Web Processing

olsen software

Contents

1. Python Web servers
2. Python Rest services
3. Python Web sockets

Annex

- HTML5 Web sockets clients

Demo folder: **11-Web**

1. Python Web Servers

- Python support for HTTP
- Starting the HTTP server
- Defining an HTTP request handler class
- Servicing HTTP requests
- Running the HTTP server
- Dynamic content example
- Static content example

Python Support for HTTP

- Python provides a set of APIs that enable you to implement an HTTP Web server in Python
  - Using classes in the `http.server` module

- Here's the big picture:
  - Create an `HTTPServer` object to listen on a particular port
  - Define a subclass of `BaseHTTPRequestHandler`, to handle incoming requests from clients
  - Start the server

- We'll see a complete example of how to do this
  - In the demo folder, see `HttpServer/webserver.py`

Starting the HTTP Server

- The following code shows how to start an HTTP server
  - Note: `MyHandler` is our custom HTTP request handler class
  - We'll discuss this on the following slides

```
from os import curdir, sep
from http.server import BaseHTTPRequestHandler, HTTPServer
import mimetypes
…

def main():
    try:
        server = HTTPServer(('', 8001), MyHandler)
        print('Started HTTP server...')
        server.serve_forever()

    except KeyboardInterrupt:
        print('Ctrl+C received, shutting down server')
        server.socket.close()

if __name__ == '__main__':
    main()
```

Defining an HTTP Request Handler Class

- To define an HTTP request handler class, to handle incoming requests from the client:
  - Define a class that inherits from `BaseHTTPRequestHandler`
  - Implement do_`GET()` if you want to handle HTTP GET requests
  - Implement do_`POST()` if you want to handle HTTP POST requests

- Example

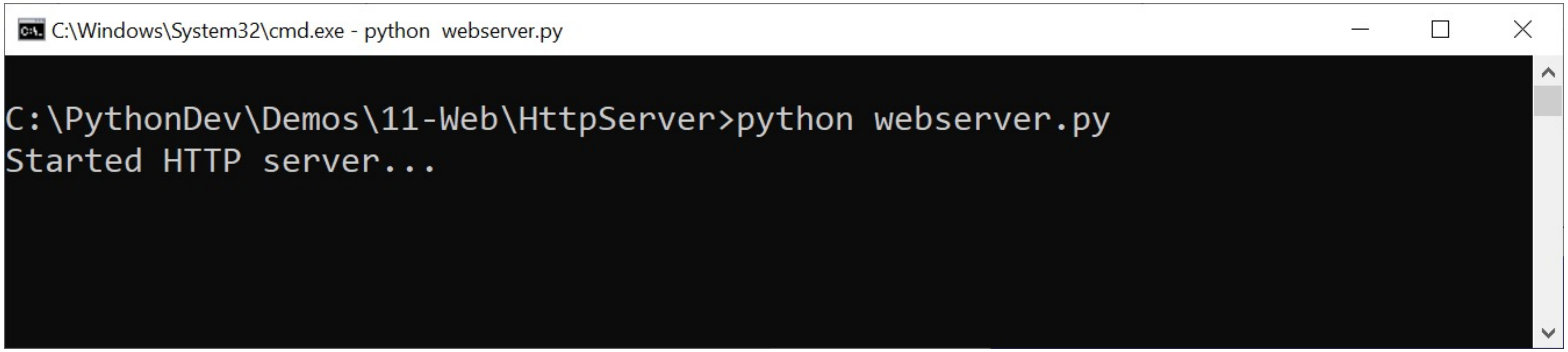
```
class MyHandler(BaseHTTPRequestHandler):

    def do_GET(self):
        …

    def do_POST(self):
        …
```

Servicing HTTP Requests

```
def do_GET(self):
    …

    if self.path.endswith(".zzz"):   # Our made-up dynamic content.

        self.send_response(200)
        self.send_header('Content-type', 'text/html')
        self.end_headers()

        result = "You requested {0} on day {1} in {2}" \
                       .format(self.path,
                               time.localtime()[7],
                               time.localtime()[0])

        self.wfile.write(result.encode('utf-8'))

    else:
        f = open(curdir + sep + self.path)
        self.send_response(200)
        mimeType = mimetypes.guess_type(self.path)[0]
        self.send_header('Content-type', mimeType)
        self.end_headers()
        self.wfile.write(f.read().encode('utf-8'))
        f.close()
```

Running the HTTP Server

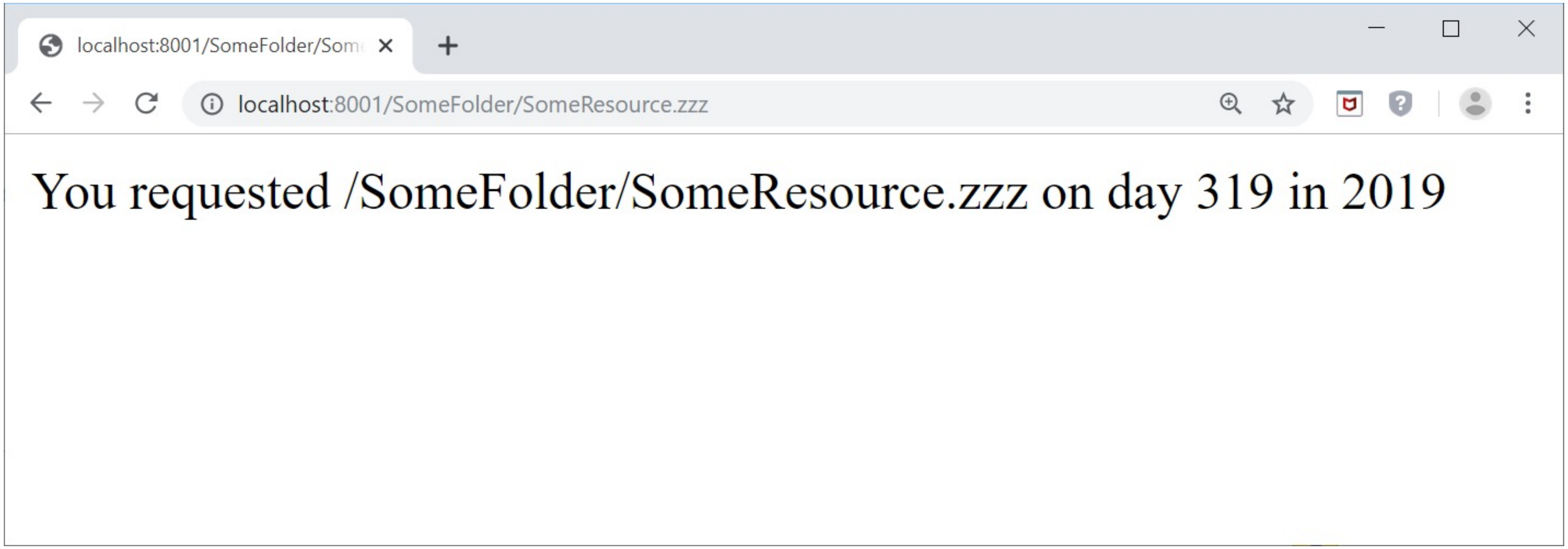
- Run the Python HTTP server script as follows:



```
C:\PythonDev\Demos\11-Web\HttpServer>python webserver.py
Started HTTP server...
```

Dynamic Content Example

- Here's what happens if we request a "dynamic" resource
  - E.g. http://localhost:8001/SomeFolder/SomeResource.zzz

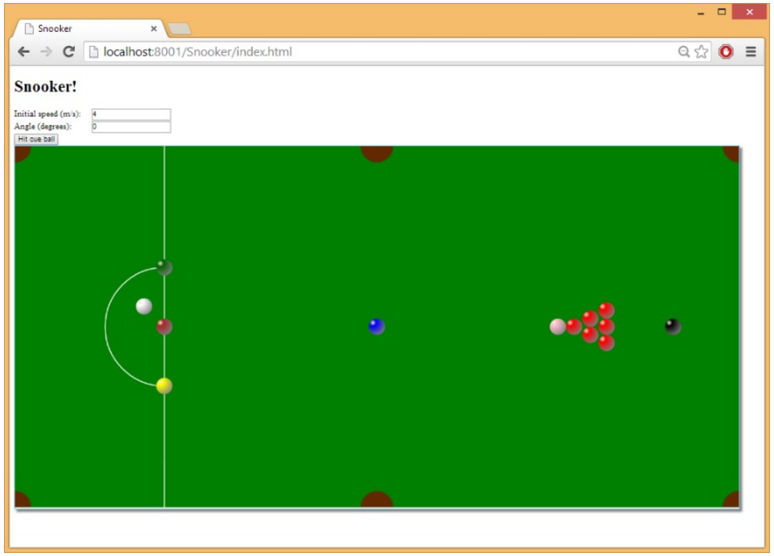


localhost:8001/SomeFolder/Som
localhost:8001/SomeFolder/SomeResource.zzz

You requested /SomeFolder/SomeResource.zzz on day 319 in 2019

Static Content Example

- Here's what happens if we request static resource
  - http://localhost:8001/Snooker/index.html
  - http://localhost:8001/Catch/index.html
  - http://localhost:8001/Breakout/index.html

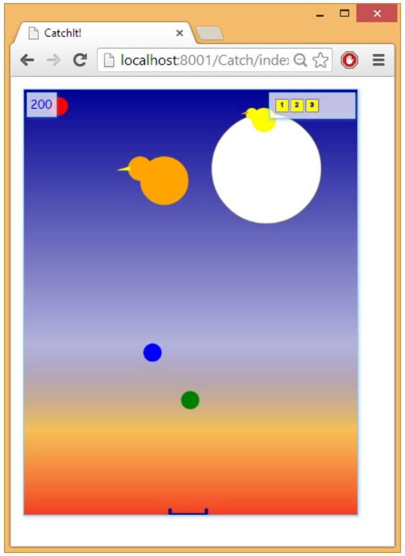


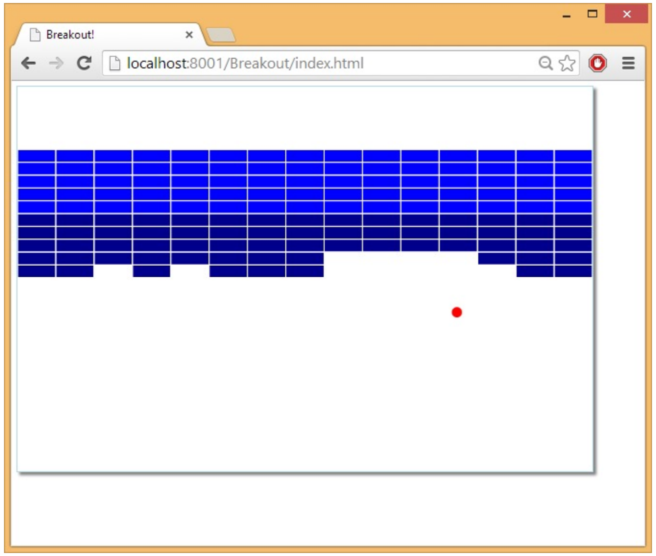
Snooker!

Initial speed (m/s): 4

Angle (degrees): 0

Hit cue ball





2. Python Rest Services

- The name "Rest"
- What is a Rest service?
- HTTP verbs
- HTTP response codes
- Key principles of Rest services
- Implementing a Rest service in Python
- Calling a Rest service in Python

The Name "Rest"

> The name “Representational State Transfer” is intended to evoke an image of how a well-designed Web application behaves: a network of Web pages forms a virtual state machine, allowing a user to progress through the application by selecting a link or submitting a short data-entry form, with each action resulting in a transition to the next state of the application by transferring a representation of that state to the user.
>
> *Fielding & Taylor 2002*

What is a Rest Service?

- Rest services are resource-centric services
  - Endpoints (URIs) represent resources
  - Endpoints are accessible via standard HTTP
  - Endpoints can be represented in a variety of formats (e.g. XML, JSON, HTML, plain text)

HTTP Verbs

- Rest services use HTTP verbs to define CRUD-style operations on resources

HTTP verb	Meaning in CRUD terms
POST	**C**reate a new resource from the request data
GET	Read a resource
PUT	**U**pdate a resource from the request data
DELETE	Delete a resource

HTTP Response Codes

- Rest services return data, and set a response code to indicate the outcome

HTTP response code	Official HTTP meaning	Rest meaning
200	OK	Request OK
201	Created	New resource created OK
400	Bad request	Request malformed
403	Forbidden	Request refused
404	Not found	Resource not found
405	Method not allowed	Method not supported
415	Unsupported media type	Content type not recognized
500	Internal server error	Request processing failed

Key Principles of Rest Services

- Rest services are based on standard technologies
  - HTTP, URIs, XML, JSON, etc.
  - But not SOAP!

- HTTP verbs specify CRUD operations
  - POST, GET, PUT, DELETE

- Focus on resources
  - Resource-centric vs. API-centric
  - Resources are identified using URIs (name everything)
  - Resources are connected through links (reveal gradually)
  - Resources may have different representations (XML, JSON, (X)HTML, plain text, ATOM, etc.)

Implementing a Rest Service in Python

- There are many Python packages available, to help you implement a Rest service…
  - We'll show an example using Flask
  - Install the following Python packages:

```
pip install flask
pip install flask_restful
```

- We've implemented a complete sample Rest service
  - In the demo folder, see `Rest\server.py`
  - The code contains detailed comments, explaining how it works

- Run the application as follows (in the `Rest` demo folder)
  - Starts the Rest service listening on `http://localhost:5000`

```
python server.py
```

Calling a Rest Service in Python

- There are many Python packages available, to help you issue HTTP requests (e.g. to call a Rest service)
  - We'll show an example using the "requests" package
  - Install the package as follows:

```
pip install requests
```

- We've implemented a complete sample Rest client
  - In the demo folder, see `Rest\client.py`
  - The code contains detailed comments, explaining how it works

- Run the application as follows (in the `Rest` demo folder)
  - Issues GET/PUT/POST/DELETE requests to our server app

```
python client.py
```

3. Python Web Sockets

- Issues with traditional HTTP
- Web sockets to the rescue
- How Web sockets work
- Introducing the Python Web sockets API
- Implementing a Web sockets server
- Implementing a Web sockets client
- Running the server and client(s)

Issues with Traditional HTTP

- Traditionally, when a browser visits a web page:
  - An HTTP request is sent to the web server that hosts that page
  - The web server acknowledges this request and sends back the response

- In some cases, the response could be stale by the time the browser renders the page
  - E.g. stock prices, news reports, ticket sales, etc.

- How can you ensure you get up-to-date information?
  - Polling
  - Long polling

Web Sockets to the Rescue

- Web sockets are a powerful communication feature in the HTML5 specification

- Web sockets defines a full-duplex communication channel between browser and server
  - Simultaneous 2-way data exchange between browser and server
  - A large advance in HTTP capabilities
  - Extremely useful for real-time, event-driven Web applications

How Web Sockets Work

- To support real-time full-duplex communication between a client and server:
  - The client and server upgrade from the HTTP protocol to the Web sockets protocol during their initial handshake

- Thereafter, client and the server can communicate in full-duplex mode over the open connection
  - Allows the server to push information to the client, when the data becomes available
  - Allows the client and server to communicate simultaneously

Introducing the Python Web Sockets API

- You can define a Web sockets server in Python code
  - Via the `websockets` standard module

```
import websockets
```

- You must implement the server to support asynchronous calls from multiple clients
  - So you'll need the `asyncio` standard module too

```
import asyncio
```

- We'll see how to implement a Python Web sockets server in the next few slides
  - See the demo in `WebSockets\server.py`

Implementing a Web Sockets Server

- Here's the full implementation for a Web sockets server in Python!

```
import asyncio
import websockets

async def onconnect(websocket, uri):

    while True:
        datain = await websocket.recv()
        print("From client: %s" % datain)

        dataout = "ECHO! " + datain
        print("To client:   %s" % dataout)

        await websocket.send(dataout)

start_server = websockets.serve(onconnect, 'localhost', 8002)

asyncio.get_event_loop().run_until_complete(start_server)
asyncio.get_event_loop().run_forever()
```

Implementing a Web Sockets Client

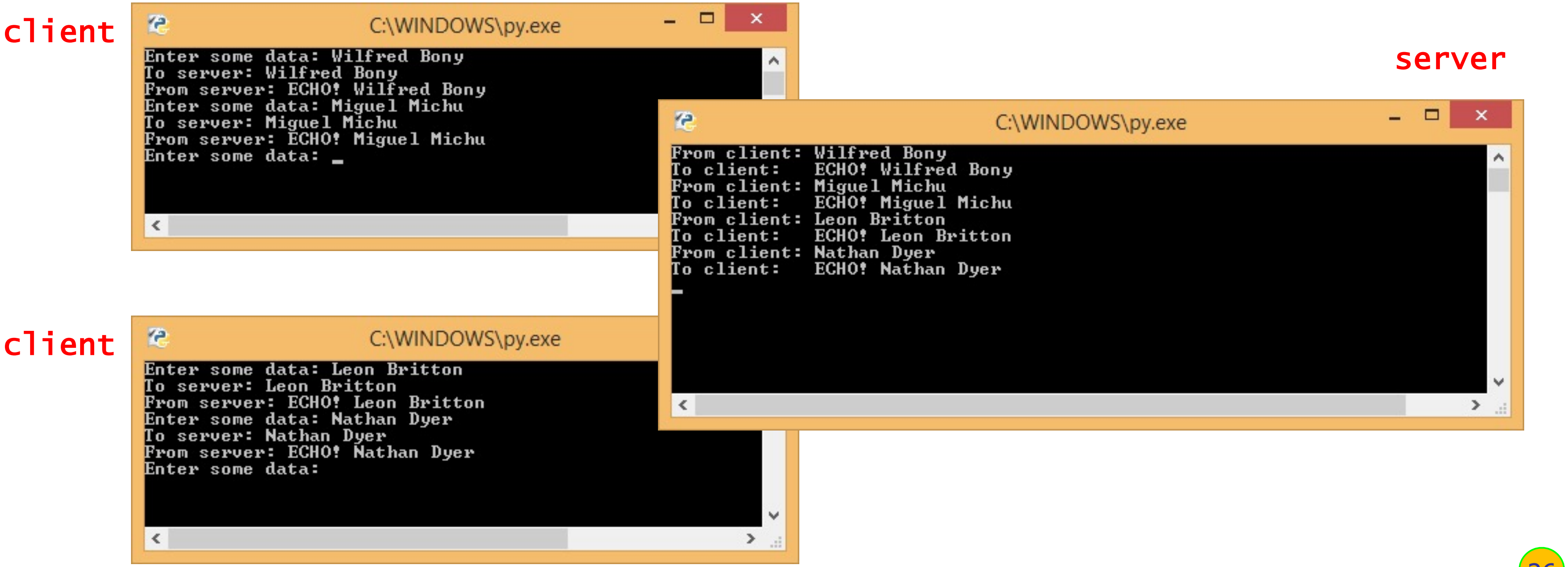
- You can implement a Web sockets client in Python too

```
import asyncio
import websockets

async def client():

    websocket = await websockets.connect('ws://localhost:8002/')

    while True:
        name = input("Enter some data: ")

        print("To server: %s" % name)
        await websocket.send(name)

        resp = await websocket.recv()
        print("From server: %s" % resp)

asyncio.get_event_loop().run_until_complete(client())
```

Running the Server and Client(s)

- First run `server.py`, then run `client.py`
  - You can fire up many instances of the client

- The client(s) and server can then communicate with each other over the Web sockets protocol



```
Enter some data: Wilfred Bony
To server: Wilfred Bony
From server: ECHO! Wilfred Bony
Enter some data: Miguel Michu
To server: Miguel Michu
From server: ECHO! Miguel Michu
Enter some data: _
```

```
From client: Wilfred Bony
To client:   ECHO! Wilfred Bony
From client: Miguel Michu
To client:   ECHO! Miguel Michu
From client: Leon Britton
To client:   ECHO! Leon Britton
From client: Nathan Dyer
To client:   ECHO! Nathan Dyer
_
```

```
Enter some data: Leon Britton
To server: Leon Britton
From server: ECHO! Leon Britton
Enter some data: Nathan Dyer
To server: Nathan Dyer
From server: ECHO! Nathan Dyer
Enter some data:
```

Summary

- Python Web servers
- Python Web sockets

Optional lab idea

Enhance the Python Web sockets server so that it keeps a collection of all the connected clients.

Whenever any client communicates with the server, the server should broadcast the message to all connected clients.

For extra merit, send back a "special" response to the client that actually sent you the data.

Annex: HTML5 Web Sockets Clients

- Overview
- Checking for Web Sockets support
- Opening a connection
- Handling events
- Sending data to the server
- Receiving data from the server
- Closing the connection
- Complete client example

Overview

- In this section we'll show how to write a client Web page to call a Web sockets service
  - The client creates a `WebSocket` JavaScript object
  - Does your browser support this object …?

- The HTML web page is available here:
  - `WebSockets/client.html`
  - Concentrate on the JavaScript code

Checking for Web Sockets Support

- To check whether your browser supports HTML5 Web sockets:

```
function testWebSocketSupport() {

  if (window.WebSocket) {
    alert("Your browser supports HTML5 Web sockets");
  }
  else {
    alert("Your browser doesn't support HTML5 Web sockets");
  }
}
```

Opening a Connection

- Using the `WebSocket` interface is straightforward…

- To open a connection to the server:
  - Create a `WebSocket` object, specifying the URL to connect to
  - Use `ws://` prefix for WebSocket connections
  - Use `wss://` prefix for secure WebSocket connections

```
var url = "ws://localhost:8002/";
var ws;

function doInit() {
  ws = new WebSocket(url);
  …
}
```

Handling Events

- The Web sockets JavaScript API is asynchronous
  - You therefore have to handle events as follows

```
var url = "ws://localhost:8002/";
var ws;

function doInit() {
  ws = new WebSocket(url);

  ws.onopen    = function(e) { … };
  ws.onclose   = function(e) { … };
  ws.onmessage = function(e) { … };
  ws.onerror   = function(e) { … };
}
```

Sending Data to the Server

- To send data to the Web Socket server
  - Call the `send()` method
  - You can pass text, binary, or array data

```
ws.send(sometextdata);

ws.send(somebinarydata);

ws.send(somearraydata);
```

Receiving Data from the Server

- To receive data messages from the server:
  - Handle the `message` event

- The event argument has `type` and `data` properties
  - The type property is either `"text"` or `"binary"`
  - If `"binary"`, the `WebSocket` object has a `binaryType` property that indicates if it's a `"blob"` or an `"arrayBuffer"`

```
function onMessage(e) {
  alert("Received data from server: " + e.data);

  if (e.type == "text") {
    alert("It's text data");
  }
  else {
    if (ws.binaryType == "blob")
      alert("It's a blob [e.g. an image]");
    else if (ws.binaryType == "arrayBuffer")
      alert("It's an array");
  }
}
```

Closing a Connection

- To open a connection to the server:
  - Call `close()` on the WebSocket object
  - Optionally pass `code` and `reason` parameters

```
ws.close();
```

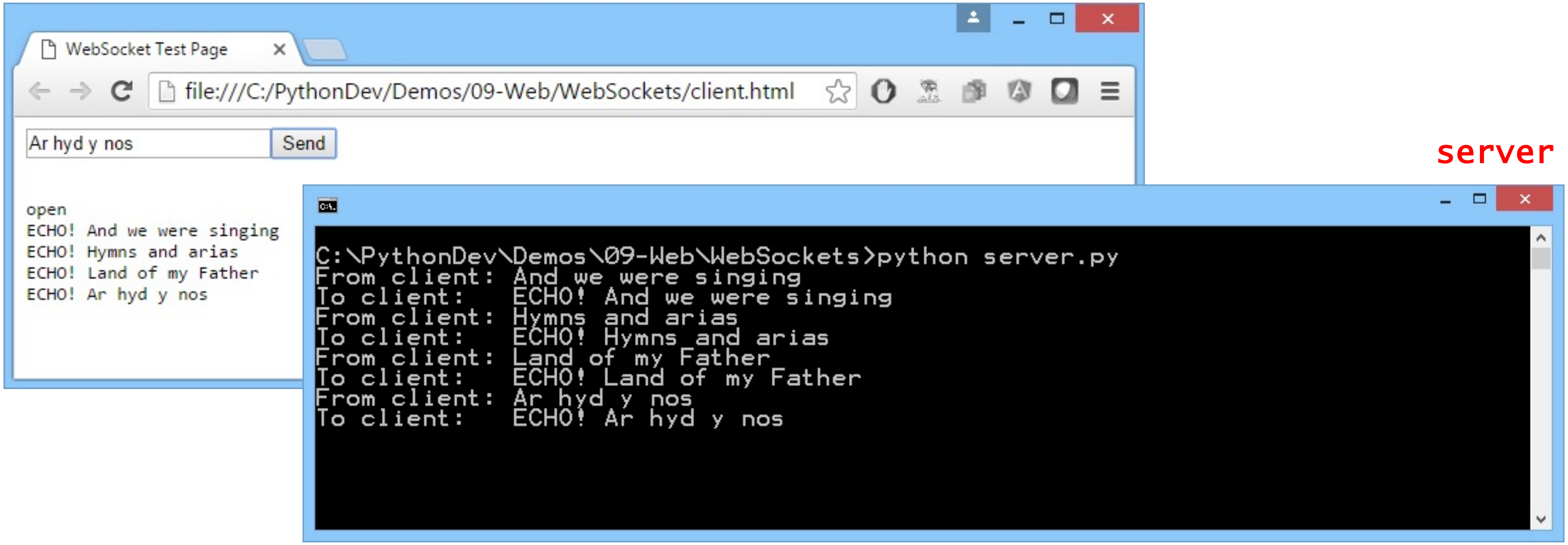
- When the connection has been closed, the `close` event occurs
  - The event object has `wasClean`, `code`, and `reason` properties

Running the Server and Client(s)

- Run `server.py`, then open `client.html` in a browser
  - You can fire up many instances of the client Web page

HTML5 client

server



```
open
ECHO! And we were singing
ECHO! Hymns and arias
ECHO! Land of my Father
ECHO! Ar hyd y nos
```

```
C:\PythonDev\Demos\09-Web\WebSockets>python server.py
From client: And we were singing
To client:   ECHO! And we were singing
From client: Hymns and arias
To client:   ECHO! Hymns and arias
From client: Land of my Father
To client:   ECHO! Land of my Father
From client: Ar hyd y nos
To client:   ECHO! Ar hyd y nos
```

Any Questions?