Advanced Web Programming

1. Advanced Web Programming
   - what we have covered so far

2. The SocketServer Module
   - simplified development of network servers
   - a server tells clients the time

3. A Forking Server
   - instead of threads use processes
   - process to handle a client

4. The BaseHTTPServer Module
   - creating a very simple HTTP server
   - code for the simple HTTP web server

MCS 275 Lecture 33
Programming Tools and File Management
Jan Verschelde, 3 April 2017
Advanced Web Programming

1. Advanced Web Programming
   - what we have covered so far

2. The SocketServer Module
   - simplified development of network servers
   - a server tells clients the time

3. A Forking Server
   - instead of threads use processes
   - process to handle a client

4. The BaseHTTPServer Module
   - creating a very simple HTTP server
   - code for the simple HTTP web server
Plan of the Course
since the first midterm

In the four weeks after the midterm exam we covered:

1. CGI programming: handling forms
2. database programming: MySQL and MySQLdb
3. network programming: using sockets
4. multithreaded programming

Anything left to cover?

*Advanced Web Programming*

→ gluing various programming tools
Advanced Web Programming

1. Advanced Web Programming
   - what we have covered so far

2. The SocketServer Module
   - simplified development of network servers
   - a server tells clients the time

3. A Forking Server
   - instead of threads use processes
   - process to handle a client

4. The BaseHTTPServer Module
   - creating a very simple HTTP server
   - code for the simple HTTP web server
With the `SocketServer` module we do not need to import the `socket` module for the server script.

Follow these steps:

1. `from socketserver import StreamRequestHandler`
   `from socketserver import TCPServer`

2. Inheriting from `StreamRequestHandler` define a request handler class. Override `handle()`.
   → `handle()` processes incoming requests

3. Instantiate `TCPServer` with `(address, port)` and an instance of the request handler class.
   → this returns a server object

4. Apply the method `handle_request()` or `serve_forever()` to the server object.
Advanced Web Programming

1. Advanced Web Programming
   - what we have covered so far

2. The SocketServer Module
   - simplified development of network servers
   - a server tells clients the time

3. A Forking Server
   - instead of threads use processes
   - process to handle a client

4. The BaseHTTPServer Module
   - creating a very simple HTTP server
   - code for the simple HTTP web server
a server to tell the time with `SocketServer`

In the window running the server:

```
$ python clockserver.py
server is listening to 12091
connected at ('127.0.0.1', 49142)
read "What is the time? " from client
writing "Sun Apr 4 18:16:14 2010" to client
```

In the window running the client:

```
$ python clockclient.py
client is connected
Sun Apr 4 18:16:14 2010
```
```python
from socket import socket as Socket
from socket import AF_INET, SOCK_STREAM

HOSTNAME = 'localhost'  # on same host
PORTNUMBER = 12091      # same port number
BUFFER = 25             # size of the buffer

SERVER_ADDRESS = (HOSTNAME, PORTNUMBER)
CLIENT = Socket(AF_INET, SOCK_STREAM)
CLIENT.connect(SERVER_ADDRESS)

print('client is connected')
QUESTION = 'What is the time?'
DATA = QUESTION + (BUFFER-len(QUESTION))*''
CLIENT.send(DATA.encode())
DATA = CLIENT.recv(BUFFER)
print(DATA.decode())

CLIENT.close()
```
from socketserver import StreamRequestHandler
from socketserver import TCPServer
from time import ctime

PORT = 12091

class ServerClock(StreamRequestHandler):
    """
    The server tells the clients the time.
    """
    def handle(self):
        """
        Handler sends time to client.
        """

    def main():
        """
        Starts the server and serves requests.
        """
def handle(self):
    """
    Handler sends time to client.
    """
    print("connected at", self.client_address)
    message = self.rfile.read(25)
    data = message.decode()
    print(‘read ‘ + data + ‘\” from client’)  
    now = ctime()
    print(‘writing ‘ + now + ‘\” to client’)  
    self.wfile.write(now.encode())
def main():
    """
    Starts the server and serves requests.
    """
    ss = TCPServer(('', PORT), ServerClock)
    print('server is listening to', PORT)
    try:
        print('press ctrl c to stop server')
        ss.serve_forever()
    except KeyboardInterrupt:
        print(' ctrl c pressed, closing server')
        ss.socket.close()

if __name__ == "__main__":
    main()
About `rfile` and `wfile` attributes in the class `StreamRequestHandler`

- `rfile` contains input stream to read data from client
  
  example: `data = self.rfile.read(25)`

  *client must send exactly 25 characters!*

- `wfile` contains output stream to write data to client
  
  example: `self.wfile.write(data)`

  *all data are strings of characters!*
alternatives to the simple example

Instead of `StreamRequestHandler`, we can use `DatagramRequestHandler`.

Instead of `TCPServer`, we can use `UDPServer`, if we want UDP instead of TCP protocol.

On Unix (instead of `TCPServer`): `UnixStreamServer` or `UnixDatagramServer`.

Choice between

1. `handle_request()`: handle one single request, or
2. `serve_forever()`: indefinitely many requests.
using `serve_forever()`

With `serve_forever()`, we can
1. serve indefinitely many requests,
2. simultaneously from multiple clients.

```python
ss = TCPServer(('', port), ServerClock)
print 'server is listening to', port
try:
    print 'press ctrl c to stop server'
    ss.serve_forever()
except KeyboardInterrupt:
    print 'ctrl c pressed, closing server'
    ss.socket.close()
```
Advanced Web Programming

1. Advanced Web Programming
   - what we have covered so far

2. The SocketServer Module
   - simplified development of network servers
   - a server tells clients the time

3. A Forking Server
   - instead of threads use processes
   - process to handle a client

4. The BaseHTTPServer Module
   - creating a very simple HTTP server
   - code for the simple HTTP web server
a forking server

Threads in Python are not mapped to cores.
For computationally intensive request, we want to spawn a new process.

>>> import os
>>> help(os.fork)
Help on built-in function fork in module posix:

fork(...)  
    fork() -> pid

Fork a child process.
Return 0 to child process
and PID of child to parent process.
The child process will just print `hello`.

```python
import os

def child():
    """
    The code executed by the forked process.
    """
    print('hello from child', os.getpid())
    os._exit(0)  # go back to parent loop
```
def parent():
    
    # Code executed by the forking process.
    Type q to quit this process.
    #
    while True:
        newpid = os.fork()
        if newpid == 0:
            child()
        else:
            print('hello from parent',
                  os.getpid(), newpid)
        if input() == 'q':
            break

parent()
running fork.py

$ python fork.py
hello from parent 854 855
hello from child 855

In another terminal window:

$ ps -e | grep "Python"
  854 ttys000  0:00.03 /Library/Frameworks/Python.framework/Versions/2.7/Resources/Python.app/Contents/MacOS/Python
  855 ttys000  0:00.00 (Python)
  895 ttys001  0:00.00 grep Python

Then we type q in the first terminal window to quit the parent process.
Advanced Web Programming

1. Advanced Web Programming
   - what we have covered so far

2. The SocketServer Module
   - simplified development of network servers
   - a server tells clients the time

3. A Forking Server
   - instead of threads use processes
   - process to handle a client

4. The BaseHTTPServer Module
   - creating a very simple HTTP server
   - code for the simple HTTP web server
Are we there yet?

Consider the following simulation:

- Any number of clients connect from time to time and they ask for the current time.

- For every request, the server forks a process. The child process exits when the client stops.

Two advantage of forking processes over threads:

1. We have parallelism, as long as there are enough cores.
2. Unlike threads, processes can be killed explicitly.
clockforkclient.py

We have the same start as in clockclient.py

print('client is connected')
data = 'What is the time?'

while True:
    message = data + (buffer-len(data))*''
    client.send(message.encode())
data = client.recv(buffer).decode()
print(data)
nbr = randint(3, 10)
print('client sleeps for %d seconds' % nbr)
sleep(nbr)

client.close()
process handling a client

def handle_client(sck):
    """
    Handling a client via the socket sck.
    """

    print("client is blocked for ten seconds ...")
    sleep(10)
    print("handling a client ...")
    while True:
        data = sck.recv(buffer).decode()
        if not data:
            break
        print('received "' + data + '" from client')
        now = ctime()
        print('sending "' + now + '" to client')
        sck.send(now.encode())
    print('closing client socket, exiting child process')
    sck.close()
    os._exit(0)
killing the handling child processes

With the `os` module, we can kill a process, once we have its process id.

```python
import os

active_processes = []

def kill_processes():
    """
    kills handler processes
    """

    while len(active_processes) > 0:
        pid = active_processes.pop(0)
        print('-> killing process %d' % pid)
        os.system('kill -9 %d' % pid)
```
def main():
    
    """
    Listen for connecting clients.
    """
    try:
        print('press ctrl c to stop server')
        while True:
            client, address = server.accept()
            print('server connected at', address)
            child_pid = os.fork()
            if child_pid == 0:
                handle_client(client)
            else:
                print('appending PID', child_pid)
                active_processes.append(child_pid)
shutting down the server

Before closing the server socket, all active child processes are killed.

```
except:
    print('ctrl c pressed, closing server')
    print('active processes :', active_processes)
    kill_processes()
    server.close()

if __name__ == "__main__":
    main()
```
Advanced Web Programming

1. Advanced Web Programming
   - what we have covered so far

2. The SocketServer Module
   - simplified development of network servers
   - a server tells clients the time

3. A Forking Server
   - instead of threads use processes
   - process to handle a client

4. The BaseHTTPServer Module
   - creating a very simple HTTP server
   - code for the simple HTTP web server
Client Accesses the HTTP Server

The client is the web browser.

Working offline, with URL http://localhost:8000/

This is the default page displayed in response to a GET request.
Not Serving Files

For now, our server does not make files available.

If a user requests a file, e.g.: test, then the server answers:

![Image of server response: /test not found](image-url)
running the web server in ourwebserver.py

Recall the script `myserver.py` which allowed us to do server side Python scripting without Apache.

We can also serve html pages without Apache:

```
$ python3 ourwebserver.py
welcome to our web server
press ctrl c to stop server
127.0.0.1 - - [04/Apr/2016 09:20:55] "GET / HTTP/1.1" ^C ctrl c pressed, shutting down
$
```
Advanced Web Programming

1. Advanced Web Programming
   - what we have covered so far

2. The SocketServer Module
   - simplified development of network servers
   - a server tells clients the time

3. A Forking Server
   - instead of threads use processes
   - process to handle a client

4. The BaseHTTPServer Module
   - creating a very simple HTTP server
   - code for the simple HTTP web server
The BaseHTTPServer Module
writing code for a web server

Using the BaseHTTPServer module is similar to using SocketServer.

Execute these steps:

1. Import the following:
   ```python
   from BaseHTTPServer import BaseHTTPRequestHandler
   from BaseHTTPServer import HTTPServer
   ```
2. Inheriting from `BaseHTTPRequestHandler` define request handler class. Override `do_GET()`.
   ```python
   → do_GET() defines how to serve GET requests
   ```
3. Instantiate `HTTPServer` with `(address, port)` and an instance of the request handler class.
   ```python
   → this returns a server object
   ```
4. Apply `serve_forever()` to server object.
part I of `ourwebserver.py`

```python
from http.server import BaseHTTPRequestHandler
from http.server import HTTPServer

dynhtml = """"
<HTML>
<HEAD><TITLE>My Home Page</TITLE></HEAD>
<BODY>  <CENTER>
<H1> hello client </H1>
</CENTER>  </BODY>
</HTML>"""

This defines the HTML code we display.
class WebServer(BaseHTTPRequestHandler):
    ""
    Illustration to set up a web server.
    ""

def do_GET(self):
    ""
    Defines what server must do when it receives a GET request.
    ""
    if self.path == '/':
        self.send_response(200)
        self.send_header('Content-type','text/html')
        self.end_headers()
        self.wfile.write(dynhtml.encode())
    else:
        message = self.path + ' not found'
        self.wfile.write(message.encode())
def main():
    ""
    a simple web server
    ""
try:
    ws = HTTPServer((' ', 8000), WebServer)
    print('welcome to our web server')
    print('press ctrl c to stop server')
    ws.serve_forever()
except KeyboardInterrupt:
    print(' ctrl c pressed, shutting down')
    ws.socket.close()
Summary + Assignments

Assignments:

1. Use the `SocketServer` module to implement a server to swap one data string between two clients. Clients A and B send a string to the server, client B receives what A sent and A receives what B sent.

2. Implement a server which generates a secret number. Clients connect to the server sending their guess for the secret. In response, the server sends one of these three messages: (1) wrong, (2) right, or (3) secret found. If a client has sent the right answer, all future clients must get reply (3).

3. Consider the previous exercise and set up a simple web server to guess a secret word. The word is the name typed in after `localhost:8000/` in the URL.