GET request

```
GET /a_file HTTP/1.1
Host: localhost:55555
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.7; rv
Accept: text/html,application/xhtml+xml,application/xml;q=0
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
```

HTTP Responses

Response Codes

- 200 OK
- 404 Not Found
- 301 Moved Permanently
- 302 Moved Temporarily
- 303 See Other (HTTP 1.1 only)
- 500 Server Error

There are four others – but these are the ones most used

HTTP Response header

```
HTTP/1.1 200 OK
Date: Fri, 31 Dec 1999 23:59:59 GMT
Content-Type: text/html
Content-Length: 1354

<html>
<body>
<h1>Happy New Millennium!</h1>
(more file contents)
... </body> </html>
```

Blank line between header and body critical!

(\r\n linefeeds)

HTTP Response header

Header-Name: value

Quick reference to HTTP headers:

`http://www.cs.tut.fi/~jkorpela/http.html`

HTTP Response header

body data:

```
Content-Type: xyz/abc
```

Mime types we might want:

- `text/plain`
- `text/html`
- `image/png`
- `image/jpeg`

`http://www.webmaster-toolkit.com/mime-types.shtml`

Debugging

Debugging Tools

- windows:
  `http://www.fiddler2.com/fiddler2/`
- windows & mac:
  `http://www.charlesproxy.com/`
- Firefox:
  `http://getfirebug.com/`
- Safari, Chrome and IE: built in

Building an HTTP Server

We've got everything we need to know to build a simple server

(GET only for now...)

Build an HTTP server that can serve up the files in:

`week-05\code\web`

NOTE: you can use the date formatting found in `httpdate.py`

Building an HTTP Server

Incremental Development:

1. A socket server that can receive a request (and print that request to the console)
2. Server returns a simple reply
3. Server returns a properly formatted HTTP reply
4. Server returns a 404 error
5. Server returns the file asked for
6. Server returns a directory listing
7. Server returns multiple file types
8. Server returns a calculated response

http_serve1.py

Edit `echo_serve1.py` to print the request

1. Point your browser at `echo_server.py` – what do you get?
2. Save it as `http_serve1.py`
3. Edit it to print the request to the console
4. Edit it to return a bit of html (`tiny_html.html`)
5. What happens when you point your browser to it?
   Try a couple different browsers – I get a different result with Firefox and Safari

http_serve2.py

Return a properly formatted HTTP response

1. Save `http_serve1.py` as `http_serve2.py`
2. Add code that generates an HTTP "OK" header (don't forget the blank line! (`\r\n`)
3. Use `httpdate.httpdate_now()` to give you an HTTP date string
4. What happens when you point your browser to it now?

http_serve3.py

Parse the request

1. Save `http_serve2.py` as `http_serve3.py`
2. Add code that parses the HTTP request – it should give you the URI requested
3. Have it check to make sure it's a GET request
4. print the URI (file name) to the console

http_serve4.py

Parse the request

1. Save `http_serve3.py` as `http_serve4.py`
2. Add code that parses the URI – so you can figure out what file is requested
3. check to see if is a directory or a file
4. return a listing (simple text) of the dir if it's a dir
5. return a 404 otherwise

http_serve5.py

Support various file types

1. Save `http_serve4.py` as `http_serve5.py`
2. If the request is for a file – return that file
3. have it be a different mime type depending on the type of file
4. support: `.html`, `.txt`, `.jpeg`, `.png`

You now have a pretty functional web server!

http_serve6.py

If we have time...

1. Save `http_serve5.py` as `http_serve6.py`
2. Format the Dir listing as HTML
3. Make the files in the listing clickable links.

and / or

1. Make a simple web app (non-static)
2. Have `localhost:50000/the_time` return a web page with the current time.

You now have a very functional web server!