Comp 2322 Computer Networking

Project: Multi-thread Web Server

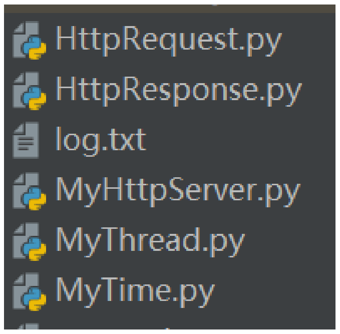
21099695d

LU Zhoudao

Design:

Main codes:

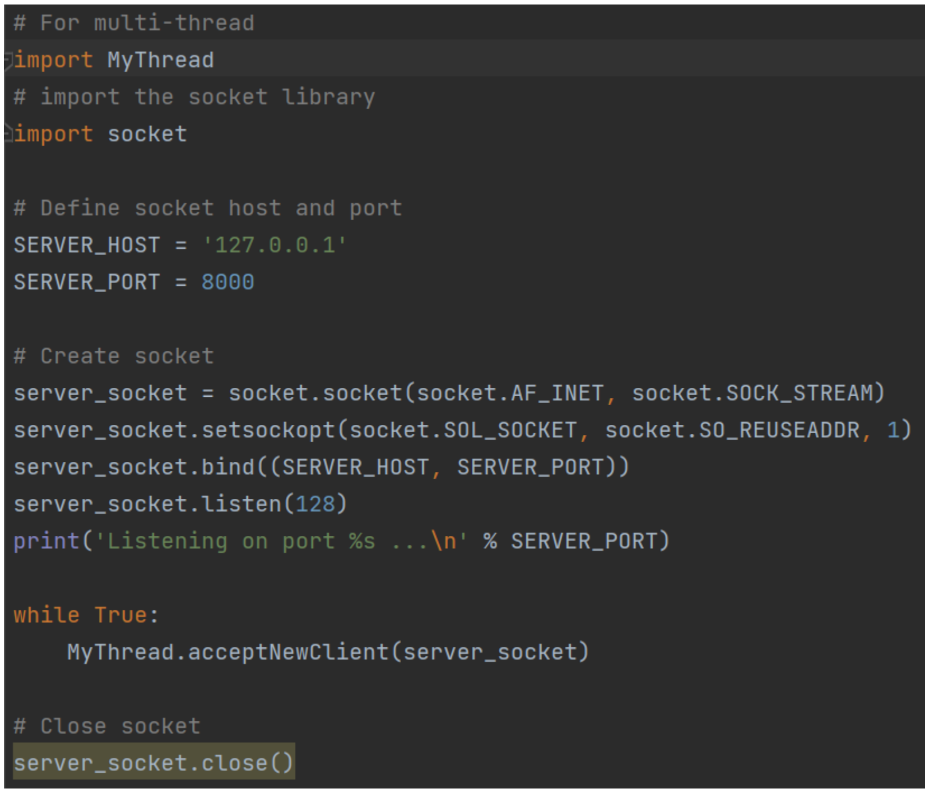
My main codes can be divided into 5 files, 'HTTPRequest.py', 'HTTPResonse.py', 'MyHttpServer.py', 'MyThread.py', and 'MyTime.py'.



```
HttpRequest.py
HttpResponse.py
log.txt
MyHttpServer.py
MyThread.py
MyTime.py
```

a) MyHttpServer.py:

If users want to run these codes, they should run 'MyHTTPServer.py' first. This file is to set the hostname and host port, and to build the connection with clients by TCP.



```python
# For multi-thread
import MyThread
# import the socket library
import socket

# Define socket host and port
SERVER_HOST = '127.0.0.1'
SERVER_PORT = 8000

# Create socket
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
server_socket.bind((SERVER_HOST, SERVER_PORT))
server_socket.listen(128)
print('Listening on port %s ...\n' % SERVER_PORT)

while True:
    MyThread.acceptNewClient(server_socket)

# Close socket
server_socket.close()
```

b) MyThread.py:

This project asks us to build a multi-threaded Web server. I build 'MyThread.py' to deal with multi-threads better.

1) acceptNewClient (server_socket):

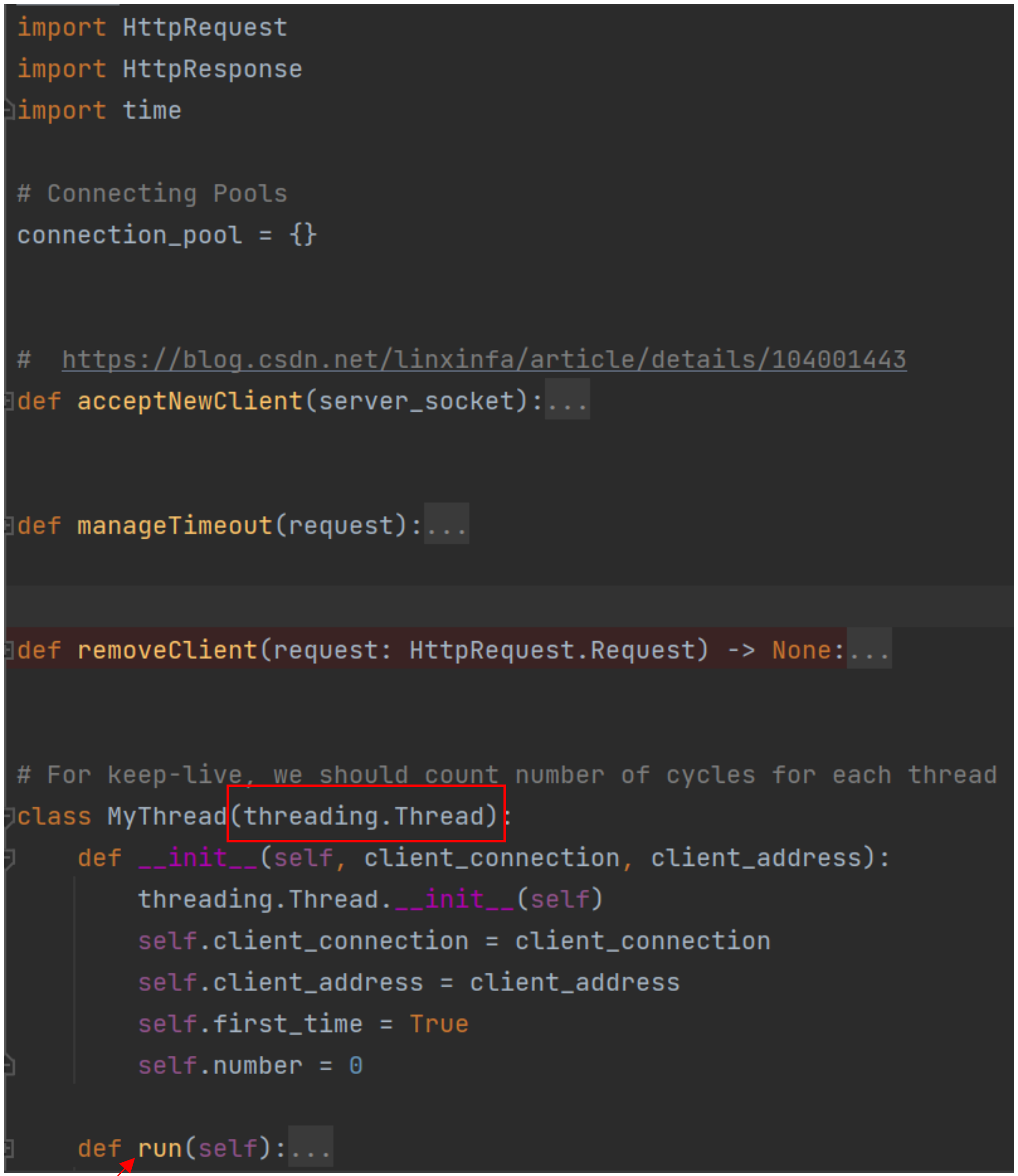
This function is used to build new connections with new clients.

2) *manageTimeout (request):*

This function is used to manage the timeout mechanisms if the connection is keep-alive. '*manageTimeout (request)*' will monitor if the time limit has been reached. The details of the codes will be shown in 'Implementation'.

3) RemoveClient (request):

When some situations happen, like time is out or the maximum number of connections is achieved, we should remove the TCP connections between the client and the server. This function will help me to do this job.



```
import HttpRequest
import HttpResponse
import time

# Connecting Pools
connection_pool = {}


#  https://blog.csdn.net/linxinfa/article/details/104001443
def acceptNewClient(server_socket):...


def manageTimeout(request):...


def removeClient(request: HttpRequest.Request) -> None:...


# For keep-live, we should count number of cycles for each thread
class MyThread(threading.Thread):
    def __init__(self, client_connection, client_address):
        threading.Thread.__init__(self)
        self.client_connection = client_connection
        self.client_address = client_address
        self.first_time = True
        self.number = 0

    def run(self):...
```

4) *Class: MyThread:*

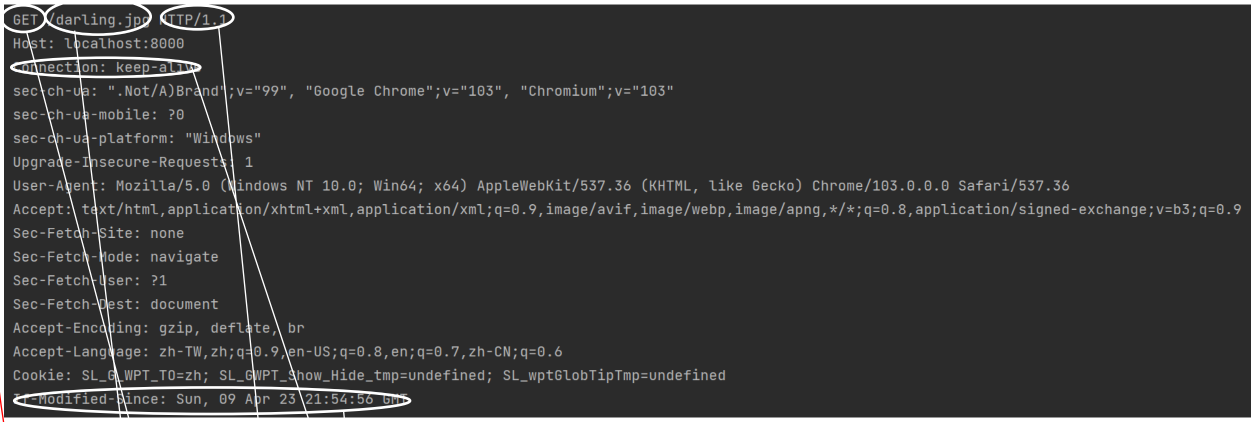
'MyThread' is a class that inherits 'thread.Thread'. Every time builds a new connection, a new

'MyThread' will run to maintain interaction with customers respectively. This is a good way to control the connection of different clients and introduce collisions.

c) *HTTPRequest.py:*

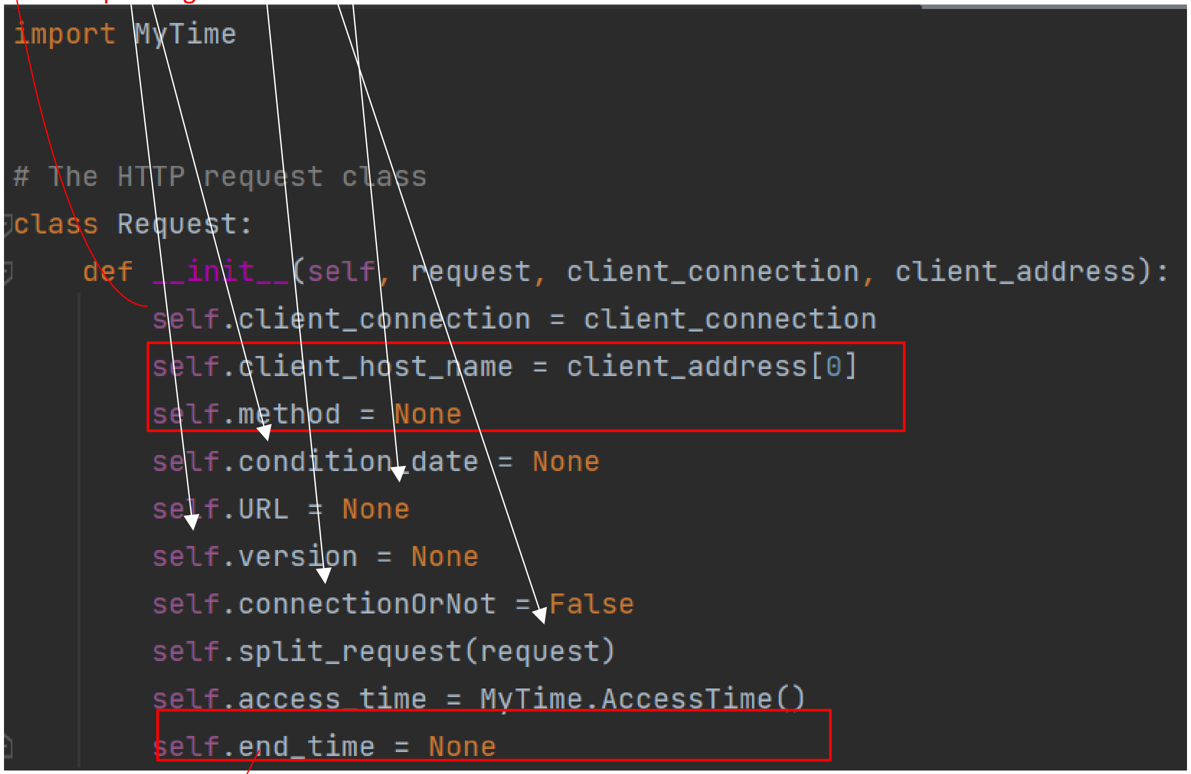
When the clients send the HTTP requests, this file will help to handle these messages. The key point is that the class 'Request'. After setting the attributes of the 'Request' objects, it is easier to call them to use in the follow-up process. For example, "method" is to store the method types like GET, HEAD. Object-oriented programming can make me know the specific methods quickly.

This is a sample of a request from a browser. The corresponding stored values are as follows:



```
GET /darling.jpg HTTP/1.1
Host: localhost:8000
Connection: keep-alive
sec-ch-ua: ".Not/A)Brand";v="99", "Google Chrome";v="103", "Chromium";v="103"
sec-ch-ua-mobile: ?0
sec-ch-ua-platform: "Windows"
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/103.0.0.0 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
Sec-Fetch-Site: none
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Accept-Encoding: gzip, deflate, br
Accept-Language: zh-TW,zh;q=0.9,en-US;q=0.8,en;q=0.7,zh-CN;q=0.6
Cookie: SL_G_WPT_TO=zh; SL_GWPT_Show_Hide_tmp=undefined; SL_wptGlobTipTmp=undefined
If-Modified-Since: Sun, 09 Apr 23 21:54:56 GMT
```

This is from 'run()', every time the server receives the messages, it will record the corresponding client connection.

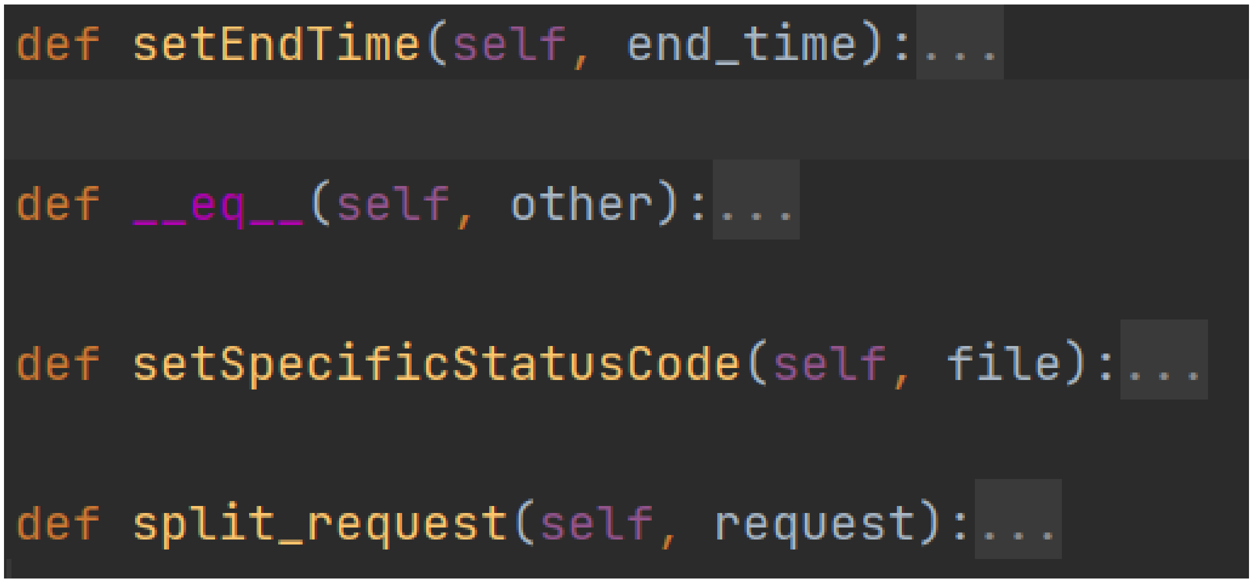


```python
import MyTime


# The HTTP request class
class Request:
    def __init__(self, request, client_connection, client_address):
        self.client_connection = client_connection
        self.client_host_name = client_address[0]
        self.method = None
        self.condition_date = None
        self.URL = None
        self.version = None
        self.connectionOrNot = False
        self.split_request(request)
        self.access_time = MyTime.AccessTime()
        self.end_time = None
```

This is to set the access time of this request.

1) setEndTime(self,end_time):
This function is to set the ‘end_time’. ‘end_time’ is for the timeout of the keep-alive connection. After the server send the response, the connection remain open until time is out. When the time is beyond the ‘end_time’, the open connection should be closed.

2) __eq__(self, other):
Thus function override __eq__(). It is to compare whether two requests are equal or not.



```
def setEndTime(self, end_time):...

def __eq__(self, other):...

def setSpecificStatusCode(self, file):...

def split_request(self, request):...
```

3) setSpecificStatusCode(self, file):
To handle the error like web page not being found, I prepare simple HTML files for each status codes except “200 OK”. This function is to set the HTML files by setting the URL.

4) Split_request(self, request):
This function is used to split the string of the request and set the attributes of a “Request” object. For example, set these kinds of Nonetype attributes like ‘version’ and ‘method’.

d) *HTTPResponse.py:*
When the server receivesthe request, the server should analyze the information and reply a correct response.

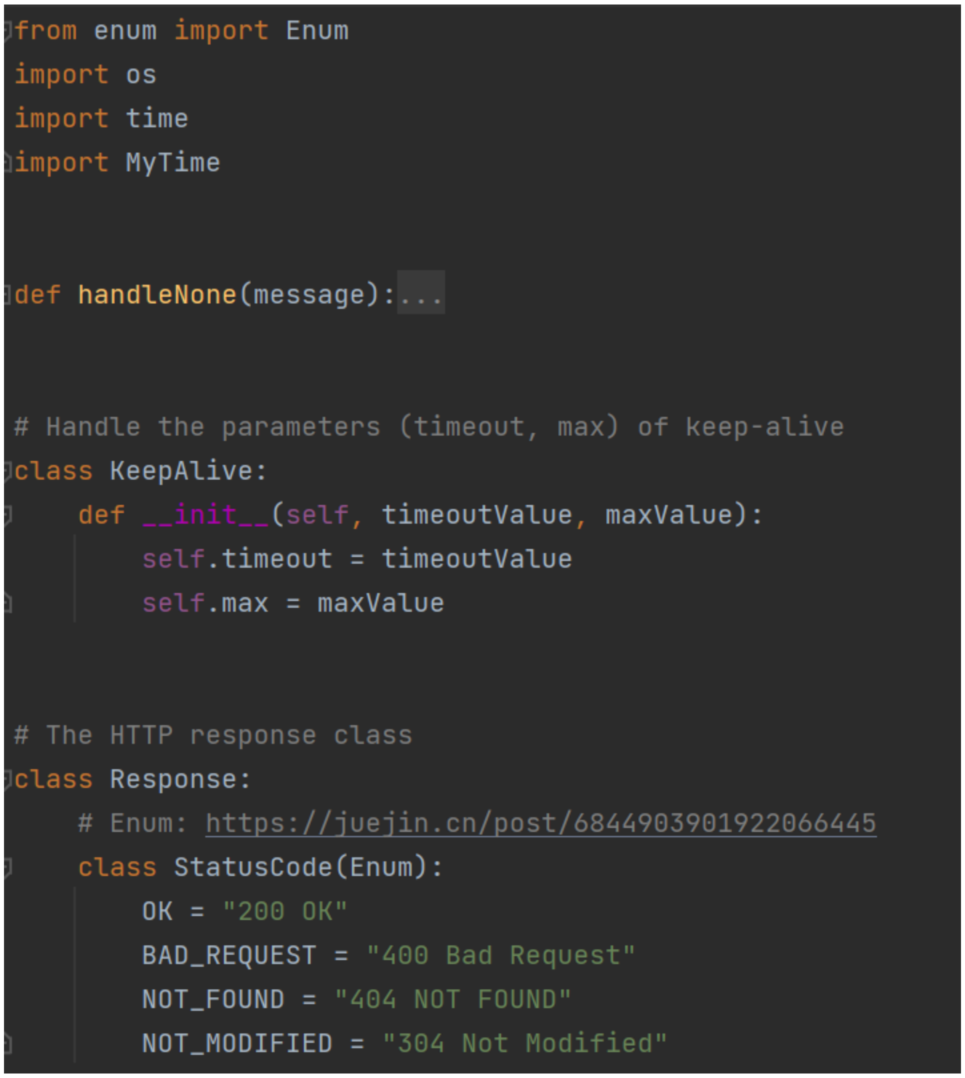
1) handNone(message):
This function is to record “None” when the request file s None. This is to help keep no errors when updating the log file.

2) Class: KeepAlive:
Because when the connection is open, the keep-alive need two parameters, timeout and

max. I set a specialized class to manage timeout and max better.

The key to this file is class "Response". For each request, the system will set a corresponding 'Response' object.

3) *class: StatusCode:*
I have an enum class inside 'Response' to store the status codes better because this project just needs4 codes. This enum class is easier to update new statues codes if possible.



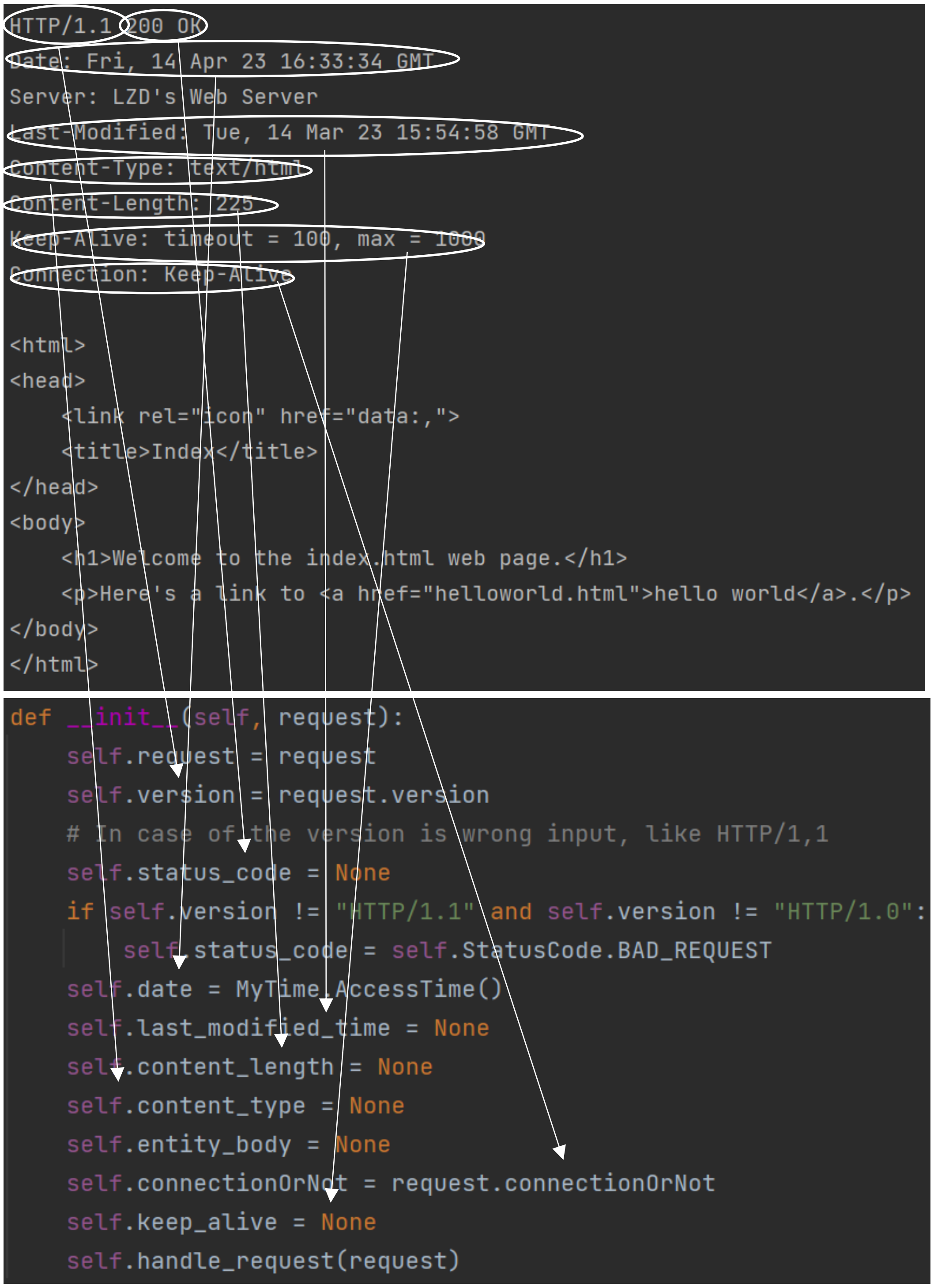
```python
from enum import Enum
import os
import time
import MyTime


def handleNone(message):...


# Handle the parameters (timeout, max) of keep-alive
class KeepAlive:
    def __init__(self, timeoutValue, maxValue):
        self.timeout = timeoutValue
        self.max = maxValue


# The HTTP response class
class Response:
    # Enum: https://juejin.cn/post/6844903901922066445
    class StatusCode(Enum):
        OK = "200 OK"
        BAD_REQUEST = "400 Bad Request"
        NOT_FOUND = "404 NOT FOUND"
        NOT_MODIFIED = "304 Not Modified"
```

4) *setWrongFile(self):*
To handle the error like web page not being found, I prepare simple HTML files for each status codes except "200 OK". This function is to set the HTML file.

5) *Hand_ exension(self):*
The files usually have extensions like 'jpg', 'png', and 'html'. The function is to rectieve the

extension and set the 'content_type' like 'text/html' and 'image/jpg'.

This is a sample of a response from the server. The corresponding stored values are as follows:



```
HTTP/1.1 200 OK
Date: Fri, 14 Apr 23 16:33:34 GMT
Server: LZD's Web Server
Last-Modified: Tue, 14 Mar 23 15:54:58 GMT
Content-Type: text/html
Content-Length: 225
Keep-Alive: timeout = 100, max = 1000
Connection: Keep-Alive

<html>
<head>
    <link rel="icon" href="data:,">
    <title>Index</title>
</head>
<body>
    <h1>Welcome to the index.html web page.</h1>
    <p>Here's a link to <a href="helloworld.html">hello world</a>.</p>
</body>
</html>
```

```python
def __init__(self, request):
    self.request = request
    self.version = request.version
    # In case of the version is wrong input, like HTTP/1,1
    self.status_code = None
    if self.version != "HTTP/1.1" and self.version != "HTTP/1.0":
        self.status_code = self.StatusCode.BAD_REQUEST
    self.date = MyTime.AccessTime()
    self.last_modified_time = None
    self.content_length = None
    self.content_type = None
    self.entity_body = None
    self.connectionOrNot = request.connectionOrNot
    self.keep_alive = None
    self.handle_request(request)
```

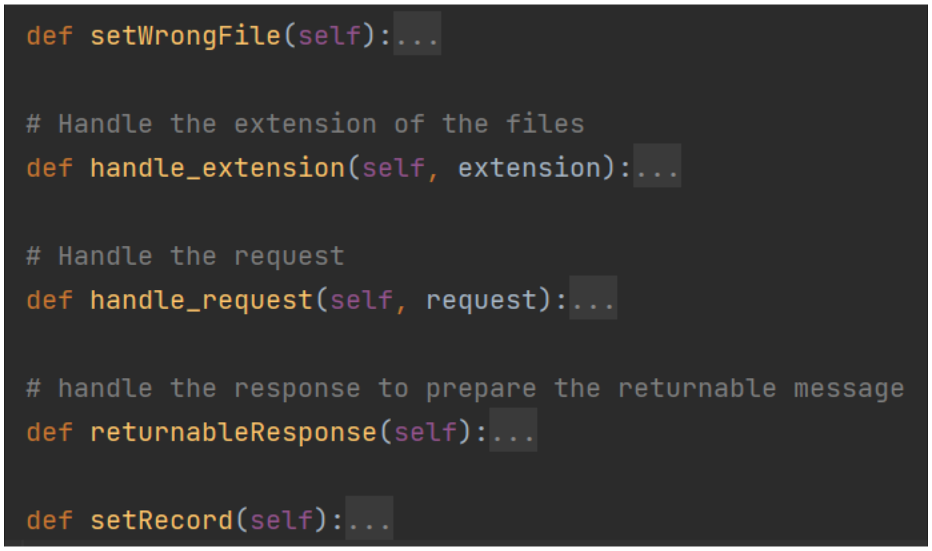
6) *returnableResponse(self):*

This function is to get a correct form responsehe sample response.

7) setRecord(self):

This function is to get the correct form to record in the log file, like

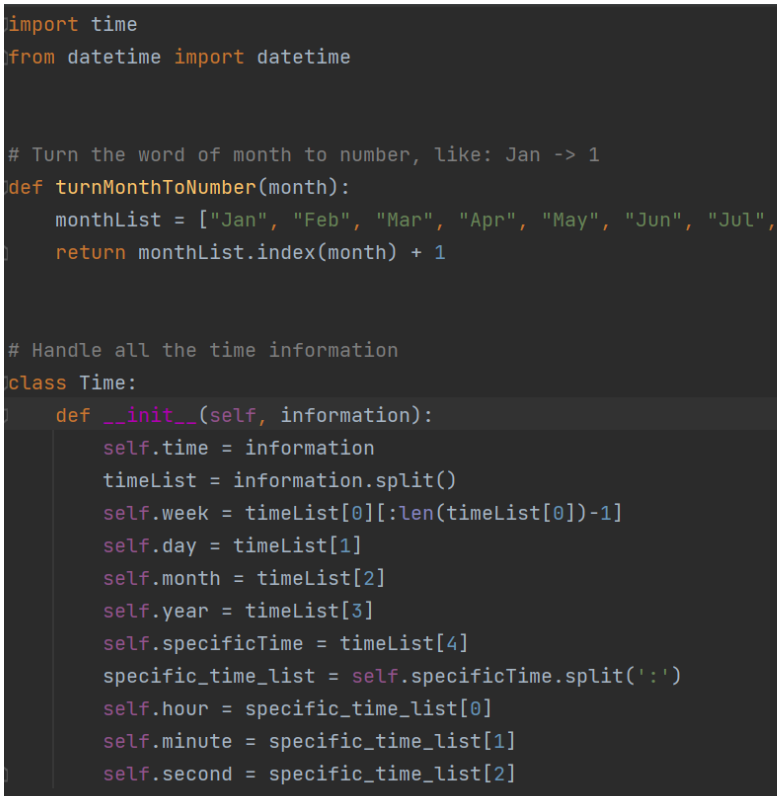
```
[127.0.0.1,Fri, 14 Apr 23 16:33:34,index.html,200 OK]
```



```python
def setWrongFile(self):...

# Handle the extension of the files
def handle_extension(self, extension):...

# Handle the request
def handle_request(self, request):...

# handle the response to prepare the returnable message
def returnableResponse(self):...

def setRecord(self):...
```

e) *MyTime.py:*

This file is to handle the information about the time specialty. For example, the 'date' and 'lat-modified-date' of the response and the 'if-science' of the request.



```python
import time
from datetime import datetime


# Turn the word of month to number, like: Jan -> 1
def turnMonthToNumber(month):
    monthList = ["Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul",
    return monthList.index(month) + 1


# Handle all the time information
class Time:
    def __init__(self, information):
        self.time = information
        timeList = information.split()
        self.week = timeList[0][:len(timeList[0])-1]
        self.day = timeList[1]
        self.month = timeList[2]
        self.year = timeList[3]
        self.specificTime = timeList[4]
        specific_time_list = self.specificTime.split(':')
        self.hour = specific_time_list[0]
        self.minute = specific_time_list[1]
        self.second = specific_time_list[2]
```

1)turnMonthToNumber(month):

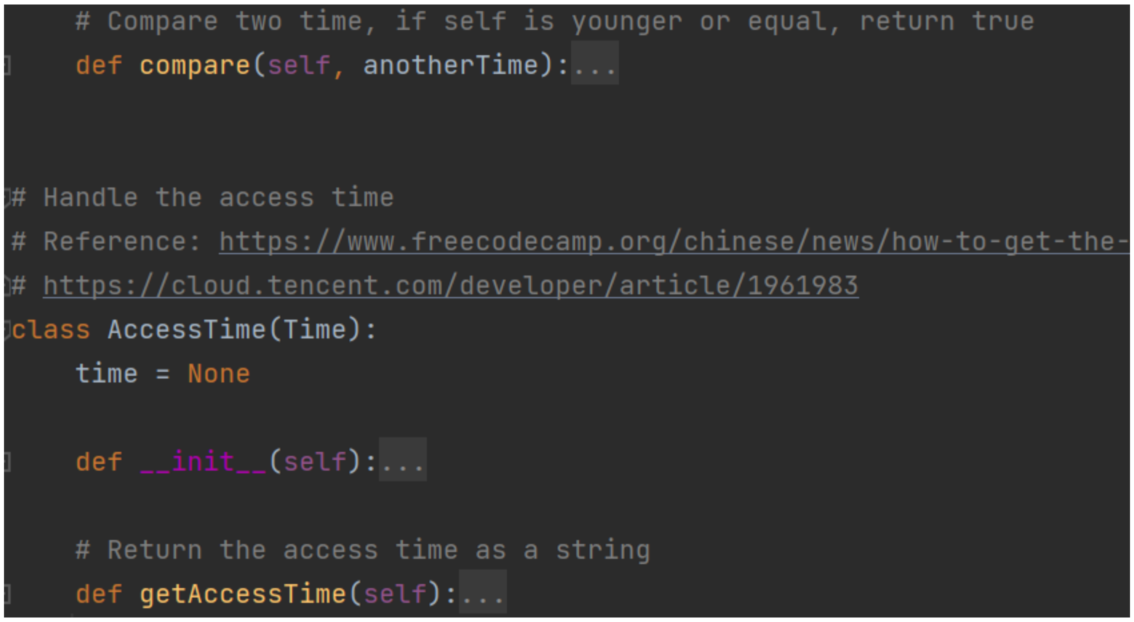
This function is to turn the textual month to the number, like 'Jan'-> 1

Inside class Time:

2) compare(self,anotherTime):

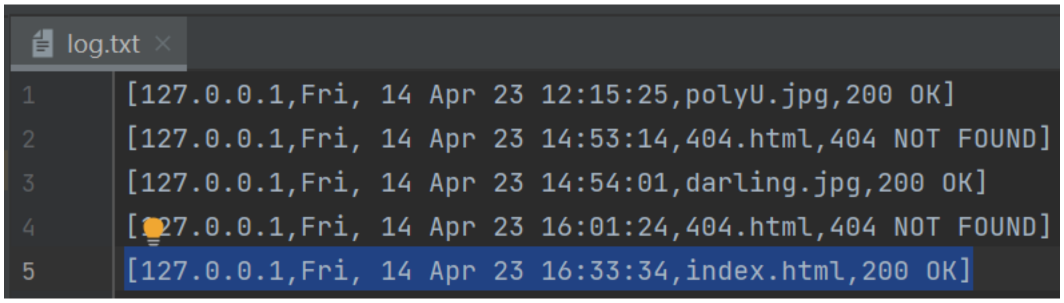
This function is used to compare the two times which is new and which is old. If self is younger or equal, return true.

Class AccessTime inherits Time. This class is to record the assess time of the request or response. I set it because 'Time' class is to turn characteristic time into different time attributes, but 'AccessTime' is to turn the time the characters.



```
    # Compare two time, if self is younger or equal, return true
    def compare(self, anotherTime):...


# Handle the access time
# Reference: https://www.freecodecamp.org/chinese/news/how-to-get-the-
# https://cloud.tencent.com/developer/article/1961983
class AccessTime(Time):
    time = None

    def __init__(self):...

    # Return the access time as a string
    def getAccessTime(self):...
```

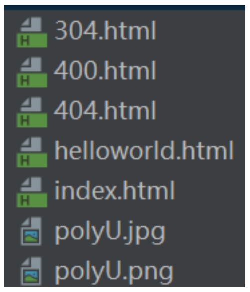
log:
There is a log file to record statistics of the client requests. Each record includes IP address, access time, requested file name and response type for each record.



log.txt

```
[127.0.0.1,Fri, 14 Apr 23 12:15:25,polyU.jpg,200 OK]
[127.0.0.1,Fri, 14 Apr 23 14:53:14,404.html,404 NOT FOUND]
[127.0.0.1,Fri, 14 Apr 23 14:54:01,darling.jpg,200 OK]
[127.0.0.1,Fri, 14 Apr 23 16:01:24,404.html,404 NOT FOUND]
[127.0.0.1,Fri, 14 Apr 23 16:33:34,index.html,200 OK]
```

htdocs:
I put all requested files in this file. ‘index.html’ and ‘helloworld.html’ is the examples of text files; ‘polyU.jpg’ and ‘polyU.png’ are the examples of the image files.



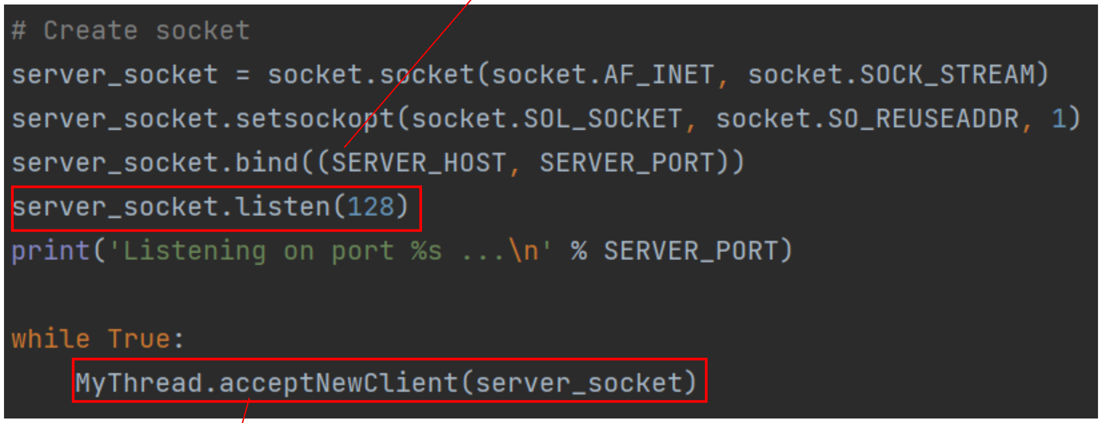
304.html
400.html
404.html
helloworld.html
index.html
polyU.jpg
polyU.png

‘304.html’ will be sent if ‘304 Not Modified’ happens. ‘400.html’ and ‘404.html’ are similar.

Implementation:

a) MyHttpServer.py:

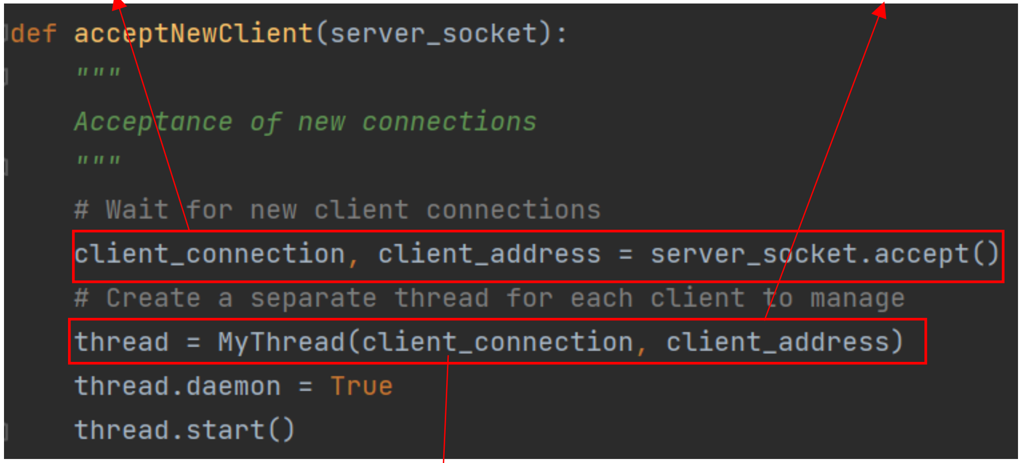
This idea is from 'Lab5-HTTPSocketProgramming' to use socket library to build connection, so I don't explain too much here. However, 128 specifies the maximum number of connections the operating system can hang before a connection is rejected. Because we should build a multi-threaded Webserver so I set this number to 128.



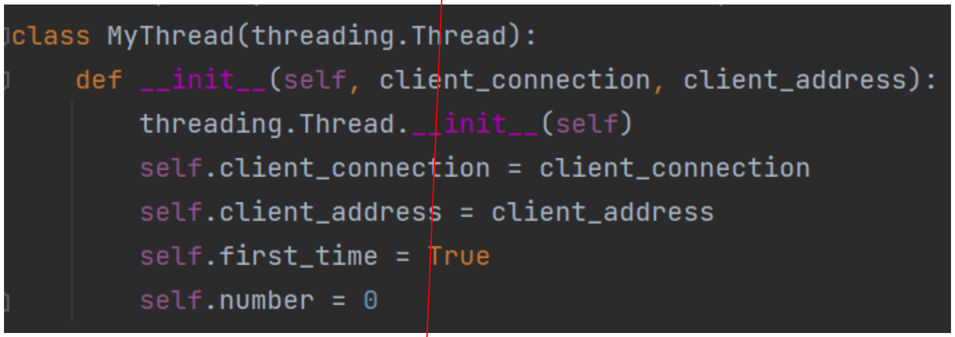
```
# Create socket
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
server_socket.bind((SERVER_HOST, SERVER_PORT))
server_socket.listen(128)
print('Listening on port %s ...\n' % SERVER_PORT)

while True:
    MyThread.acceptNewClient(server_socket)
```

b) MyThread.py:

The server will keep open to wait the client's request to build new connection. 'acceptNewClient(server_socket)' will help to do this job. As long as the server receives new connection, it will create a new thread to manage this connection in this TCP.



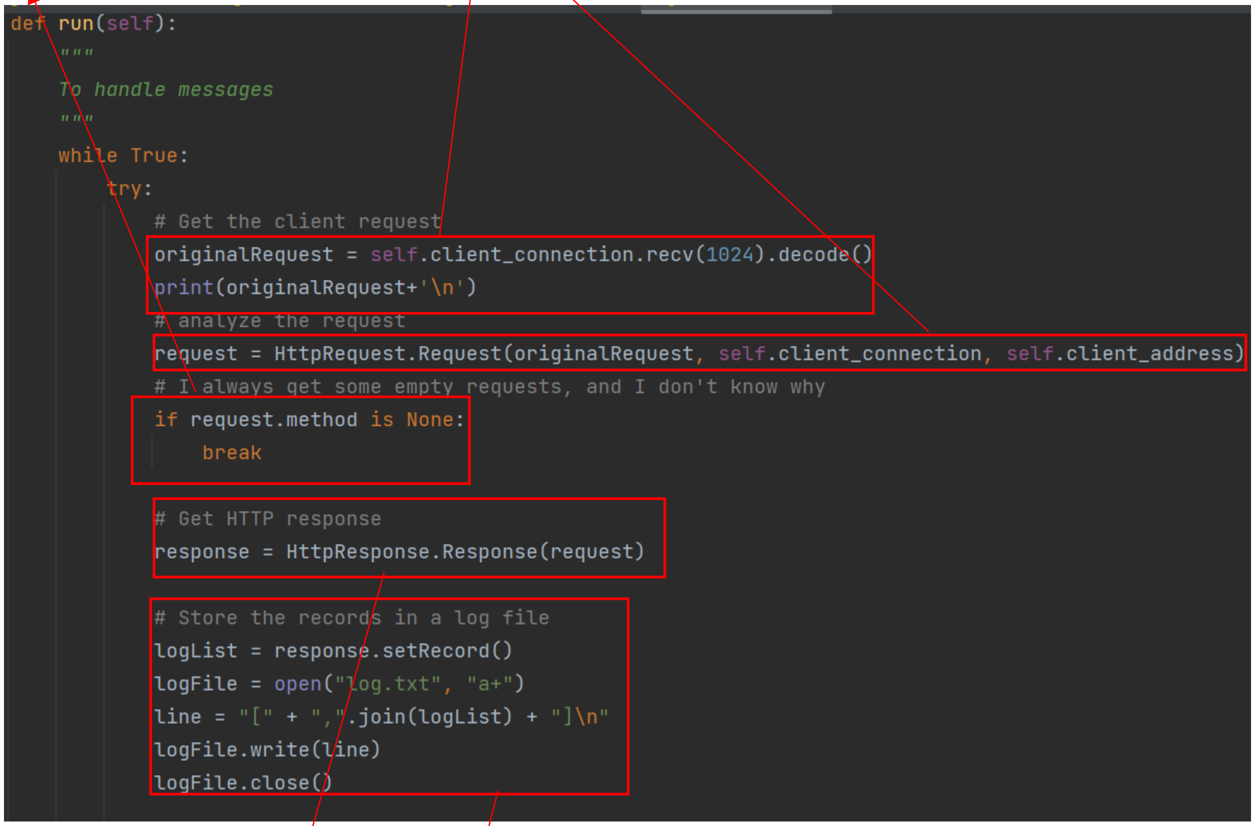
```
def acceptNewClient(server_socket):
    """
    Acceptance of new connections
    """
    # Wait for new client connections
    client_connection, client_address = server_socket.accept()
    # Create a separate thread for each client to manage
    thread = MyThread(client_connection, client_address)
    thread.daemon = True
    thread.start()
```



```
class MyThread(threading.Thread):
    def __init__(self, client_connection, client_address):
        threading.Thread.__init__(self)
        self.client_connection = client_connection
        self.client_address = client_address
        self.first_time = True
        self.number = 0
```

Each time when 'MyThread' is called, the overridden function 'run()' will run automatically. Because the connection may be kept open, we use 'while True'. At first, the thread will wait the client to send the request, and the sever will decode it. I choose to print the original request to see it clearly. Then the class 'Request' will deal with this message. The details will be shown later.

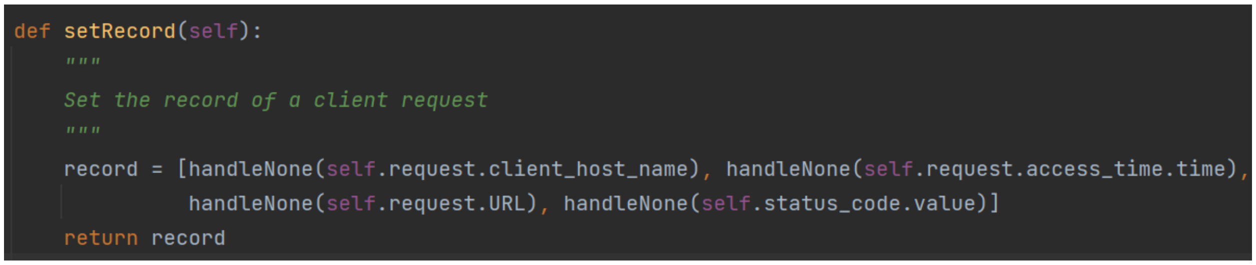
This is to handle some strange situations.



```python
def run(self):
    """
    To handle messages
    """
    while True:
        try:
            # Get the client request
            originalRequest = self.client_connection.recv(1024).decode()
            print(originalRequest+'\n')
            # analyze the request
            request = HttpRequest.Request(originalRequest, self.client_connection, self.client_address)
            # I always get some empty requests, and I don't know why
            if request.method is None:
                break

            # Get HTTP response
            response = HttpResponse.Response(request)

            # Store the records in a log file
            logList = response.setRecord()
            logFile = open("log.txt", "a+")
            line = "[" + ",".join(logList) + "]\n"
            logFile.write(line)
            logFile.close()
```

After handling the request, the class 'Response' will be called to set the response. The details will be shown later.

These codes are in order to save the statistics of each request. By using 'setRecord()' to this response, the IP address, access time, requested file name and response type for each

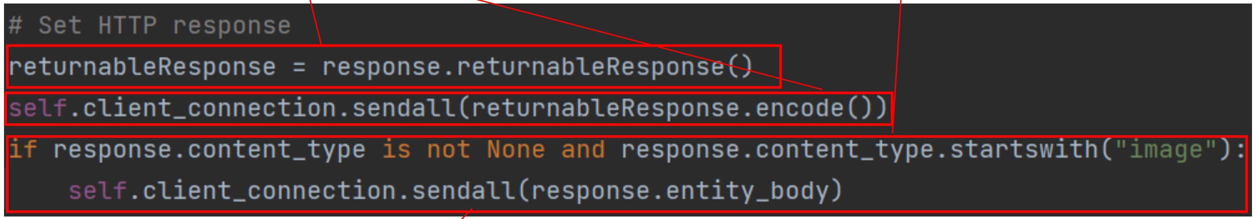


```python
def setRecord(self):
    """
    Set the record of a client request
    """
    record = [handleNone(self.request.client_host_name), handleNone(self.request.access_time.time),
              handleNone(self.request.URL), handleNone(self.status_code.value)]
    return record
```

record will be stored.

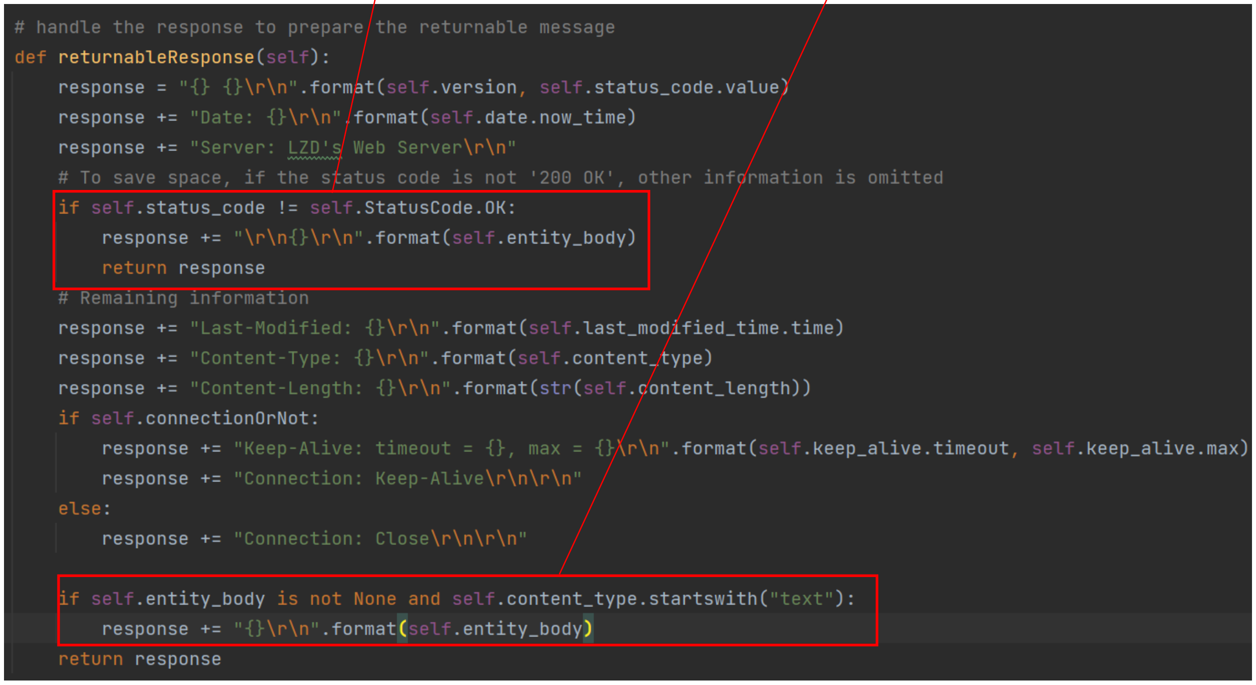
Continuing for 'run()'. 'returnableReponse()' will analyze the response to get a returnable message.

Then the server will send this response by this TCP connection. Because it is textual information so that 'encode()' should be used. However, if the requested file is an image, users shouldn't encode it because it is already binary documents. So the server sent it directly.



```
# Set HTTP response
returnableResponse = response.returnableResponse()
self.client_connection.sendall(returnableResponse.encode())
if response.content_type is not None and response.content_type.startswith("image"):
    self.client_connection.sendall(response.entity_body)
```

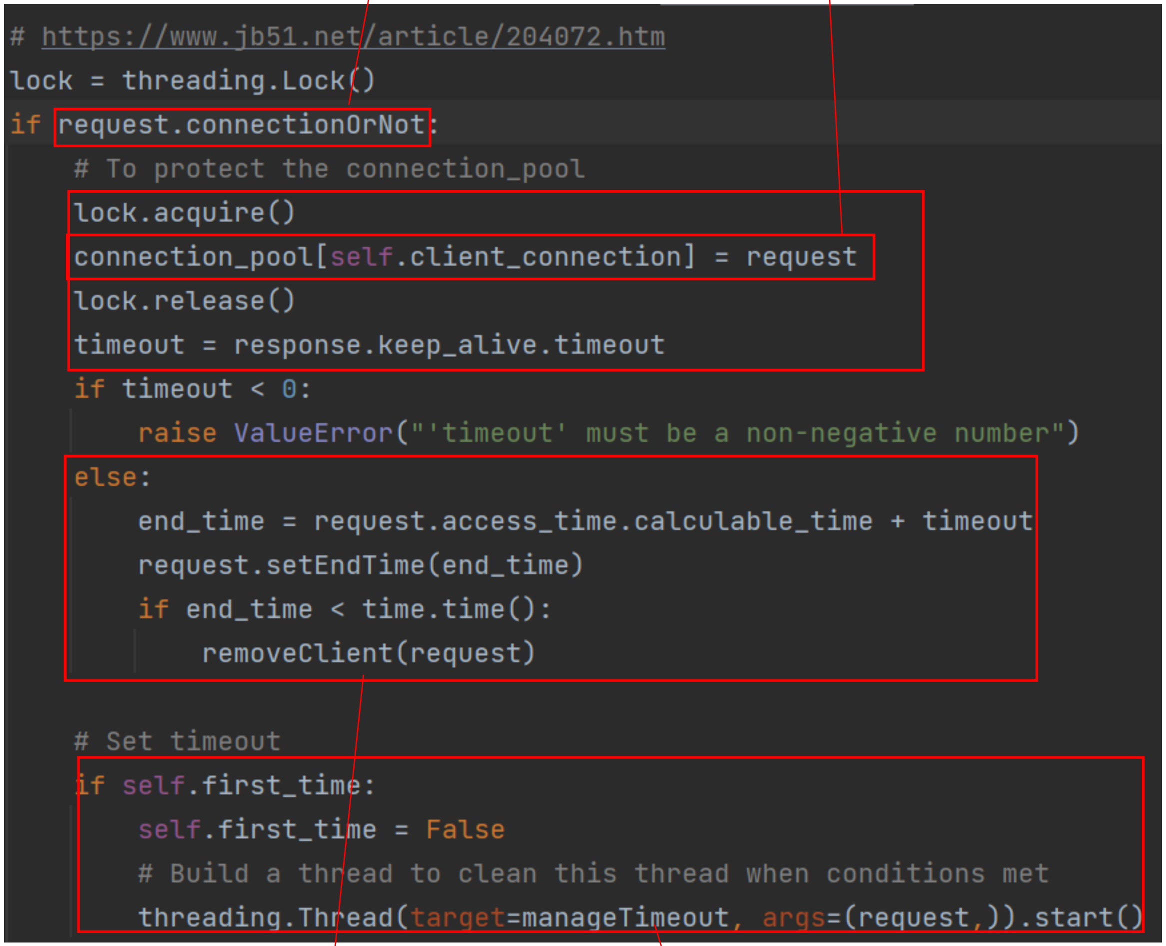
The 'returnableResponse()' is easy to understand. I order these attributes to the correct output. For example, the first line is the version and status code, and the second line is the access date. If the status code is not '200 OK', the remaining information is redundant and wastes space and time, so the response will just be returned. For '200 OK', the remaining information is also added, like 'Last-Modified', 'Content-Type'. Last but not least, if this file is a text file, the form is the same as the response so that we can add it before the response directly. However, if the file is an image, direct sending will lead to a mistake so the system should send it in addition.



```
# handle the response to prepare the returnable message
def returnableResponse(self):
    response = "{} {}\r\n".format(self.version, self.status_code.value)
    response += "Date: {}\r\n".format(self.date.now_time)
    response += "Server: LZD's Web Server\r\n"
    # To save space, if the status code is not '200 OK', other information is omitted
    if self.status_code != self.StatusCode.OK:
        response += "\r\n{}\r\n".format(self.entity_body)
        return response
    # Remaining information
    response += "Last-Modified: {}\r\n".format(self.last_modified_time.time)
    response += "Content-Type: {}\r\n".format(self.content_type)
    response += "Content-Length: {}\r\n".format(str(self.content_length))
    if self.connectionOrNot:
        response += "Keep-Alive: timeout = {}, max = {}\r\n".format(self.keep_alive.timeout, self.keep_alive.max)
        response += "Connection: Keep-Alive\r\n\r\n"
    else:
        response += "Connection: Close\r\n\r\n"

    if self.entity_body is not None and self.content_type.startswith("text"):
        response += "{}\r\n".format(self.entity_body)
    return response
```

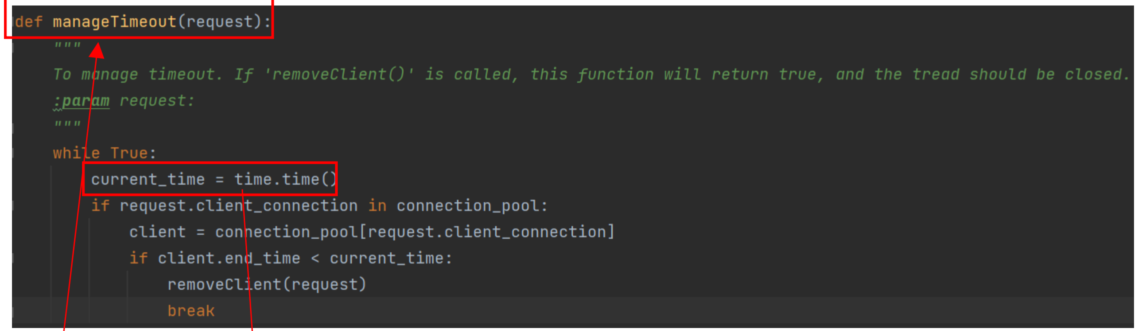
Continuing for 'run()'. If the connection is 'keep-alive', the systems should consider maintaining it by 'timeout' and 'max'. 'connection_pool' is a list to store the open connections. If there is a new connection, the system will add it to this list. The key is the 'client_connection' received by

'acceptNewClient(server_socket)', and the value is the corresponding value.
I set a lock because 'connection_pool' is global and each thread may access it at the same time. 'lock.acquire()' is to add lock before accessing this pool and 'lock.release()' is to release this pool.



```
# https://www.jb51.net/article/204072.htm
lock = threading.Lock()
if request.connectionOrNot:
    # To protect the connection_pool
    lock.acquire()
    connection_pool[self.client_connection] = request
    lock.release()
    timeout = response.keep_alive.timeout
    if timeout < 0:
        raise ValueError("'timeout' must be a non-negative number")
    else:
        end_time = request.access_time.calculable_time + timeout
        request.setEndTime(end_time)
        if end_time < time.time():
            removeClient(request)

    # Set timeout
    if self.first_time:
        self.first_time = False
        # Build a thread to clean this thread when conditions met
        threading.Thread(target=manageTimeout, args=(request,)).start()
```

After each access, the 'end_time' of the request should be set. If the 'end_time' is reached, this connection should be removed. However, this code just runs once for each request but we should check the timeout during the thread. In that case, I build a specific thread for each connection. If the running time is the first time, the system will call the new thread. So there is an attribute named 'first_time' to help manage the timeout.

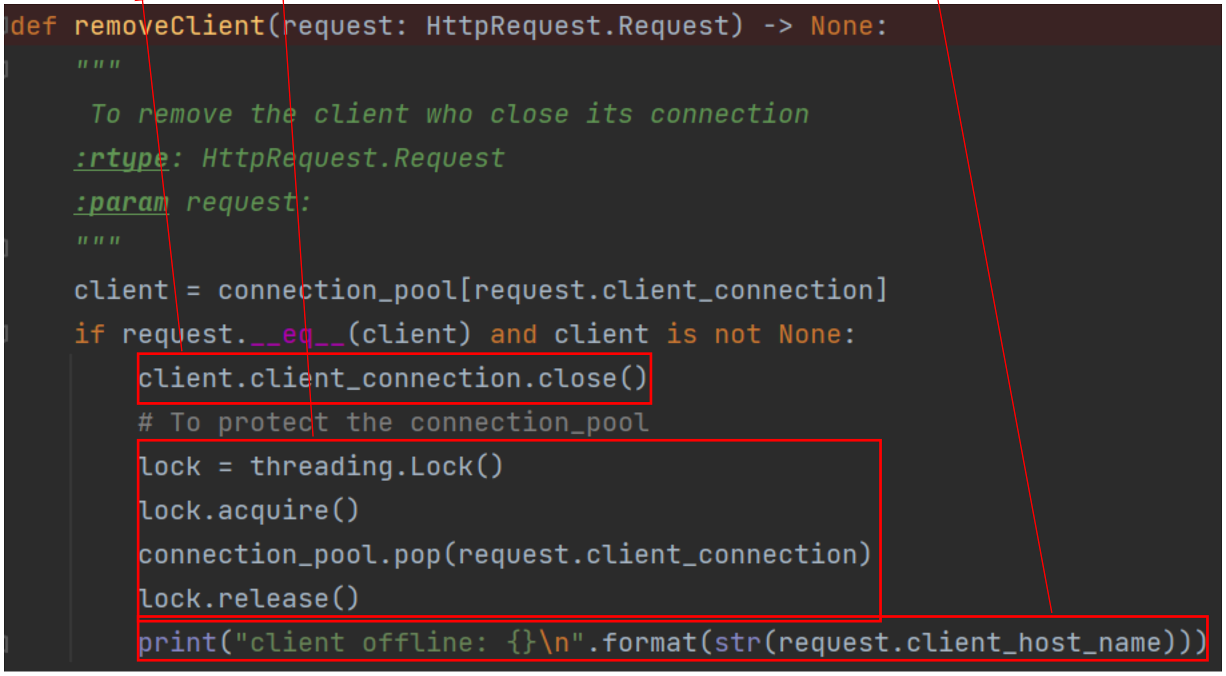


```
def manageTimeout(request):
    """
    To manage timeout. If 'removeClient()' is called, this function will return true, and the tread should be closed.
    :param request:
    """
    while True:
        current_time = time.time()
        if request.client_connection in connection_pool:
            client = connection_pool[request.client_connection]
            if client.end_time < current_time:
                removeClient(request)
                break
```

'manageTimeout(request)' uses 'while' to check whether the 'end_time' is reached. First, the system will retrieve the request from the connection pool. Then checking the time. If timeout, the system will use 'removeClient(request)' to remove the connection.

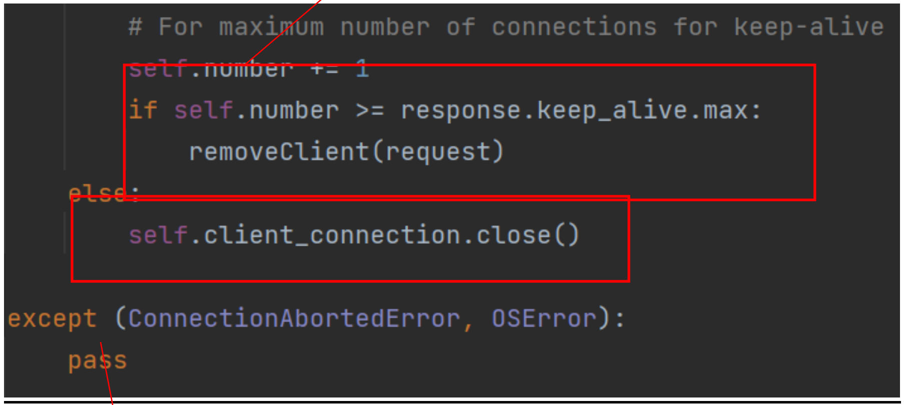
The system uses 'time.time()' to get the time because it is easier to calculate like adding.

I continue to explain 'removeClient(request)'. If the system wants to remove the connection, it should close the connection at first. Because different threads may access 'connection_pool' at the same time, I still choose to add a lock, and remove the information of this TCP connection from the 'connection_pool'. The server should know the offline so "client offline" will be printed out.



```
def removeClient(request: HttpRequest.Request) -> None:
    """
     To remove the client who close its connection
    :rtype: HttpRequest.Request
    :param request:
    """
    client = connection_pool[request.client_connection]
    if request.__eq__(client) and client is not None:
        client.client_connection.close()
        # To protect the connection_pool
        lock = threading.Lock()
        lock.acquire()
        connection_pool.pop(request.client_connection)
        lock.release()
        print("client offline: {}\n".format(str(request.client_host_name)))
```

Continuing for 'run()'. For each TCP connection, we will let the attribute, number add 1 when there is any new request in this connection. If the number is larger than the maximum connection number 'max', we should remove this connection.



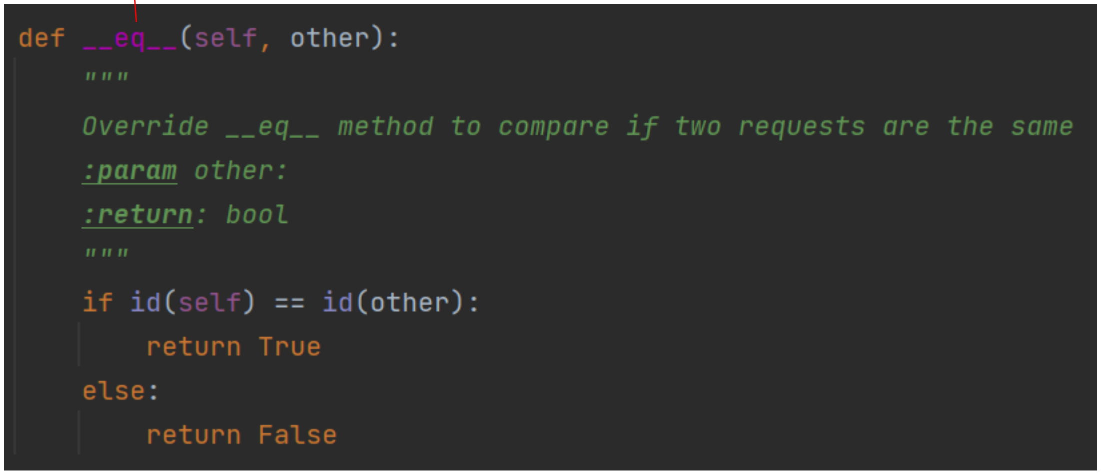
```
        # For maximum number of connections for keep-alive
        self.number += 1
        if self.number >= response.keep_alive.max:
            removeClient(request)
    else:
        self.client_connection.close()

except (ConnectionAbortedError, OSError):
    pass
```

If “Connection: close” is sent in the request. The ‘connectionOrNot’ is False, and the ‘client)connection’ should be closed directly.

c) HttpRequest.py:

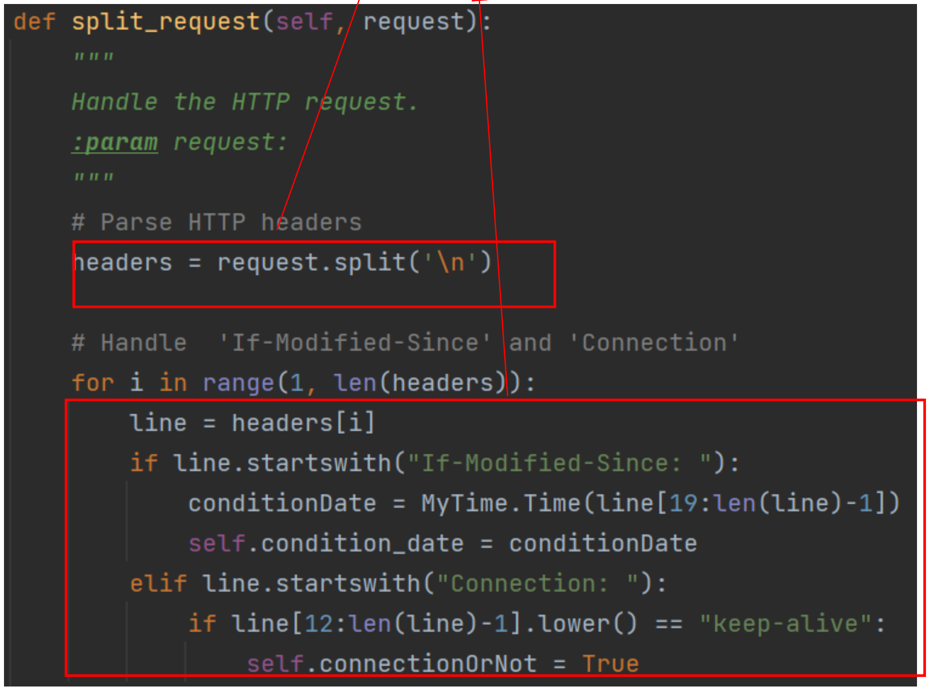
There are still some details should be supplied for ‘HttpRequest.py’.

‘__eq__(self,other)’ is to compare two objects are equal or not. If their ID is the same, we can judge they are the same.



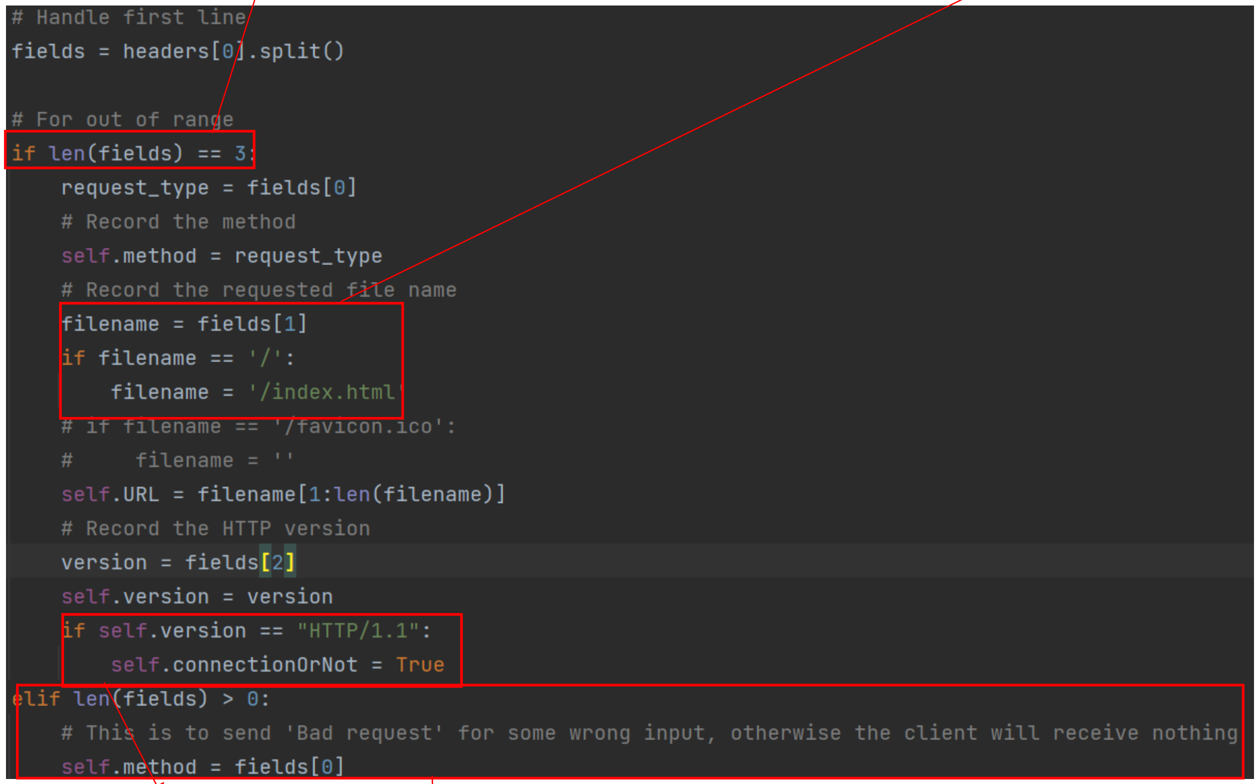
```python
def __eq__(self, other):
    """
    Override __eq__ method to compare if two requests are the same
    :param other:
    :return: bool
    """
    if id(self) == id(other):
        return True
    else:
        return False
```

‘split_request(self, request)’ is an important function. First, it will divide the whole received messages into some small parts by ‘/n’. Second, the system will retrieve some headers and give some attributes value, like ‘If-Modified-Since’ and ‘Connection’.



```python
def split_request(self, request):
    """
    Handle the HTTP request.
    :param request:
    """
    # Parse HTTP headers
    headers = request.split('\n')

    # Handle  'If-Modified-Since' and 'Connection'
    for i in range(1, len(headers)):
        line = headers[i]
        if line.startswith("If-Modified-Since: "):
            conditionDate = MyTime.Time(line[19:len(line)-1])
            self.condition_date = conditionDate
        elif line.startswith("Connection: "):
            if line[12:len(line)-1].lower() == "keep-alive":
                self.connectionOrNot = True
```

Because the first line has 3 key values, the system should deal with it individually. However, the form of wrong input may lead to undesirable errors. In that case, the system just handles the message in which the length is 3. The first value is the method, the second one is the file name the third one is the version. By the way, if the filename is ‘/’, the file name is ‘/index.html’ by default.



```
# Handle first line
fields = headers[0].split()

# For out of range
if len(fields) == 3:
    request_type = fields[0]
    # Record the method
    self.method = request_type
    # Record the requested file name
    filename = fields[1]
    if filename == '/':
        filename = '/index.html'
    # if filename == '/favicon.ico':
    #     filename = ''
    self.URL = filename[1:len(filename)]
    # Record the HTTP version
    version = fields[2]
    self.version = version
    if self.version == "HTTP/1.1":
        self.connectionOrNot = True
elif len(fields) > 0:
    # This is to send 'Bad request' for some wrong input, otherwise the client will receive nothing
    self.method = fields[0]
```

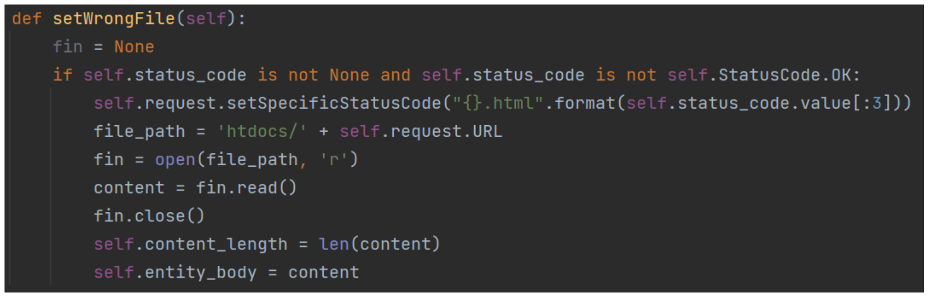
In some specific situations, the request doesn’t have information about whether the connection is kept or not. The system use version to judge.

The last codes are to send ‘Bad request’ for some wrong input. Otherwise, the system will consider the request as a None request for one line request.

d) HttpResponse.py:

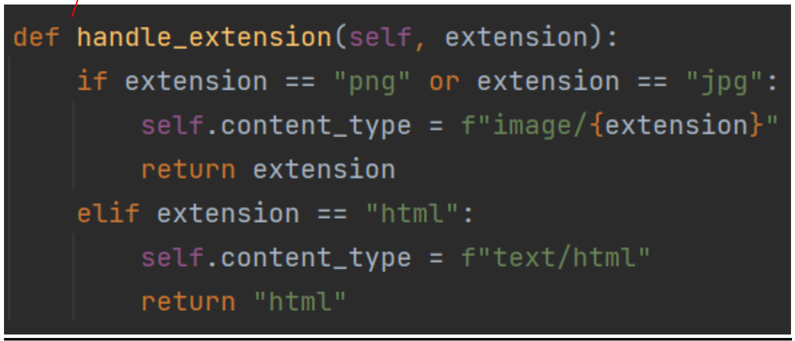
There are still some details should be supplied for ‘HttpResponse.py’.

‘setWrongFile(self)’ is to handle some wrong requests. Although the request is wrong, the system cannot just break down so the system should handle it correctly. If the status code of the response Is not ‘200 OK’, the system still chooses to send an HTML file.



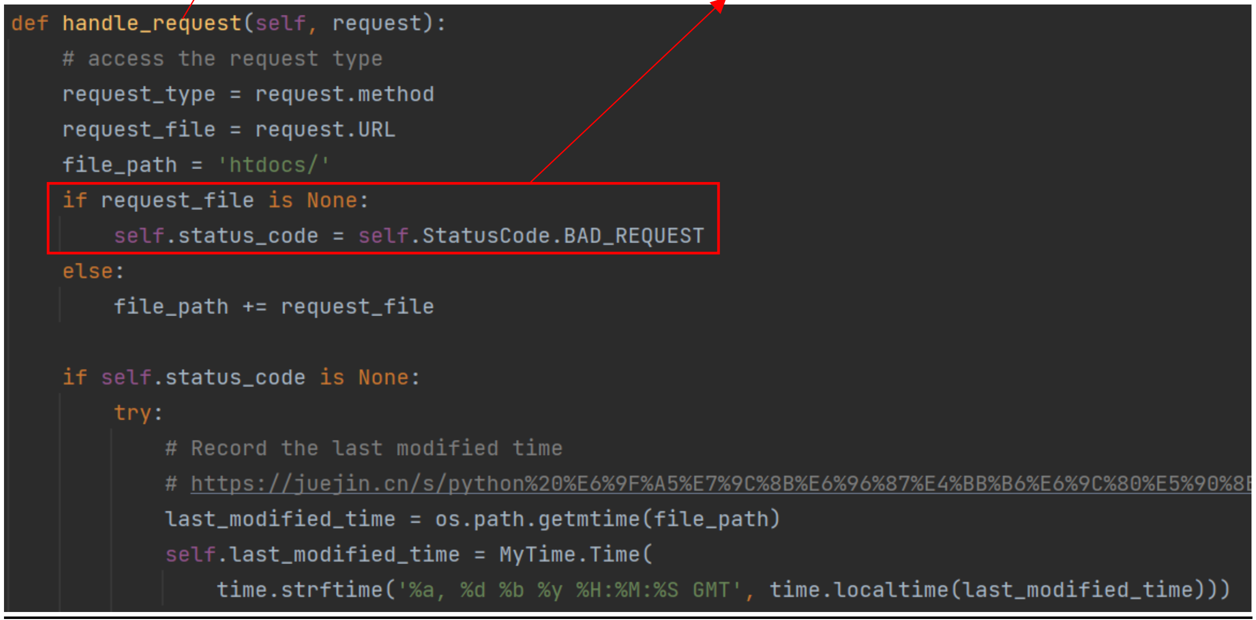
```python
def setWrongFile(self):
    fin = None
    if self.status_code is not None and self.status_code is not self.StatusCode.OK:
        self.request.setSpecificStatusCode("{}.html".format(self.status_code.value[:3]))
        file_path = 'htdocs/' + self.request.URL
        fin = open(file_path, 'r')
        content = fin.read()
        fin.close()
        self.content_length = len(content)
        self.entity_body = content
```

‘handle_extension(self, extension)’ is used to handle the extension and set the content type. I just choose the form of the images are ‘png’ or ‘jpg’, and the text files are ‘html’. If the extension is ‘png’, the content type is ‘image/png’; if the extension is ‘jpg’, the content type is ‘image/jpg’; if the extension is ‘html’, the content type is ‘text/html’.



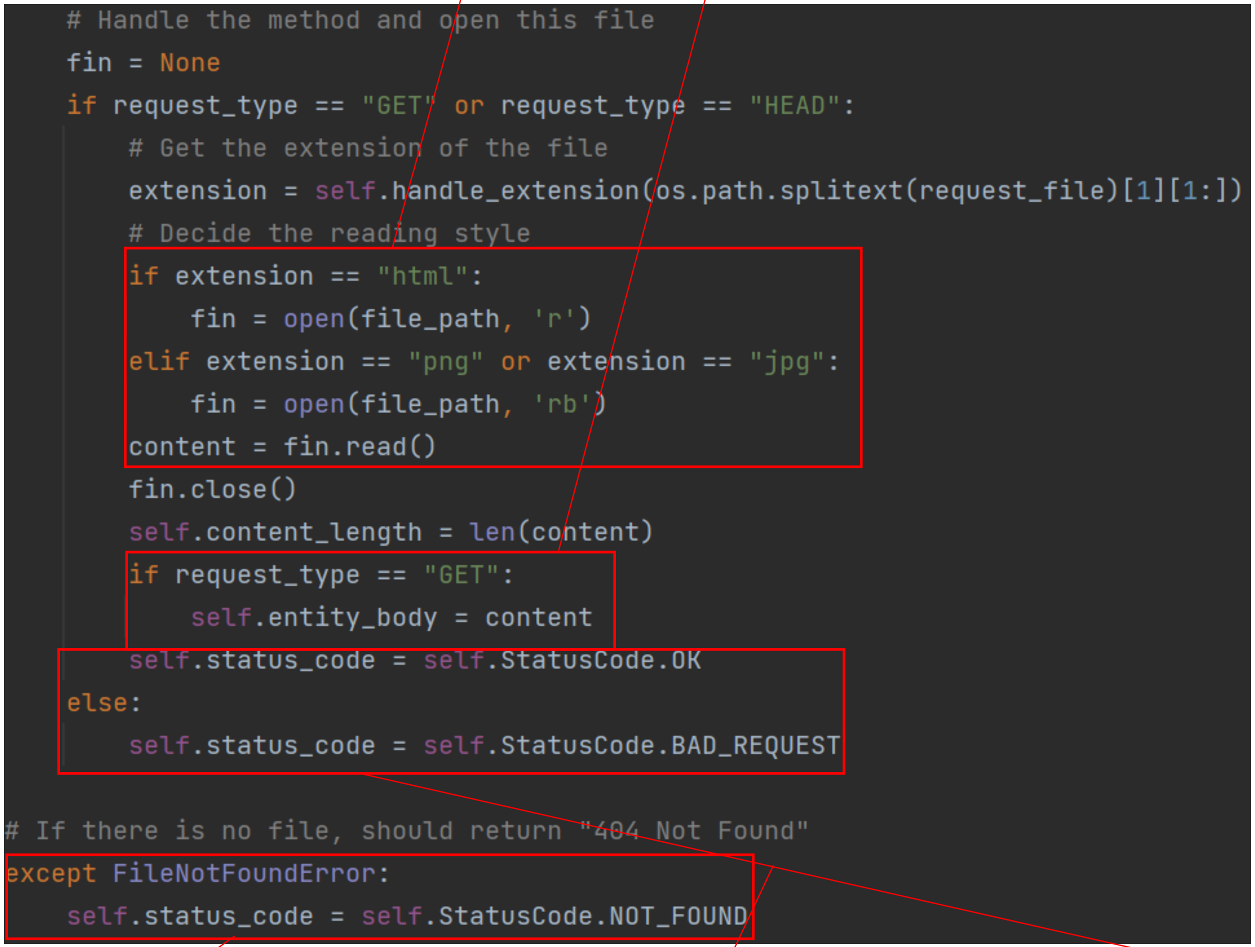
```python
def handle_extension(self, extension):
    if extension == "png" or extension == "jpg":
        self.content_type = f"image/{extension}"
        return extension
    elif extension == "html":
        self.content_type = f"text/html"
        return "html"
```

For ‘handle_request(self,request)’, the key is to set the file name. At first, the method and URL should be set. If the file is None, the status code is ‘Bad request’.



```python
def handle_request(self, request):
    # access the request type
    request_type = request.method
    request_file = request.URL
    file_path = 'htdocs/'
    if request_file is None:
        self.status_code = self.StatusCode.BAD_REQUEST
    else:
        file_path += request_file

    if self.status_code is None:
        try:
            # Record the last modified time
            # https://juejin.cn/s/python%20%E6%9F%A5%E7%9C%8B%E6%96%87%E4%BB%B6%E6%9C%80%E5%90%8E
            last_modified_time = os.path.getmtime(file_path)
            self.last_modified_time = MyTime.Time(
                time.strftime('%a, %d %b %y %H:%M:%S GMT', time.localtime(last_modified_time)))
```

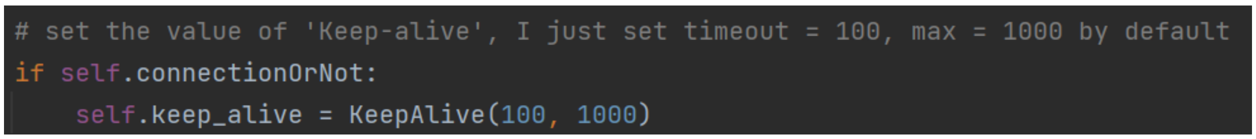
Then we should open the files. Because the open ways to open text files and image files are different, the system should choose ‘r’ or ‘rb’ separately to open the files. If the requested type is ‘GET’, the system should update the content to the ‘entity_body’; if the request type is ‘HEAD’, the system just sent the headers.



```python
    # Handle the method and open this file
    fin = None
    if request_type == "GET" or request_type == "HEAD":
        # Get the extension of the file
        extension = self.handle_extension(os.path.splitext(request_file)[1][1:])
        # Decide the reading style
        if extension == "html":
            fin = open(file_path, 'r')
        elif extension == "png" or extension == "jpg":
            fin = open(file_path, 'rb')
        content = fin.read()
        fin.close()
        self.content_length = len(content)
        if request_type == "GET":
            self.entity_body = content
        self.status_code = self.StatusCode.OK
    else:
        self.status_code = self.StatusCode.BAD_REQUEST

# If there is no file, should return "404 Not Found"
except FileNotFoundError:
    self.status_code = self.StatusCode.NOT_FOUND
```

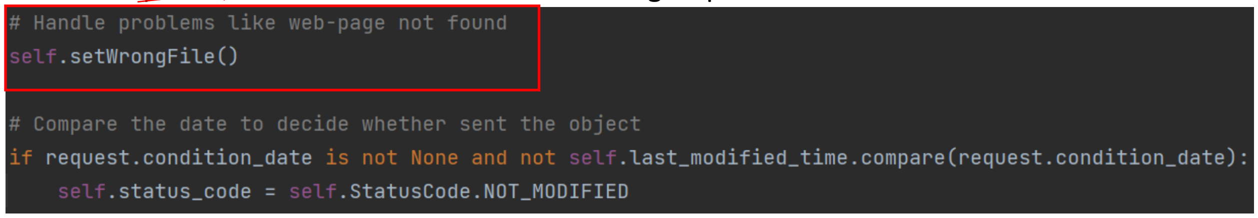
The setting of the status codes is also important in this part. If all is well, the status code is ‘200 OK’; if the request is wrong, the status code is ‘400 Bad Request’; if the file is not found, ‘404 Not Found’ should be set.

Then, the system should set the parameter of ‘keepAlive()’ if the connection is kept. The first one is the number of timeout, the second one is the maximum number. The service providers can set it by themselves.



```python
# set the value of 'Keep-alive', I just set timeout = 100, max = 1000 by default
if self.connectionOrNot:
    self.keep_alive = KeepAlive(100, 1000)
```

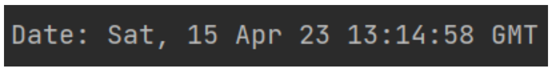
Then the system will set the web file for the wrong requests.



```
# Handle problems like web-page not found
self.setWrongFile()

# Compare the date to decide whether sent the object
if request.condition_date is not None and not self.last_modified_time.compare(request.condition_date):
    self.status_code = self.StatusCode.NOT_MODIFIED
```

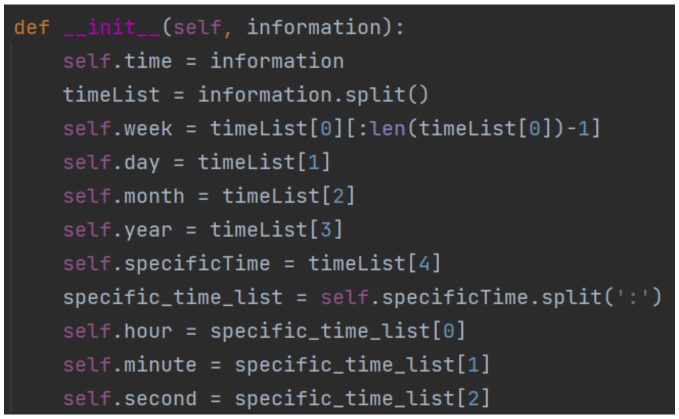
This part is for "If-Modified-Since" and "Last-Modified". If nowadays time is smaller than the last modified time, the status code should be "304 Not Modified" and the file shouldn't be sent.

e) MyTime.py:



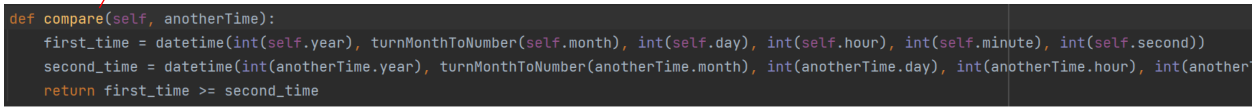
```
Date: Sat, 15 Apr 23 13:14:58 GMT
```

'Time' class can divide the text information into small part.



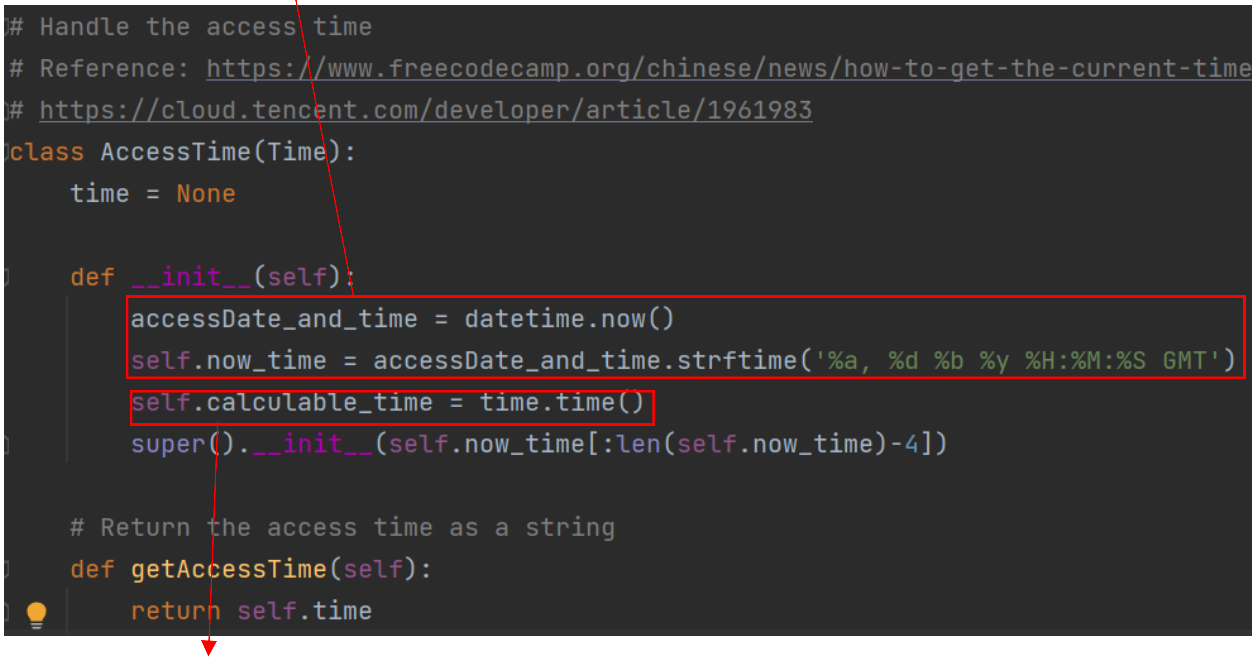
```
def __init__(self, information):
    self.time = information
    timeList = information.split()
    self.week = timeList[0][:len(timeList[0])-1]
    self.day = timeList[1]
    self.month = timeList[2]
    self.year = timeList[3]
    self.specificTime = timeList[4]
    specific_time_list = self.specificTime.split(':')
    self.hour = specific_time_list[0]
    self.minute = specific_time_list[1]
    self.second = specific_time_list[2]
```

'compare(self, another)' compares all the detailed time to judge whether two time is the same or not.



```
def compare(self, anotherTime):
    first_time = datetime(int(self.year), turnMonthToNumber(self.month), int(self.day), int(self.hour), int(self.minute), int(self.second))
    second_time = datetime(int(anotherTime.year), turnMonthToNumber(anotherTime.month), int(anotherTime.day), int(anotherTime.hour), int(another
    return first_time >= second_time
```

'AccessTime(Time)' is dedicated setup to handle access time. The system uses ''datetime.now()' to get the real time, then uses 'strftime()' to get the textual form.



```
# Handle the access time
# Reference: https://www.freecodecamp.org/chinese/news/how-to-get-the-current-time
# https://cloud.tencent.com/developer/article/1961983
class AccessTime(Time):
    time = None

    def __init__(self):
        accessDate_and_time = datetime.now()
        self.now_time = accessDate_and_time.strftime('%a, %d %b %y %H:%M:%S GMT')
        self.calculable_time = time.time()
        super().__init__(self.now_time[:len(self.now_time)-4])

    # Return the access time as a string
    def getAccessTime(self):
        return self.time
```

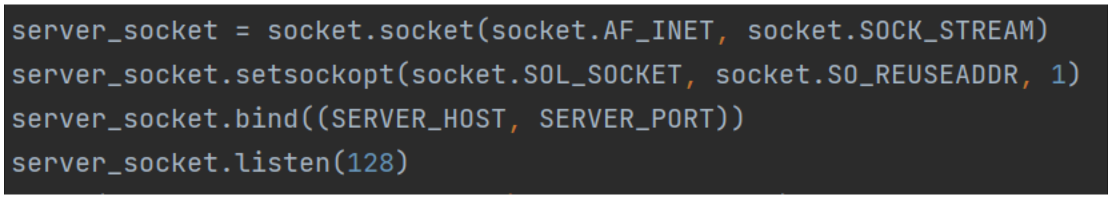
Because ‘time.time()’ is a string and easier to calculate, the system still uses ‘time.time()’ to get a calculable time.

Demonstration:

I set a Python file named ‘TestClient.py’ to help demonstrate the steps. However, I just copy it from the ‘Lab5’ and don’t change it too much. It is just a tool to help me to test and present because it isn’t the part of this project.

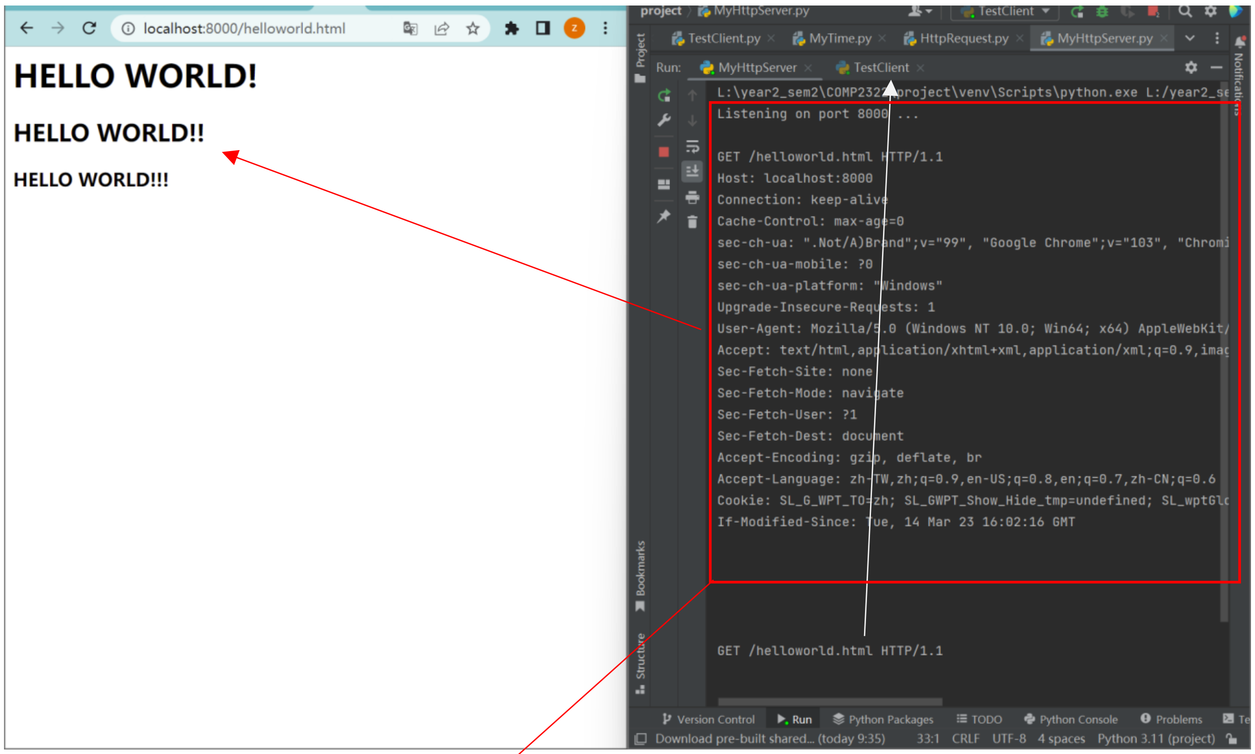
i) *Multi-threaded Web server (create a connection socket when contacted by a client (browser)):*

This part is not easy to show. I create ‘server_socket’ to connect. And 128 specifies the maximum number of connections the operating system can hang before a connection is rejected. Because we should build a multi-threaded Web server so I set this number to 128.



```
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
server_socket.bind((SERVER_HOST, SERVER_PORT))
server_socket.listen(128)
```

The system can receive the request from different clients at the same time.



HELLO WORLD!

HELLO WORLD!!

HELLO WORLD!!!

```
L:\year2_sem2\COMP2322\project\venv\Scripts\python.exe L:/year2_se
Listening on port 8000 ...

GET /helloworld.html HTTP/1.1
Host: localhost:8000
Connection: keep-alive
Cache-Control: max-age=0
sec-ch-ua: ".Not/A)Brand";v="99", "Google Chrome";v="103", "Chromi
sec-ch-ua-mobile: ?0
sec-ch-ua-platform: "Windows"
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,imag
Sec-Fetch-Site: none
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Accept-Encoding: gzip, deflate, br
Accept-Language: zh-TW,zh;q=0.9,en-US;q=0.8,en;q=0.7,zh-CN;q=0.6
Cookie: SL_G_WPT_TO=zh; SL_GWPT_Show_Hide_tmp=undefined; SL_wptGlo
If-Modified-Since: Tue, 14 Mar 23 16:02:16 GMT



GET /helloworld.html HTTP/1.1
```

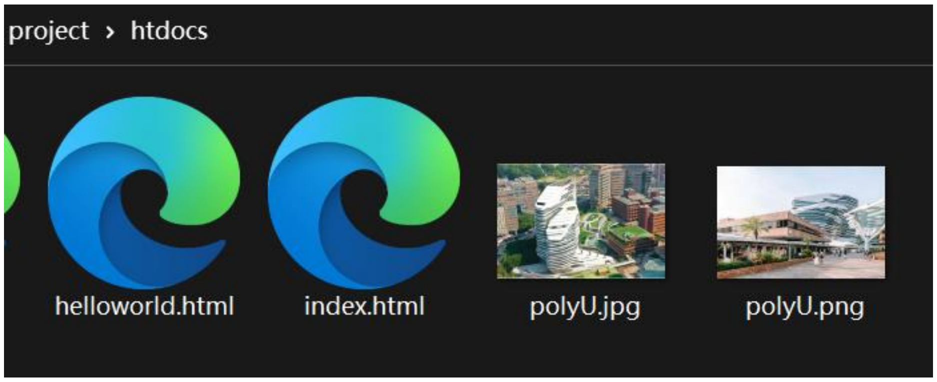
ii) *<u>receive the HTTP request from this connection (Proper request)</u>*

The Web server can receive the HTTP request correctly.

iii) *<u>parse the request to determine the specific file being requested; get the requested file from the server's file system(GET command for both text files and image files)</u>*

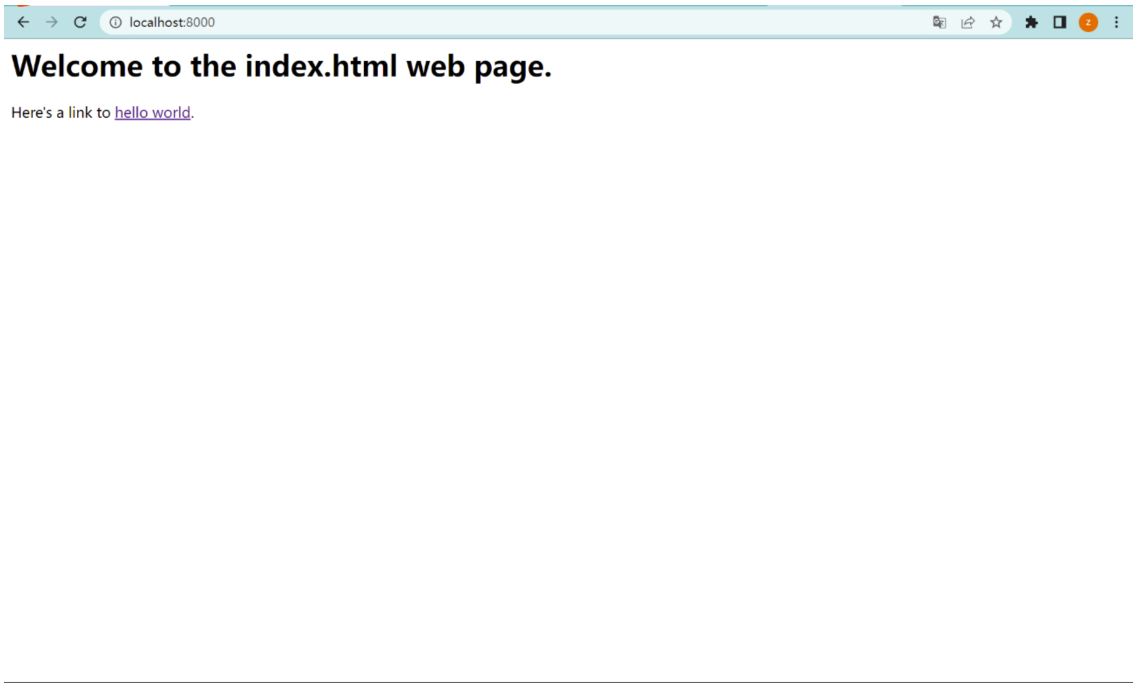
From browsers:

I set 2 text files and 2 image files as examples.



Text files:

Localhost:8000/

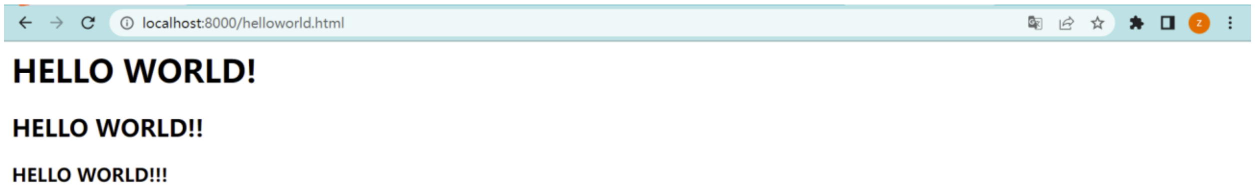


localhost:8000

Welcome to the index.html web page.

Here's a link to hello world.

Localhost:8000/helloworld.html

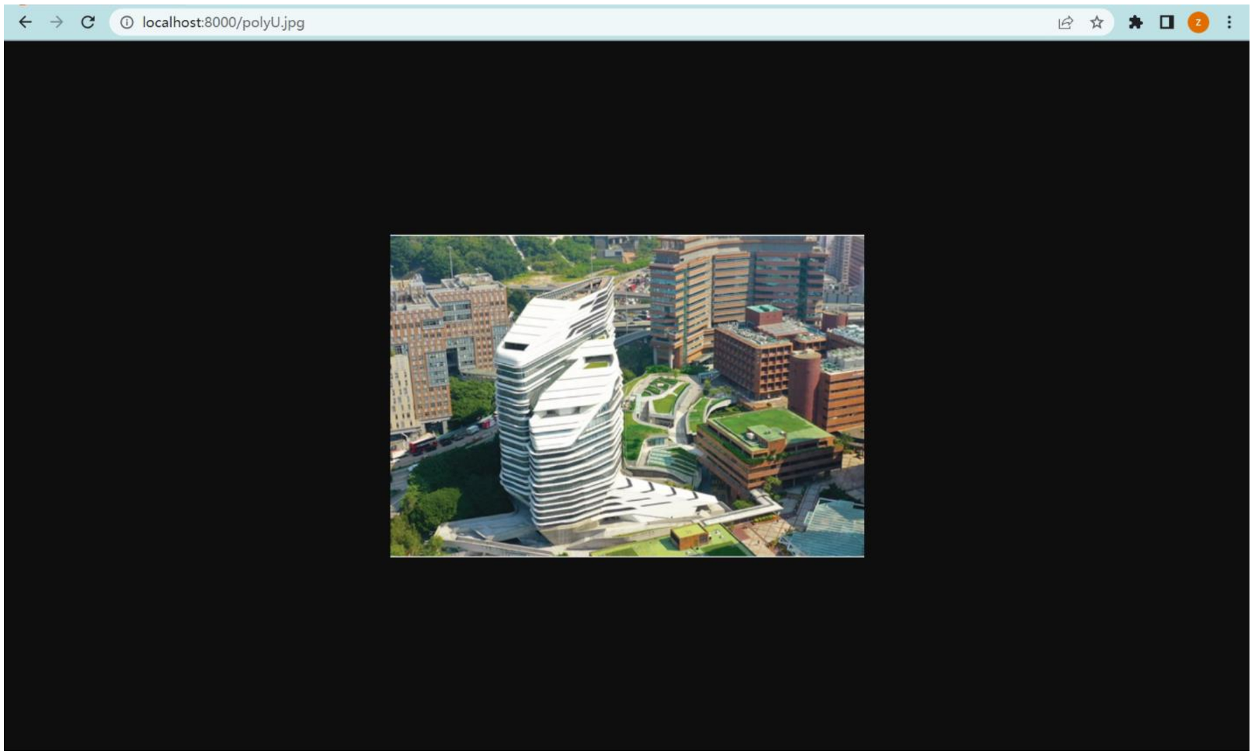


HELLO WORLD!

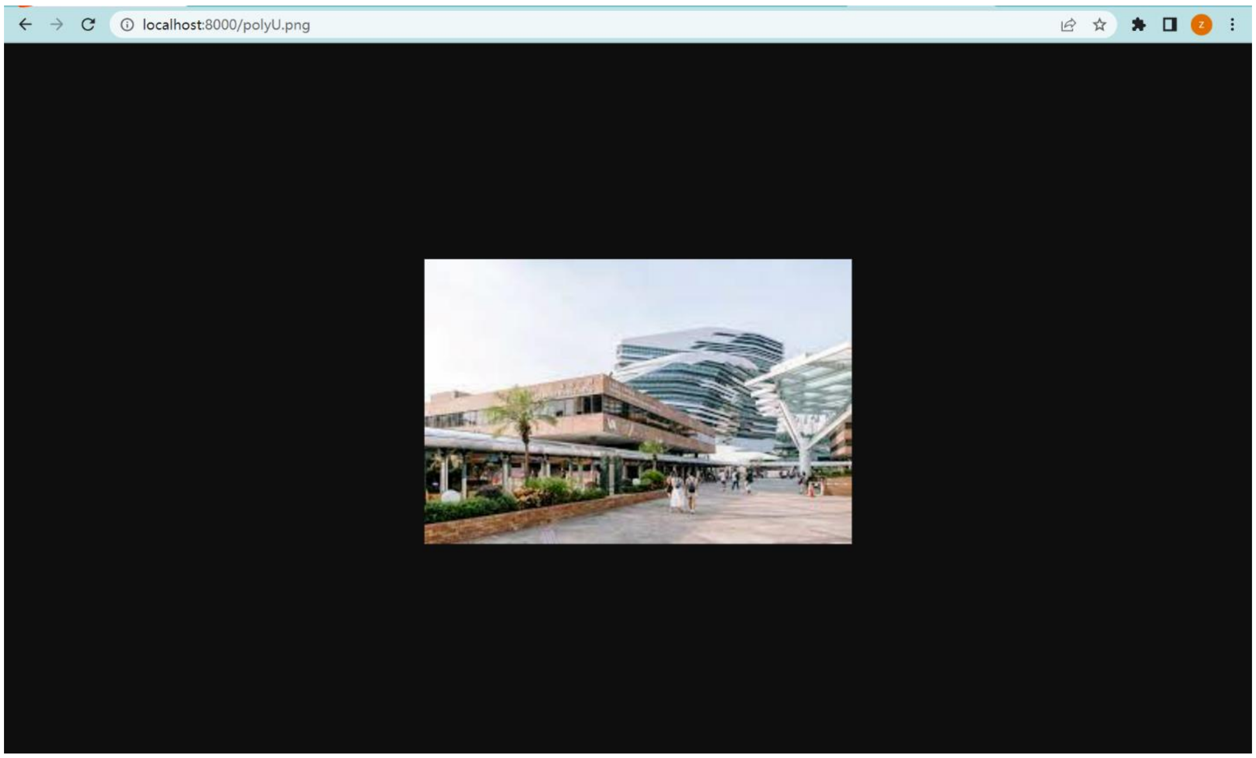
HELLO WORLD!!

HELLO WORLD!!!

Image files:
Jpg: localhost:8000/polyU.jpg

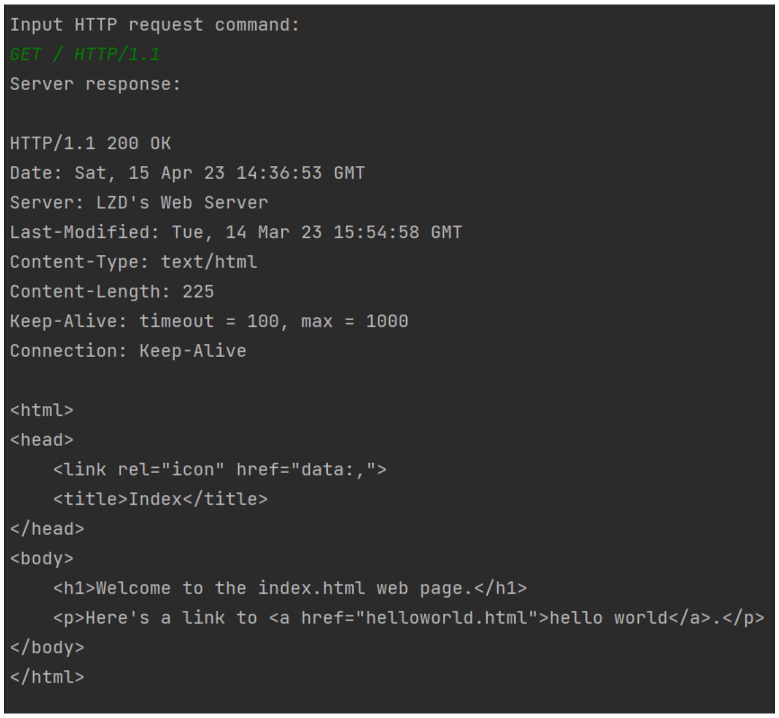


Png: localhost:8000/polyU.png



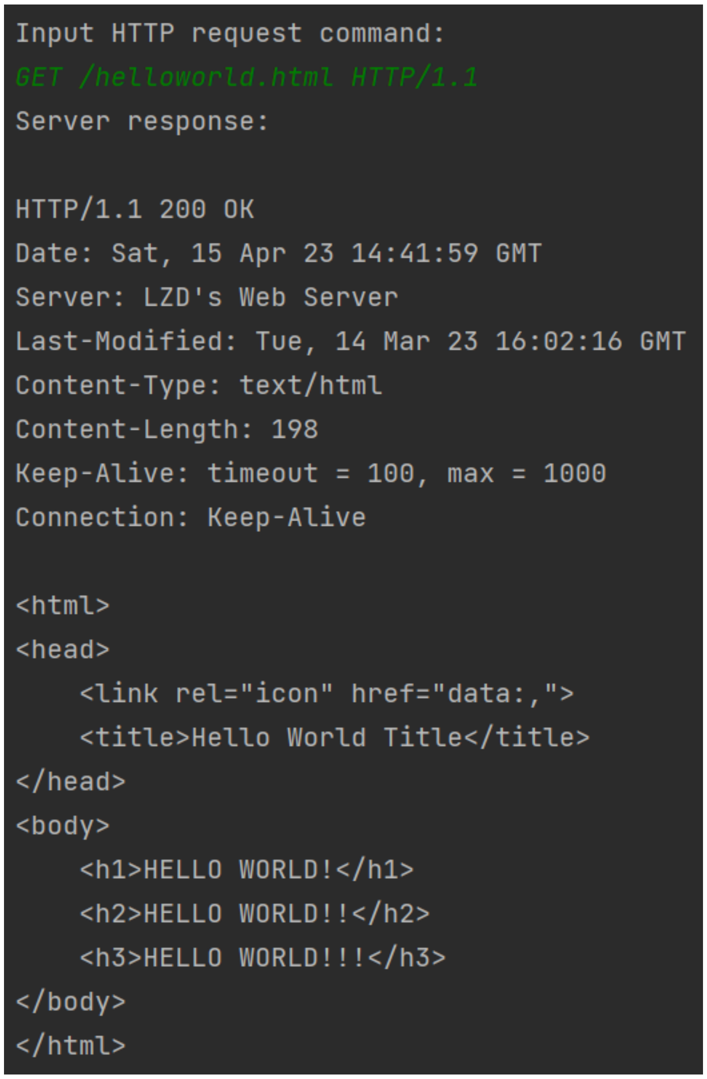
From ‘TestClient.py’:

index.html:



```
Input HTTP request command:
GET / HTTP/1.1
Server response:

HTTP/1.1 200 OK
Date: Sat, 15 Apr 23 14:36:53 GMT
Server: LZD's Web Server
Last-Modified: Tue, 14 Mar 23 15:54:58 GMT
Content-Type: text/html
Content-Length: 225
Keep-Alive: timeout = 100, max = 1000
Connection: Keep-Alive

<html>
<head>
    <link rel="icon" href="data:,">
    <title>Index</title>
</head>
<body>
    <h1>Welcome to the index.html web page.</h1>
    <p>Here's a link to <a href="helloworld.html">hello world</a>.</p>
</body>
</html>
```

helloworld.html:



```
Input HTTP request command:
GET /helloworld.html HTTP/1.1
Server response:

HTTP/1.1 200 OK
Date: Sat, 15 Apr 23 14:41:59 GMT
Server: LZD's Web Server
Last-Modified: Tue, 14 Mar 23 16:02:16 GMT
Content-Type: text/html
Content-Length: 198
Keep-Alive: timeout = 100, max = 1000
Connection: Keep-Alive

<html>
<head>
    <link rel="icon" href="data:,">
    <title>Hello World Title</title>
</head>
<body>
    <h1>HELLO WORLD!</h1>
    <h2>HELLO WORLD!!</h2>
    <h3>HELLO WORLD!!!</h3>
</body>
</html>
```

Because the images are binary files, and 'sendall()' will buffer the too large file, I don't use images to show examples.

iv) *create an HTTP response message consisting of the requested file preceded by header lines(Proper response)*

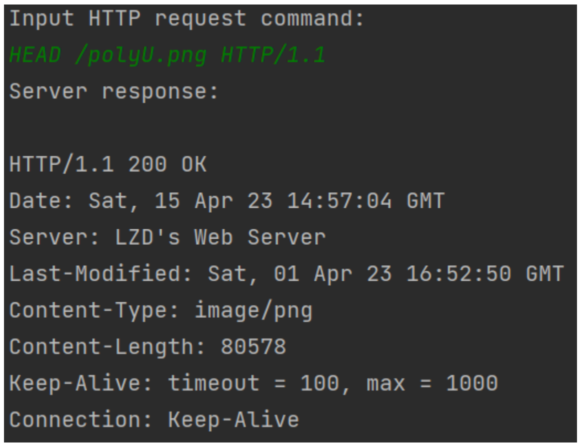
This also demonstrate,

1) HEAD command

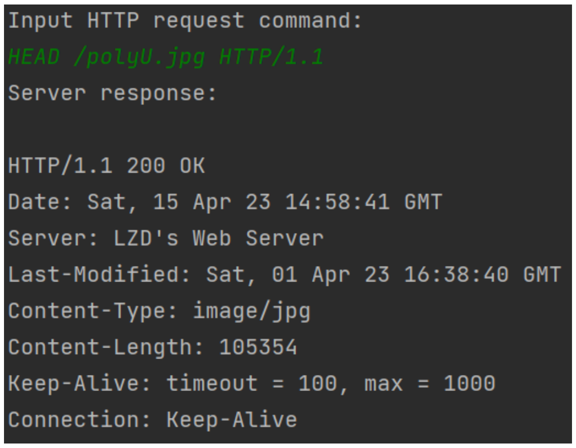
2) Four types of response statuses ONLY, including 200 OK, 400 Bad Request, 404 File Not Found, 304 Not Modified

I use 'TestCLient.py' to show mainly.

Head:

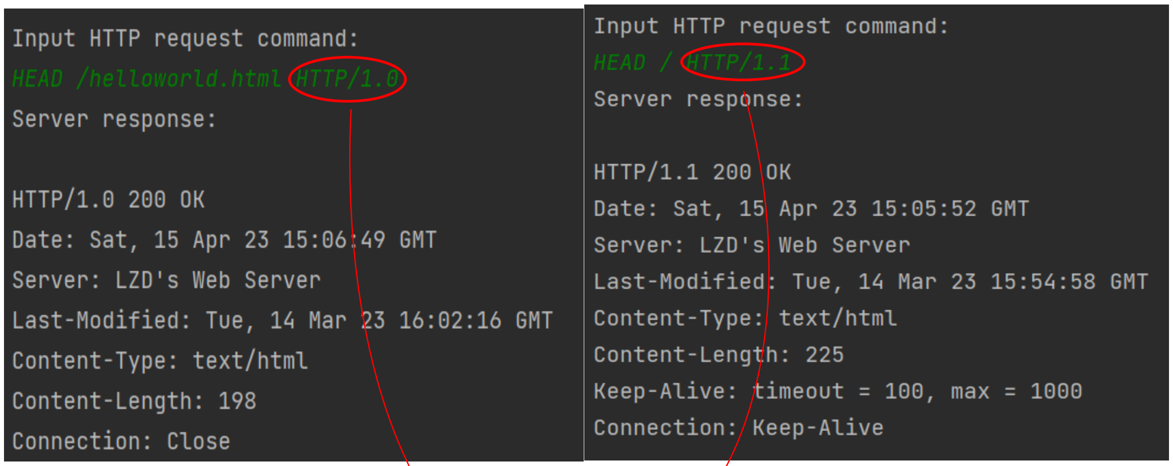


```
Input HTTP request command:
HEAD /polyU.png HTTP/1.1
Server response:

HTTP/1.1 200 OK
Date: Sat, 15 Apr 23 14:57:04 GMT
Server: LZD's Web Server
Last-Modified: Sat, 01 Apr 23 16:52:50 GMT
Content-Type: image/png
Content-Length: 80578
Keep-Alive: timeout = 100, max = 1000
Connection: Keep-Alive
```



```
Input HTTP request command:
HEAD /polyU.jpg HTTP/1.1
Server response:

HTTP/1.1 200 OK
Date: Sat, 15 Apr 23 14:58:41 GMT
Server: LZD's Web Server
Last-Modified: Sat, 01 Apr 23 16:38:40 GMT
Content-Type: image/jpg
Content-Length: 105354
Keep-Alive: timeout = 100, max = 1000
Connection: Keep-Alive
```

By the way, we cannot use 'textClient.py' continuously to handle the images. Because the project didn't require us to design a nice client program, I haven't tried to solve these problems.



```
Input HTTP request command:
HEAD /helloworld.html HTTP/1.0
Server response:

HTTP/1.0 200 OK
Date: Sat, 15 Apr 23 15:06:49 GMT
Server: LZD's Web Server
Last-Modified: Tue, 14 Mar 23 16:02:16 GMT
Content-Type: text/html
Content-Length: 198
Connection: Close
```

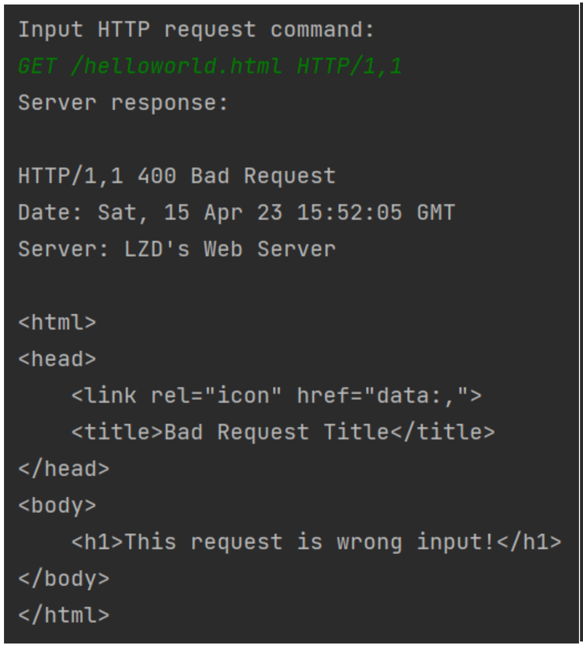
```
Input HTTP request command:
HEAD / HTTP/1.1
Server response:

HTTP/1.1 200 OK
Date: Sat, 15 Apr 23 15:05:52 GMT
Server: LZD's Web Server
Last-Modified: Tue, 14 Mar 23 15:54:58 GMT
Content-Type: text/html
Content-Length: 225
Keep-Alive: timeout = 100, max = 1000
Connection: Keep-Alive
```

I have already shown '200 OK', so I don't show it here.

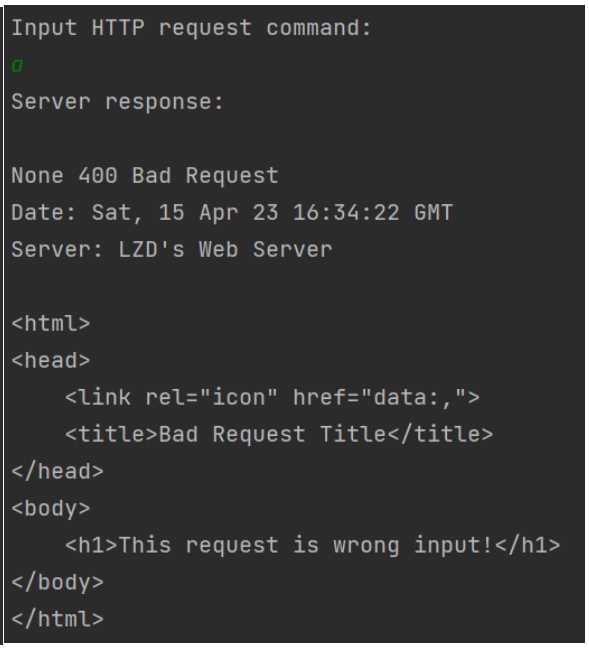
I also want to show 'HTTP/1.0' means 'Connection: Close' by default; 'HTTP/1.1' means 'Connection: Keep-Alive' by default.

400 Bad Request:

I put the web responses in the entity body to handle some simple errors.



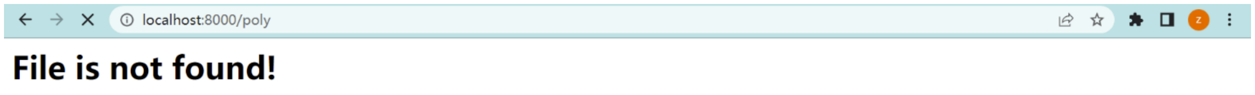
```
Input HTTP request command:
GET /helloworld.html HTTP/1,1
Server response:

HTTP/1,1 400 Bad Request
Date: Sat, 15 Apr 23 15:52:05 GMT
Server: LZD's Web Server

<html>
<head>
    <link rel="icon" href="data:,">
    <title>Bad Request Title</title>
</head>
<body>
    <h1>This request is wrong input!</h1>
</body>
</html>
```



```
Input HTTP request command:
a
Server response:

None 400 Bad Request
Date: Sat, 15 Apr 23 16:34:22 GMT
Server: LZD's Web Server

<html>
<head>
    <link rel="icon" href="data:,">
    <title>Bad Request Title</title>
</head>
<body>
    <h1>This request is wrong input!</h1>
</body>
</html>
```

404 File Not Found:

From browser:

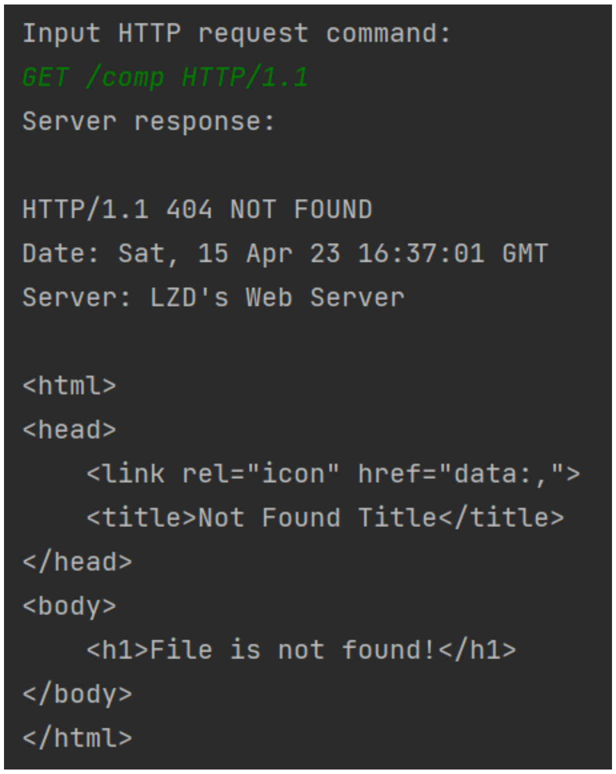


localhost:8000/poly

File is not found!

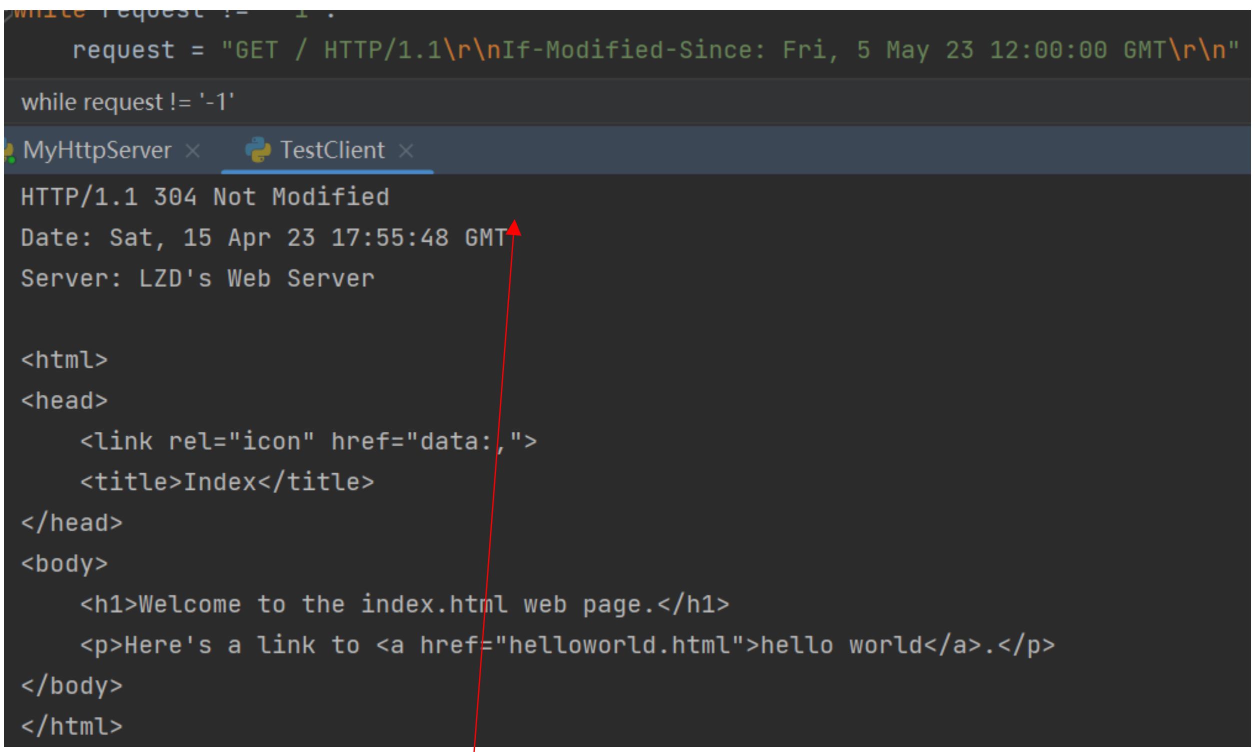
By the way, I also set the corresponding page for '304 Not Modified' and '400 Bad Request' but don't show them here.

From 'testClient.py':



```
Input HTTP request command:
GET /comp HTTP/1.1
Server response:

HTTP/1.1 404 NOT FOUND
Date: Sat, 15 Apr 23 16:37:01 GMT
Server: LZD's Web Server

<html>
<head>
    <link rel="icon" href="data:,">
    <title>Not Found Title</title>
</head>
<body>
    <h1>File is not found!</h1>
</body>
</html>
```

304 Not Modified:
‘2023 May 5 12:00:00’ is older than ‘2023 Apr 15 17:55:48’, so the file isn’t sent. A web page about 304 is displayed.

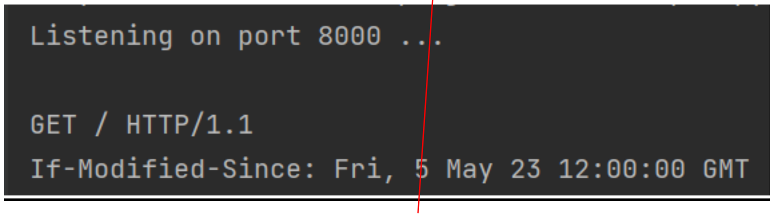


```
    request = "GET / HTTP/1.1\r\nIf-Modified-Since: Fri, 5 May 23 12:00:00 GMT\r\n"
```

```
HTTP/1.1 304 Not Modified
Date: Sat, 15 Apr 23 17:55:48 GMT
Server: LZD's Web Server

<html>
<head>
    <link rel="icon" href="data:,">
    <title>Index</title>
</head>
<body>
    <h1>Welcome to the index.html web page.</h1>
    <p>Here's a link to <a href="helloworld.html">hello world</a>.</p>
</body>
</html>
```

v) Handle Last-Modified and If-Modified-Since header fields

The request is as follows,
“GET / HTTP/1.1
If-Modified-Since: Fri, 5 May 23 12:00:00 GMT”

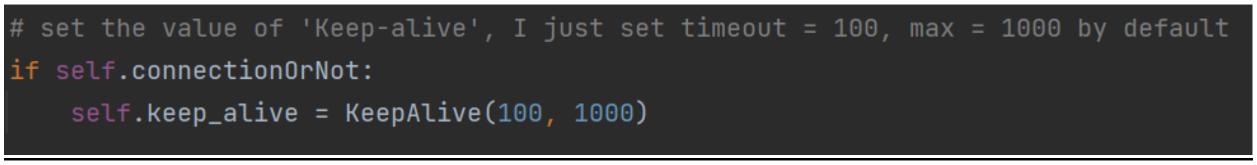


```
Listening on port 8000 ...

GET / HTTP/1.1
If-Modified-Since: Fri, 5 May 23 12:00:00 GMT
```

The response is shown in “304 Not Modified”.

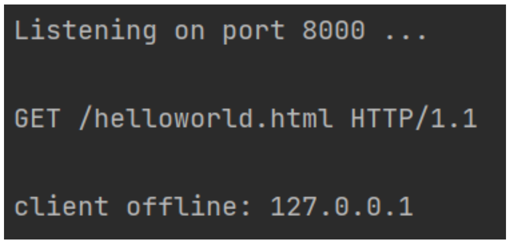
Vi) Handle Connection: Keep-Alive header field

Timeout is 100, the maximum number to connect is 1000.



```
# set the value of 'Keep-alive', I just set timeout = 100, max = 1000 by default
if self.connectionOrNot:
    self.keep_alive = KeepAlive(100, 1000)
```

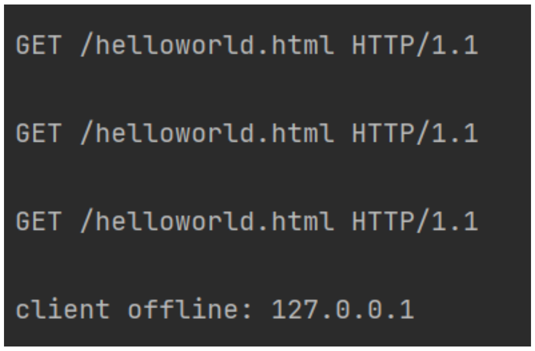
If “Connection: keep-alive” is found in the request. The ‘keep-alive’ is kept. The server will allow an idle connection of 100 seconds to remain open before it is closed. If time is up, the system will show "client offline:"+IP address. By the way, one client may build many connections at the same time. They have the same IP address.



```
Listening on port 8000 ...

GET /helloworld.html HTTP/1.1

client offline: 127.0.0.1
```

For a more convenient presentation, I set ‘max’ to 3.
If the number of connections is larger the maximum number, 3. The client will be offline.

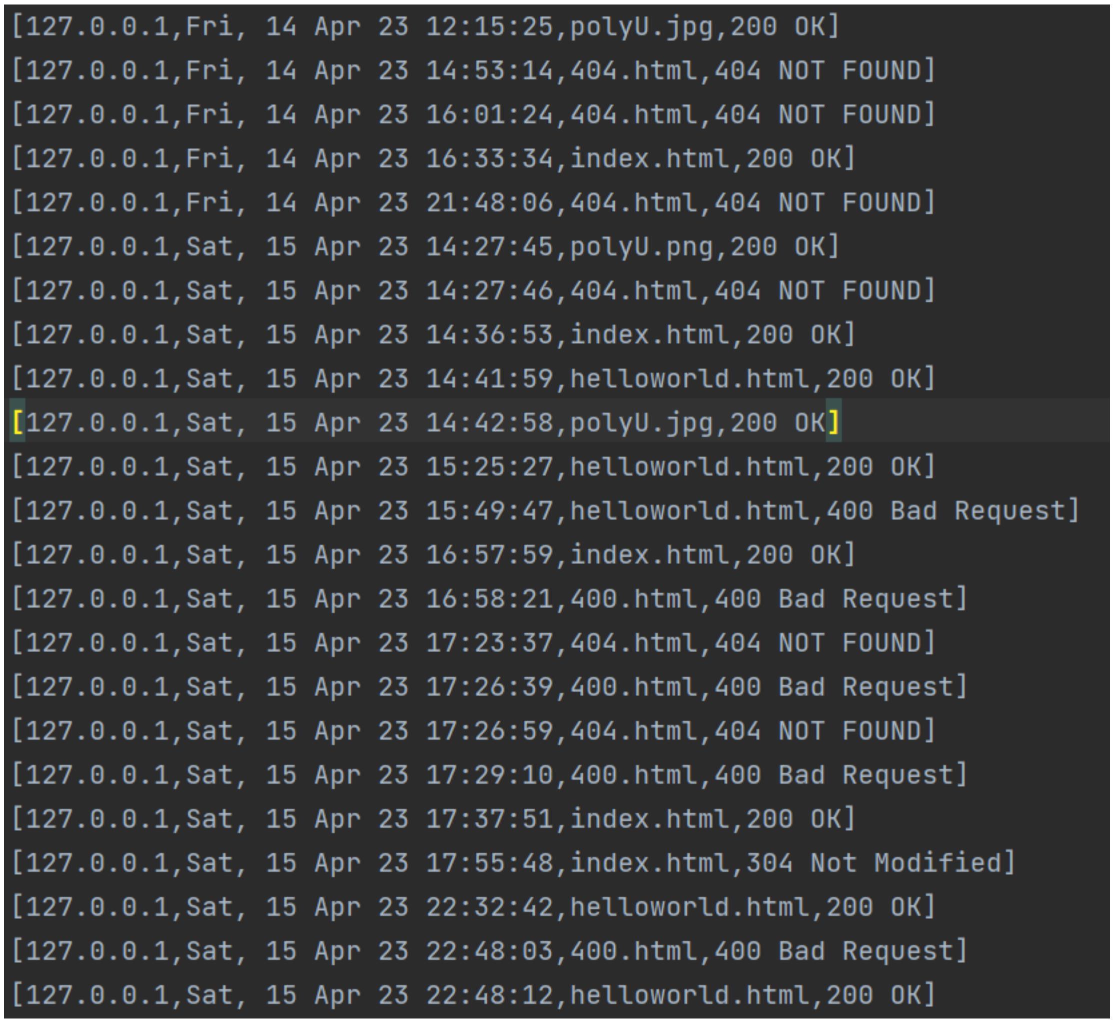
One more emphasis, ‘TestClient.py’ just help to do the test. Every problem of it is not because the Web server. Like we cannot use ‘GET’ to retrieve 3 images at once time when using ‘TestClient.py’



```
GET /helloworld.html HTTP/1.1

GET /helloworld.html HTTP/1.1

GET /helloworld.html HTTP/1.1

client offline: 127.0.0.1
```

Vii) log file

There is a log file that records the historical information about the client requests and server responses.



```
[127.0.0.1,Fri, 14 Apr 23 12:15:25,polyU.jpg,200 OK]
[127.0.0.1,Fri, 14 Apr 23 14:53:14,404.html,404 NOT FOUND]
[127.0.0.1,Fri, 14 Apr 23 16:01:24,404.html,404 NOT FOUND]
[127.0.0.1,Fri, 14 Apr 23 16:33:34,index.html,200 OK]
[127.0.0.1,Fri, 14 Apr 23 21:48:06,404.html,404 NOT FOUND]
[127.0.0.1,Sat, 15 Apr 23 14:27:45,polyU.png,200 OK]
[127.0.0.1,Sat, 15 Apr 23 14:27:46,404.html,404 NOT FOUND]
[127.0.0.1,Sat, 15 Apr 23 14:36:53,index.html,200 OK]
[127.0.0.1,Sat, 15 Apr 23 14:41:59,helloworld.html,200 OK]
[127.0.0.1,Sat, 15 Apr 23 14:42:58,polyU.jpg,200 OK]
[127.0.0.1,Sat, 15 Apr 23 15:25:27,helloworld.html,200 OK]
[127.0.0.1,Sat, 15 Apr 23 15:49:47,helloworld.html,400 Bad Request]
[127.0.0.1,Sat, 15 Apr 23 16:57:59,index.html,200 OK]
[127.0.0.1,Sat, 15 Apr 23 16:58:21,400.html,400 Bad Request]
[127.0.0.1,Sat, 15 Apr 23 17:23:37,404.html,404 NOT FOUND]
[127.0.0.1,Sat, 15 Apr 23 17:26:39,400.html,400 Bad Request]
[127.0.0.1,Sat, 15 Apr 23 17:26:59,404.html,404 NOT FOUND]
[127.0.0.1,Sat, 15 Apr 23 17:29:10,400.html,400 Bad Request]
[127.0.0.1,Sat, 15 Apr 23 17:37:51,index.html,200 OK]
[127.0.0.1,Sat, 15 Apr 23 17:55:48,index.html,304 Not Modified]
[127.0.0.1,Sat, 15 Apr 23 22:32:42,helloworld.html,200 OK]
[127.0.0.1,Sat, 15 Apr 23 22:48:03,400.html,400 Bad Request]
[127.0.0.1,Sat, 15 Apr 23 22:48:12,helloworld.html,200 OK]
```