Merhawi Haile

Data communication

Programming Assignment 1: Programming with Sockets

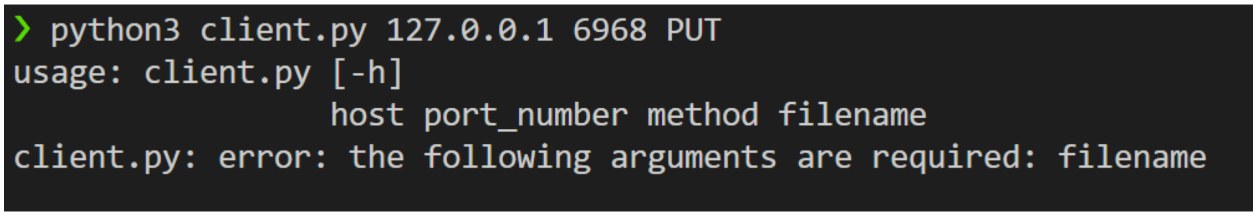
Design and working of the client program:

For the client program, I used the following python libraries: socket, argparse, and sys.

Arguments parsing:

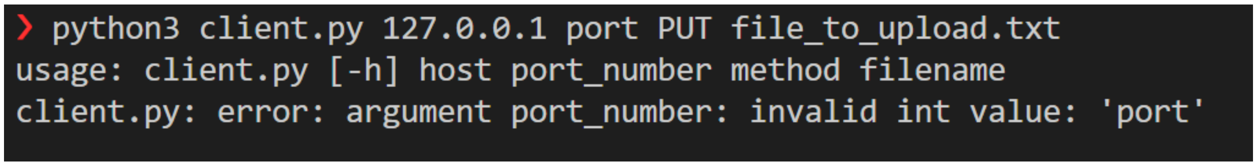
The reason I used argparse is because it can handle missing arguments and throws exception if the user tries to run the program without any required fields.

For example, if we try to run the client program with PUT request without providing the filename, it will throw an error message that the filename argument is required.



```
❯ python3 client.py 127.0.0.1 6968 PUT
usage: client.py [-h]
                 host port_number method filename
client.py: error: the following arguments are required: filename
```

If we try to provide an invalid port number. Same validations will be done for the other required arguments as per the requirements.



```
❯ python3 client.py 127.0.0.1 port PUT file_to_upload.txt
usage: client.py [-h] host port_number method filename
client.py: error: argument port_number: invalid int value: 'port'
```

Connecting to the server:

For socket connection, first we initialize the socket and then connect to the provided host and port. After connecting to the host, we send the request to the server.

If it is **GET** request, then we form the request body as per the assignment requirement and send the request using the ***sendall*** method of the socket object, and we wait for the response data from the server. If we get any response, we print it to the console, and close the socket connection.

If it is **PUT** request, then we form the initial request body as per the assignment requirement and send the request using the ***sendall*** method of the socket object, and after that read the file into a file buffer and send the data in chunks (with buffer size of 1024) to the server, and wait for the response from the server, and close the socket connection.

Also, if the client tries to send any requests other than GET or PUT, it will print a message "*Only GET or PUT method is supported!*"

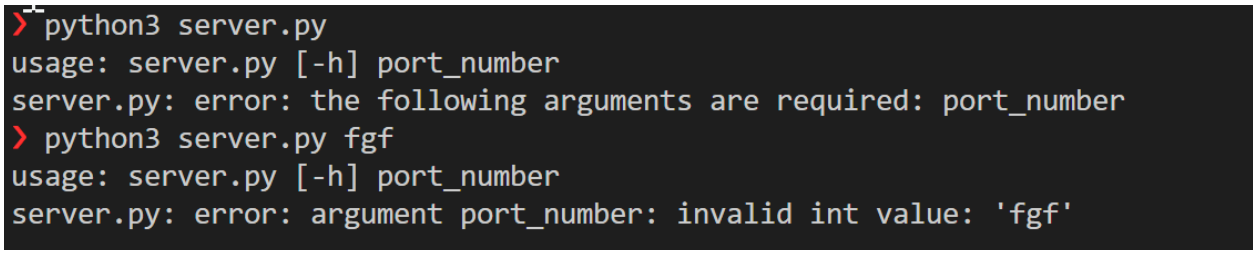
Design and working of the server program:

For the server program, I used the following python libraries: socket, and argparse.

Arguments parsing:

For the reason as in the client program, I used argparse is because it can handle missing arguments and throws exception if the user tries to run the program without any required fields.

For example, if we try to run the server program without providing the port or providing invalid ports, it will throw an error message.



```
❯ python3 server.py
usage: server.py [-h] port_number
server.py: error: the following arguments are required: port_number
❯ python3 server.py fgf
usage: server.py [-h] port_number
server.py: error: argument port_number: invalid int value: 'fgf'
```

Binding the host and port:

After initializing the socket connection, we must bind the provided host and port, and if the port is unavailable, throw the error message.

Handling the incoming connections from the client:

After the socket is ready on the provided host and port, we wait for the incoming connections using the **listen** method of the socket object.

If it gets any connection, the server accepts that connection and receive the request sent by the client and handle the request accordingly.

For **GET** request, we parse the filename from the request and try to read that file from the server directory and send the file data back to the client. If the file is not present in the server, then throw a 404 NOT FOUND error. Also close the socket connection in the end.

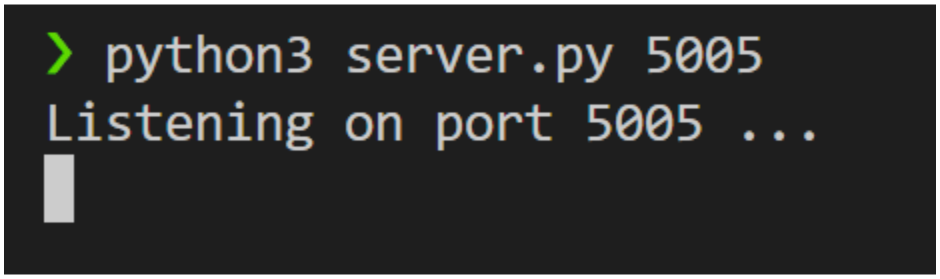
For **PUT** request, we parse the filename from the request and create an empty file with the same name in the *Received_Files* directory inside the server directory, and write the received file content to that file, and close the socket connection. After the file is received successfully, send the *200 OK File Created* message back to the client.

After a successful processing of a request, the server will continue to listen for other connections in a loop until the user interrupts the program.

Also, if the server receives any requests other than GET or PUT, it will print a message "*Only GET or PUT method is supported!*"

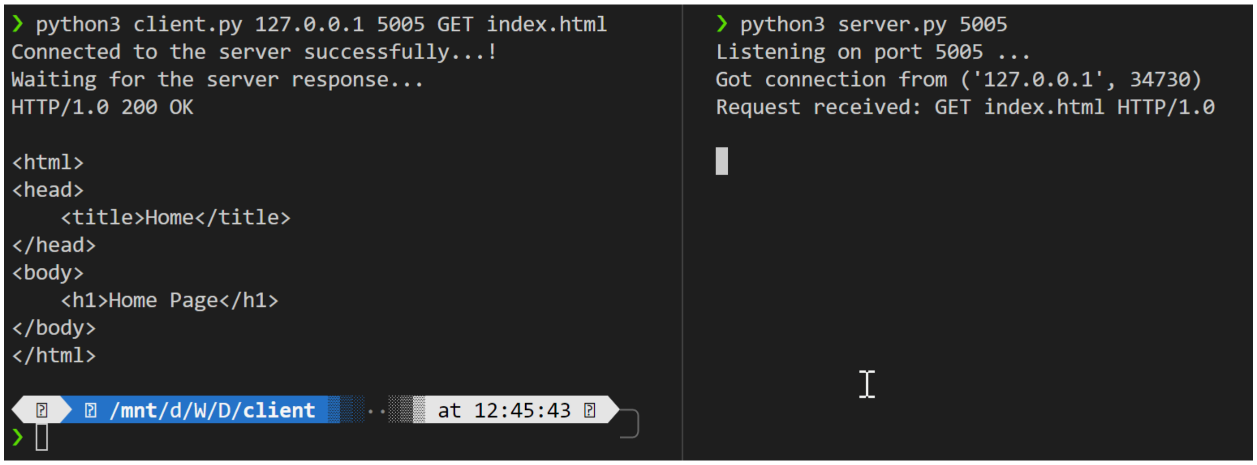
Testing the program

Running the server



```
❯ python3 server.py 5005
Listening on port 5005 ...
```

Sending a GET request from the client



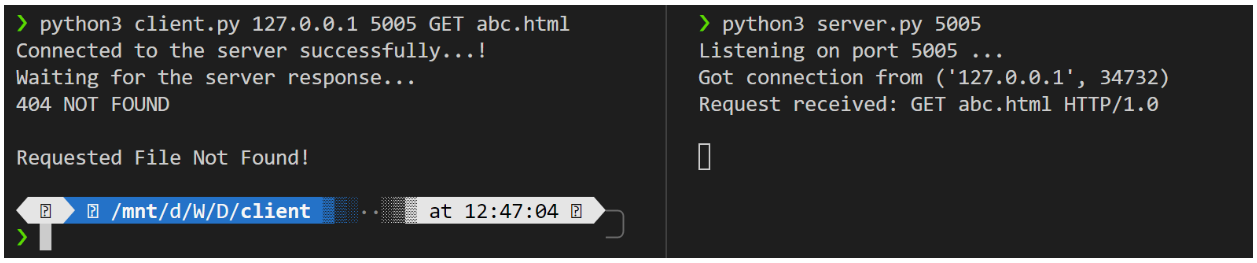
```
❯ python3 client.py 127.0.0.1 5005 GET index.html
Connected to the server successfully...!
Waiting for the server response...
HTTP/1.0 200 OK

<html>
<head>
    <title>Home</title>
</head>
<body>
    <h1>Home Page</h1>
</body>
</html>

 /mnt/d/W/D/client  at 12:45:43
❯
```

```
❯ python3 server.py 5005
Listening on port 5005 ...
Got connection from ('127.0.0.1', 34730)
Request received: GET index.html HTTP/1.0
```

Sending a GET request from the client with a filename that is not present in the server.



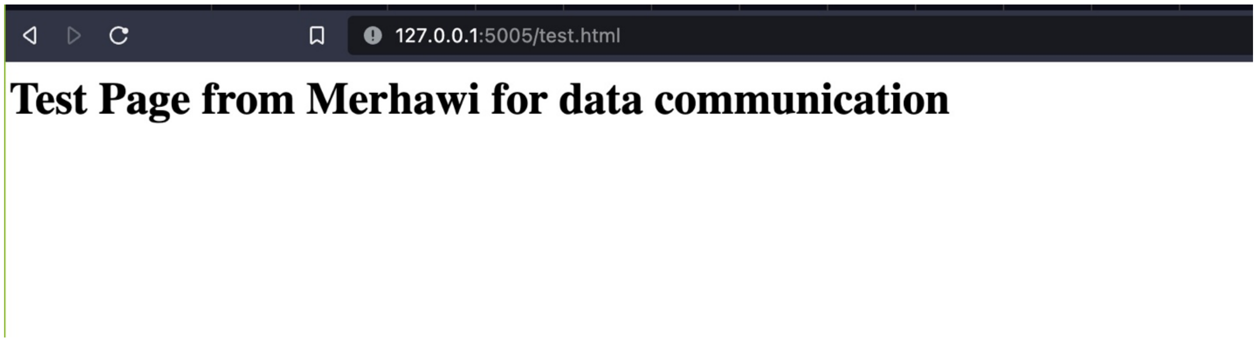
```
❯ python3 client.py 127.0.0.1 5005 GET abc.html
Connected to the server successfully...!
Waiting for the server response...
404 NOT FOUND

Requested File Not Found!

 /mnt/d/W/D/client  at 12:47:04
❯
```

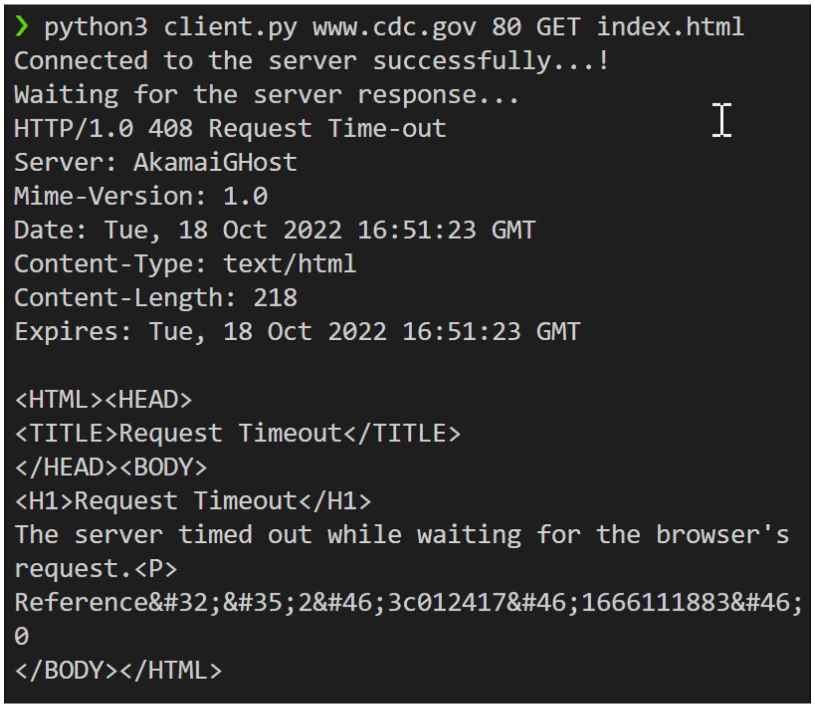
```
❯ python3 server.py 5005
Listening on port 5005 ...
Got connection from ('127.0.0.1', 34732)
Request received: GET abc.html HTTP/1.0
```

Sending a GET request from the browser



Test Page from Merhawi for data communication

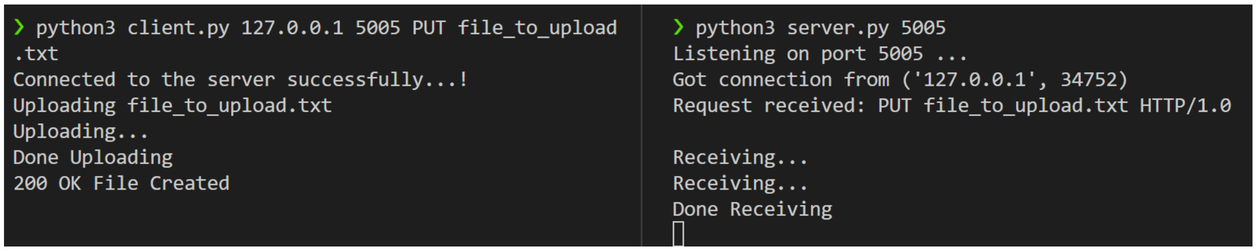
Sending a GET request to an external host



```
❯ python3 client.py www.cdc.gov 80 GET index.html
Connected to the server successfully...!
Waiting for the server response...
HTTP/1.0 408 Request Time-out
Server: AkamaiGHost
Mime-Version: 1.0
Date: Tue, 18 Oct 2022 16:51:23 GMT
Content-Type: text/html
Content-Length: 218
Expires: Tue, 18 Oct 2022 16:51:23 GMT

<HTML><HEAD>
<TITLE>Request Timeout</TITLE>
</HEAD><BODY>
<H1>Request Timeout</H1>
The server timed out while waiting for the browser's
request.<P>
Reference&#32;&#35;2&#46;3c012417&#46;1666111883&#46;
0
</BODY></HTML>
```

Sending a PUT request from the client.



```
❯ python3 client.py 127.0.0.1 5005 PUT file_to_upload
.txt
Connected to the server successfully...!
Uploading file_to_upload.txt
Uploading...
Done Uploading
200 OK File Created
```

```
❯ python3 server.py 5005
Listening on port 5005 ...
Got connection from ('127.0.0.1', 34752)
Request received: PUT file_to_upload.txt HTTP/1.0

Receiving...
Receiving...
Done Receiving
```

Program design tradeoffs:

1. Simplicity: This program might not be the flexible, but it is simple to use.

Possible improvements:

1. Fault tolerance on the unexpected inputs and larger file sizes.
2. Connection pooling, no direct connection upon instantiation