COMP 6411: ASSIGNMENT 1

Before reading the assignment description, please take note of the following:

1. *Assignments must be submitted on time in order to received full value. Appropriate late penalties (< 1 hour = 10%, < 24 hours = 25%) will be applied, as appropriate.*

2. *It is the student's responsibility to verify that the assignment has been properly submitted. You can NOT send an alternate version of the assignment at a later date, with the claim that you must have submitted the wrong version at the deadline. Only the original submission will be graded (though you can certainly re-submit multiple times before the due date).*

3. *These are individual assignments. Feel free to talk to one another and even help others understand the basic ideas. But the source code that you write must be your own.*

4. *You may use any modules found in the Python Standard Libraries. You can NOT utilize any external third-party libraries. The graders will not have these when evaluating your submissions.*

5. *You are free to use Python's OOP (Object Oriented) syntax or non-OOP. It has no impact on your grade.*

DESCRIPTION: In this assignment, you will have a chance to try your hand with the Python programming language. Your job will be to develop a small database server. Ordinarily, this would be a significant undertaking. However, with Python, this will be surprisingly straightforward and requires a relatively modest amount of code. Your DB system will work as follows:

1. You will construct a client/server application. In your case, only one client program will access the DB, so you do not have to worry about issues like concurrency or thread control. It's just a simple *one-to-one* form of communication. Python provides a `socketserver` class in its standard libraries. You can use this to get started, along with the sample code provided in the Python docs (you can use the 9999 port for the DB server).

2. Before listening for client requests, the server program must first load the database and provide access to the data (i.e., as soon as the server is started). The data base will be loaded from a simple, plain-text disk file called `data.txt`. In your case, the database will hold customer records. A customer record will be a tuple with the following format:

   ```
   first_name, age, address, phone_number
   ```

To record this information on disk, we will store one record per line, and separate each field with a *bar* symbol ('|'). A simple 3-record database might look like this:

```
john|43|123 Apple street|514 428-3452
katya|   26|49 Queen Mary Road|309 234-7654
ahmad|91|1888 Pepper Lane|
```

Note that additional *leading* or *trailing* spaces may be present (e.g., Katya's age field). These must be removed when you read the file.

In addition, basic error checking must be done when values are read, including:

a. The `first_name` field must always be present and it must consist only of alphabetic characters.
b. Other fields may not have a value at all (e.g., Ahmad's phone#). This is okay, as long as the `first_name` field is provided.
c. If `age` is present it must be an integer between 1 and 120,
d. If `address` is present it can only contain alphanumeric characters, plus spaces, periods ('.') or dashes ('-').
e. If `phone number` is present it can contain either 7 or 10 digits and must use (1) a space between the 3rd and 4th digit if it's a 10-digit number [e.g., 435 456-3295] and (2) a dash before the final four digits of the number, regardless of whether it's 7 or 10 digits [e.g., 657-4567].

If the `name` is missing, or any of the other fields contain invalid entries, the record should be ignored and an appropriate message should be display as the server is starting up:

```
Record skipped [missing field(s)]: natty|32 lane

Record skipped [invalid age field]: willy|256|Lambda Street|221-7896

Record skipped [missing field(s)]: david||456-6789

Record skipped [invalid phone field]: donald|76|123 Abe Road|5546785

Python DB server is now running...
```

For the assignment, you should create a text file with about 20 records. The marker will use a data set of the same general format for grading purposes.

3. When the server is started, the first thing it will do is load the data set into memory (The database file – `data.txt` - should be stored in the same folder as the server application). Each tuple will then be stored in an in-memory data structure. Feel free to use any standard Python data structure or combination of data structures for this purpose. In your case, you must eventually search for customers by `first_name`. So if you want to see the information for customer "john", for example, this should be a very simple operation.

4. You will also build a simple client application to access the server. The app will be located in the same folder as the server. Again, you can use the sample code on the python website as a starting point (you will likely want to use the `socket` object to connect to the `socketserver`). The idea is that you will send a request (just a text string) to the server and ask for a specific task to be performed.

   The use interface (provided by the client) will consist of the following menu:

```
Customer Management Menu

1. Find customer
2. Add customer
3. Delete customer
4. Update customer age
5. Update customer address
6. Update customer phone
7. Print report
8. Exit

Select:
```

   This may look like a lot of options, but each of these is quite trivial. The `Select` prompt at the bottom will wait for a number to be entered. When a valid menu option is provided, the client application will then take the appropriate action, as follows:

   1. **Find customer**: Prompt the user for a name and then send the name to the server. The server will respond to the client either with the full customer record or a "customer not found" message. The search itself should be case insensitive. To ensure this, all names should be maintained by the server in lower case.

      Note that the server's response MUST be sent back to the client which will then display it. The server itself should never print results directly since, in theory, it could be placed on another machine. The menu will then be displayed again so that another choice can be made.

      For example, if the user enters the name "john" (or "John"), you would either see something like:

```
Customer Name: John
Server response: john|34|ABC Street|768-3245
```
      or
```
Server response: john not found in database
```

2. **Add customer**: The client will prompt the user to enter each of the four fields. These will be sent to the server and the server will respond with either a "Customer already exists" message or a confirmation message that the customer has been added. Note that basic client-side error checking on the input must be provided, as per the constraints listed above. An appropriate message should be displayed for invalid entries and the user should be asked to enter the value again. Note that a blank age, address, or phone number is perfectly OK.

3. **Delete customer**: Delete the specified customer. The server will respond with either a "Customer does not exist" message or a confirmation message.

4. **Update** options: All update options will prompt for a `name`, then for the field to be updated. The server will respond either with a "Customer not found" message or a confirmation message. Again, error checking should be provided and a blank entry is acceptable.

5. **Print Report**: This will print the contents of the database, sorted by name. Note that the sorted contents should be sent by the server back to the client and then displayed by the client app. The `Print Report` function is **mandatory** because it is the primary way for the grader (and you) to determine if the database server is working correctly. Specifically, you can display the database before you perform any updates, then check it again later to make sure everything is working. You can NOT get a passing grade on the assignment if the `Print Report` option does not work!

   So the report might look like this:

   ```
   ** Database contents **
   ahmad|43|67 Drury Lane|897-3456
   billy|101|123 Apple Street|435 456-5768
   bob||119 Doop Road|345-5678
   donna|35|Here Road|564-6879
   john|34||768-3245
   nancy|21|Stuffy Street|768 345-7896
   sue|45|Happy Lane|456-3245
   ```

   Note that no special formatting is required for the report, as long as the individual rows and columns are readable.

6. **Exit** does the obvious thing. It should print a "Good bye" message and simply terminate the client application. Note that the server process will still be running since it executes in an infinite loop. If the client is restarted, it should reconnect with the existing server.

Finally, note that the DB Menu should always appear at the top of the computer's display. In other words, the content should not continue to simply scroll down the screen. So once the results of a selected option are shown to the user, the message "Press any key to continue…" will appear. When the user is ready to proceed, they can press a key and the screen will be cleared and the DB Menu will be shown again at the top of the screen. In practice, this is quite trivial to do on a Linux system

(i.e., the docker container). To do this, you will simply import the Python **os** module and use the `os.system('clear')` invocation in your Python program. The following image illustrates an entry for an update on customer age.

```
Customer Management Menu

1. Find customer
2. Add customer
3. Delete customer
4. Update customer age
5. Update customer address
6. Update customer phone
7. Print report
8. Exit

Select: 4
Customer Name: sue
Customer age: 345
Age must be an integer  (1 >= age <=120). Please try again...
Customer age: 45
Server response: Update: age = 45 for sue


Press any key to continue...
```

So that's the basic idea. If you haven't written Python code before, you will quickly see that it is a very accessible language with a lot of documentation and supporting materials online. If you like to code, this assignment might actually be fun.

RUNNING THE CODE: You will start the server first and then start/stop the client app. When running the server from the command line, we want to start the server as a background process so that we get a command prompt again. This will allow us to then run the client from the same command line. To run the server in the background, we simply execute the following command:

```
python3 server.py &
```

Note the trailing '&'. This instructs the shell to start server.py as a background process. If you hit the ENTER key again, you will get a new command prompt.

To stop the server, you will send a `kill` signal to the server process itself. Again, this isn't hard to do from the Linux command line. First, you will get a list of the current processes. Typing the following on the command line will work:

```
ps -aux
```

This will produce a display something like the one below

```
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.0   4472  3772 pts/0    Ss   May12   0:03 bash
root       506  0.0  0.1  12548  8604 pts/0    S    15:28   0:00 python3 server.py
root       510  0.0  0.0   7428  2612 pts/0    R+   15:32   0:00 ps -aux
```

In this case, we can see that our server has been given Process ID 506 . To stop the server, you can just type the following:

```
kill 506
```

You should always stop the server when you want to work on the code. Otherwise, you will get a networking error when you try to start the server again, as the previous server is still running and using the specified port.

NOTE: As mentioned above, the standard Python3 docs provide a simple example of a client/server interaction. You can find this online in the documentation for the `sockerserver` class. If you scroll down through the documentation, you will see a section entitled **Examples**. There you will see a tiny pair of Python code samples for the "server side" and the "client side". You can use this code almost verbatim as a starting point for your assignment. You do not have to be networking experts – this assignment is all about using Python to provide the data management functionality described above.

DELIVERABLES: Your submission will have at least two source files. Specifically, it must have a file called `server.py` and one called `client.py`. You can add other file(s) if you like. Do not include the `data.txt` file since the marker will use their own test set to ensure that all students are evaluated in the same way.

Once you are ready to submit, place the .py files into a zip file. The name of the zip file will consist of "a1" + last name + first name + student ID + ".zip", using the underscore character "_" as the separator. For example, if your name is John Smith and your ID is "123456", then your zip file would be combined into a file called a1_Smith_John_123456.zip". The final zip file will be submitted through the course web site on Moodle. You simply upload the file using the link on the assignment web page.

GRADING: Your Python source files will be downloaded and placed into a single folder on the grader's machine, along with a `data.txt` file. Grading will take place on the standard docker installation. It is expected that you will develop/test your code in this environment. If your code does not run properly in this environment, you can NOT get a full grade for the submission.

Good Luck