Scientific Programming – Network programming in Python

Exercises:

1) The first Python server presented in the first lecture on network programming does actually not do anything else but to thank the client for the connection. Implement this simple server, plus the simple client from the lecture, and test that they are working. For testing purpose, you can do this all on your local host (no need for an internet connection).

2) Let's make this a little bit more complicated: Write a server whose response depends on a specific request of the client:

- If the client sends the request string "time", the server sends as a response its own local time.
- If the client sends the string "nuqneH?", the server responds with "Qapla'!"
- If the client sends anything else, the server works like an echo server, it simply sends the received data back to the client.

The client itself should print out what response it received from the server. Test the different answers from the server.
[Note: "nuqneH?" ... "Qapla'!" is actually a complete conversation in Klingon :-)]

3) Implement both the servers and the clients from exercise 1 and 2 in such a way, that they know when they have received the full message from the remote connection.

4) Implement the server from exercise 1 in such a way that it can handle multiple requests at the same time! Test it by contacting it by three clients at the same time (one asking for the time, one speaking Klingon and one just saying "Hello world!" or something else).

5) [Optional; results not shown here] Using a termination character (e.g., '\0') for letting the receiver know when a message ends has an important drawback: *The message itself must not contain any termination character*, otherwise the rest of the message will be ignored by the receiver. If the message contains some arbitrary, user-defined text, then it is in principle possible that the message itself does somewhere contain the termination character.
An alternative (and maybe better) approach would be: instead of *ending* the message with a termination character, it can be *started* with an indication of how long the message is (e.g., number of bytes for generic data or number of characters in text messages).
Example: instead of

"Hello world!" (12 characters)

the sender might actually transmit something like

"12;Hello world!"

to indicate that the message itself (after the semicolon) has 12 characters.

Try to implement this approach (defining a generic function which takes a message and adds its length at the beginning); both the client and the server need to be able to send such messages, receive then and be able to verify that they got the entire message.

6) RESTful web service: Implement the web service presented in the lectures

a) First create a simple web server (Hallo World!) and test it with your web browser at:
http://localhost:5000/

b) Test also it's status with CURL:
`curl -i http://localhost:5000/`

c) Implement the rest of the web service for managing gene information. Test all functions and check the status (content) of the entire gene database after each update.

d) Implement an additional function which empties the database in one go (deletes all gene entries at once) and test it.

e) Finally, replace the "Hello World" function by a function which, for the same end point (GET and same URL) provides a list all available routes (URLs defined in `@app.route`).
[Hint: try `app.url_map.iter_rules()`]