Instructions

Using Python, implement a web server that uses a simplified version of the HTTP protocol. Base your solution on the provided code templates. The use of external libraries and other built-in modules, packages or functions that are not already imported into `server.py` is not allowed. You should code your solution entirely in the `server.py` and submit it to automatic grading system.

Additionally, you are highly encouraged to write as many unit tests as you wish: the more the better. A starting point for unit tests has been provided in file `tests.py`. Although the unit tests that you write will not be graded, writing them will ease the development and help you in finding bugs. Note: Unit tests delete the contents of the database. I highly recommend using TDD approach where your first write the test (which fails), and then the functionality of your web server that makes the test pass.

Your solution will be checked with an anti-plagiarism software. All offenders (those copying and those that provide material for copying) will be awarded with 0 points which means that they instantly fail this course.

Assignment specification

Your web server should support:

- Only HTTP requests following HTTP/1.1. This means that a valid request must contain a header with `host` field;
- Only two HTTP methods: `GET` and `POST`. When parsing POST requests, make sure the client is sending the `content-length` header field. It will tell you the number of bytes you have to read to parse the parameters in the body of the request. You can find a couple of request message examples on this page.

The server should be able to process both **static** and **dynamic** content. Static content denotes serving files from the file system (data inside folder `www-data`), while dynamic content denotes a simple web application that stores student data.

HTTP responses

While the HTTP protocol supports multiple status response codes, your server should support only the ones mentioned below. To get a better idea of how HTTP response messages look like, visit this page.

200 OK

When the server receives a valid request, it should respond with a response code `200 OK`. To prepare the correct response header, use the provided `HEADER_RESPONSE_200` variable.

301 Moved Permanently

If you want to redirect the client to another page, use the `301 Moved Permanently` response message. An example of a `301` message can be found on Wikipedia. The important part is setting the response line and the appropriate headers, while the response body may be arbitrary (but it should still be meaningful to a human reader).

400 Bad request

If you receive a request which does not conform to the specifications, your should return a `400 Bad request` message. As with `301`, the important part is setting the response line and the appropriate headers, while the response body may be arbitrary (but it should still be meaningful to a human reader).

404 Not found

If you receive a request to a non-existent resource, the server should response it a `404 Not found` message. Your response should be prepared with the help of the `RESPONSE_404` variable.

405 Method not allowed

If you receive a request using an invalid HTTP method, the server should respond with a `405 Method not allowed` message. As with `301`, the important part is setting the response line and the appropriate headers, while the response body may be arbitrary (but it should still be meaningful to a human reader).

Serving static pages

Serving static pages denotes serving a resource as-is from the `www-data` directory.

Methods

All static content should be served over requests using either `GET` or `POST` method. If another method is used, the server should response with a `405` message.

Response headers `content-length` and `content-type`

When responding, the server should appropriately set both the `content-length` and the `content-type` response headers. To find out the latter, you can use the `guess_type` function from the `mimetypes` package to. If the function `guess_type` returns `None` (i. e. cannot find out the file's mimetype), set it manually to `application/octet-stream`.

Parsing parameters

Always assume that the request parameters (be it either in the URL of a GET request or in the body of a POST request) are URL encoded. With POST requests, you can assume that the `content-type` is always set to `application/x-www-form-urlencoded`.

The string containing url-encoded parameters can be decoded using the `unquote_plus` function from the `urllib.parse` package.

Serving files and directories

When the client requests a URI that points to an actual file or directory, serve the client the requested resource and set the response code to `200`. If the requested resource (a file or a folder) does not exist, return error `404`.

When serving static files, you need to pay special attention to the trailing slash `/` in the URI as follows.

Examples without the trailing slash

When requesting a resource without a trailing slash in the URI, the server should do as follows; as an example, imagine requesting URI `http://localhost[:port]/dir/test`.

1. If there is a file `test` (within directory `dir`), send the file to the client. Use the `isfile` function to help determine if a file exists.
2. If there is a directory `test` (within directory `dir`), redirect the client to the address `http://localhost[:port]/dir/test/`, that is, add a trailing slash to the URI. To redirect the client, return response code `301` and add the `Location` field response header. When determining if a directory exists, use the `isdir` function.

Examples with the trailing slash

When the client requests a resource that contains the trailing slash, the server should do as follows; as an example, take again the request to URI `http://localhost[:port]/dir/test/`.

1. If there exists a file `index.html` (within the directory `dir/test/`), serve the contents of the aforementioned file, that is file `/dir/test/index.html` and set the response code to `200`.
2. If there is no `index.html` inside the `dir/test/` directory, serve the code `200` and list the files inside the directory in the response body. Always add `..` to the file list, which points to a parent the directory.

Use variables `DIRECTORY_LISTING` and`FILE_TEMPLATE` to create the file list. The first variable specifies a general HTML template, and the second outlines the display of a particular file or a directory. Use the function `listdir` to retrieve the list of files and directories for each location. Also arrange the files and directory in the ascending alphabetical order (from A to Z). The following is an example response to the `GET` request to`http://localhost/listing/`.

```
HTTP/1.1 200 OK
content-type: text/html
content-length: 669
connection: Close

<!DOCTYPE html>
<html lang="en">
<meta charset="UTF-8">
<title>Directory listing: /listing/</title>

<h1>Contents of /listing/:</h1>

<ul>
  <li><a href='..'>..</li>
  <li><a href='a.html'>a.html</li>
  <li><a href='dir'>dir</li>
  <li><a href='file_1.txt'>file_1.txt</li>
  <li><a href='file_10.txt'>file_10.txt</li>
  <li><a href='file_2.txt'>file_2.txt</li>
  <li><a href='file_3.txt'>file_3.txt</li>
  <li><a href='file_4.txt'>file_4.txt</li>
  <li><a href='file_5.txt'>file_5.txt</li>
  <li><a href='file_6.txt'>file_6.txt</li>
  <li><a href='file_7.txt'>file_7.txt</li>
  <li><a href='file_8.txt'>file_8.txt</li>
  <li><a href='file_9.txt'>file_9.txt</li>
</ul>
```

Serving dynamic content

Implement a dynamic web application that allows users to store and retrieve student data.

Database (already implemented)

The web application requires a database. The database has already been implemented in a form of a file and two functions that read and write to it.

Use function `save_to_db(first(str), last(str))` to add a user to the database, and use `read_from_db(criteria(dict))` to retrieve them from it. (Again, do not forget: running unit tests will delete the contents of the database.)

For instance, calling `save_to_db("Janez", "Novak")` will create a new DB entry with the first name `Janez` and the last name `Novak`. Additionally, the database will generate a unique number for this entry.

Each entry is internally represented as a Python dictionary with three string keys: `number`, `first` and `last`. Key `number` represents a unique identification number, while keys `first` and `last` represent the first and last name. Here's an example.

```
student = {"number": 1, "first": "Janez", "last": "Novak"}
```

Function `read_from_db(criteria(dict))` will return a list of such dictionaries. You can call the function with an optional argument of type dict to further limit the results. For instance, the following invocation will return all entries where the first name is set to `Janez` and last name is set to `Novak`.

```
students = read_from_db({"first": "Janez", "last": "Novak"})
# students is a list of dicts
```

As criteria, you may provide an arbitrary combination of keys using fields `number`, `first` and `last`.

Design (already implemented)

The design (HTML and CSS) is implemented within files `app_list.html`, `app_add.html` and `user_style.css`. Do not modify these files.

Dynamic application

The application should respond to the following two (virtual) URLs. (We call these URLs virtual, because the resources (`app-add`, `app-index`, and `app-json`) do not exist in the file system, although to the client it might seem as they do.)

Adding entries to the database

URL: **`http://localhost[:port]/app-add`**

On this URL, the web server should only accept POST requests. Each request has to contain two parameters: `first`, containing the student's first name, and `last`, containing the student's last name.

If the request contains both parameters, add an entry to the DB, return code `200`, in the body of the HTTP response, return the contents of the `app_add.html`.

If any of the parameters is missing, the server should respond with a `400` error message. If the request method is not POST, the server should respond with `405`.

Reading and filtering data in HTML

URL **`http://localhost[:port]/app-index`**

On this URL, the server should only accept GET requests; all other request types should be responded with an appropriate error code. If a request contains no parameters, simply read all entries from the database, and prepare your response with code `200`.

Prepare the body of the response using the contents of `user_list.html`. Before sending the response body to the user, make sure you replace the `{{STUDENTS}}` placeholder with the actual information about students. Use the `ROW_TEMPLATE` variable to display data for every student -- every student should be represented with a row in the table.

```
TABLE_ROW = """
<tr>
    <td>%d</td>
    <td>%s</td>
    <td>%s</td>
</tr>
"""
```

The `ROW_TEMPLATE` variable contains three placeholders for student data: replace the `%d` with student's `number`, the first `%s` with student's `first` name, and second `%s` with students `last` name.

Additionally, the GET request might contain criteria parameters for limiting the amount of students that will be displayed. The criteria could contain three parameters: `number`, `first` and `last`. If any of these request parameters is set, use it to filter the list of students. For instance, a GET request to `http://localhost[:port]/app-index?first=Janez` should return a list of students whose first name is `Janez`.

Reading and filtering data in JSON

URL **`http://localhost[:port]/app-json`**

On this URL, the server should only accept GET requests; all other request types should be responded with an appropriate error code. If a request contains no parameters, simply read all entries from the database, and prepare your response with code `200`.

The response should be rendered as a message in JSON. You may use the Python's built-it module `json` and its function `json.dumps(object)`; the data from the

database can be passed into the `dumps` function directly: `json.dumps(read_from_db())`.

The result must be returned in the response body. When returning JSON messages, one must set the `content-type` to `application/json`. An example response is given below.

```
HTTP/1.1 200 OK
content-type: application/json
content-length: 256
connection: Close

[
    {
        "number": 1,
        "first": "Janez",
        "last": "Novak"
    },
    {
        "number": 2,
        "first": "Marija",
        "last": "Novak"
    },
    {
        "number": 3,
        "first": "Cirila",
        "last": "Novak"
    }
]
```

Additional requirements

Remember, you are not allowed to use any functions, modules or packages aside from those the `server.py` has already imported (`mimetypes`, `pickle`, `socket`, `os.path.isdir`, `urllib.parse.unquote_plus`, and `json`) or those that you write yourself.

When writing unit and integration tests, there is no such constraint: some tests already a third-party library called Requests for sending HTTP requests and parsing HTTP responses.

The grading will be done automatically using integration tests, similar to those in file `tests.py`: every required functionality will be accompanied with an integration test that will start your server, send an HTTP request to it, and then validate server's HTTP response.

During grading, each test will start your server by invoking function `main(port(int))`. So make sure that your server is listening on the appropriate port number and not on some hard-coded value (like `8080`).

Feel free to either pose your questions in the forms or ask in person during office hours or lab sessions.