TEST PYTHON KNOWLEDGE – DEVELOPER I

1. File Handling and Array Operations

Write a Python script with the following methods:

- List folder contents: a function that receives a path, counts the number of elements inside, and prints them out.
- Read CSV file: a function that receives a path and a CSV filename.
  - If the file exists, the function should read it, print the number and name of columns, print the number of rows, and print the average and standard deviation per column (only for numeric columns).
  - If there is any error in the above process, e.g., non existing file, not CSV file, not numbers in CSV file, the function should use logging to print the error message.
  - (Use the provided CSV file: `./sample-01-csv.csv`)
- Read DICOM file: a function that receives a path and a DICOM filename.
  - If the file exists, use 'pydicom' to read it and print the patient's name, study date, and modality.
  - Optionally, let the function receive any amount of tag numbers (e.g., 0x0008, 0x0016) and print the contents for those tags[1].
  - If there is any error in the above process, use logging to print the error.
  - (Use the provided DICOM file: `./sample-01-dicom.dcm`)

2. Object-Oriented Programming (OOP)

Design a Python class 'PatientRecord' with the following methods:

- Store patient information such as name, age, birth date, sex, weight, patient ID, type of patient ID. Include methods to set and get this information.
- Extend the class to include a method 'update_diagnosis(new_diagnosis)' that updates the diagnosis and logs the change using logging module.

Design a Python class 'StudyRecord' that inherits PatientRecord and add the following:

- Store study information such as modality, study date, study time, study instance UID, series number, number of frames. Include methods to set and get this information
- Extend the class to load all study details from a DICOM file using 'pydicom' [1].

Overwrite the appropiate class methods to print all the patient and study information when using 'print(study_instance)'. This should show all information of the study and all information of the patient.

[1] https://www.dicomlibrary.com/dicom/dicom-tags/

(Use the provided DICOM file: `./sample-02-dicom.dcm`)

3. Multithreading and Concurrency

- Write a Python script that spawns two threads. One thread should print even numbers and the other should print odd numbers between 1 and 200, every 0.5 seconds.
- Create a Python program that receives a folder path and a JSON filename, and reads it. The structure of the JSON file is as follows:

```
{
  "1": {
    "id": "aabbcc1",
    "data": [
      "54 26 34 47 95 23 79 41 97 55",
      "75 87 75 89 81 45 52 51 79 32",
      "71 8 67 9 12 97 18 59 58 17"
    ],
    "deviceName": "RX 1"
  },
  "2": {
    "id": "aabbcc2",
    "data": [
      "32 62 18 17 60 39 97 63 64 62",
      "43 99 88 100 2 30 96 21 12 88",
      "49 31 73 24 53 63 55 57 88 72"
    ],
    "deviceName": "RX 1"
  }
}
```

- The program should create one thread to process every element of the JSON file. The program should also limit the number of active threads to 4, if there are more than 4 elements in the JSON file, new threads should be created as soons as old threads are terminated.
- The purpose of the thread is to print the element id, read the data inside each element and:
  - Normalize data from 0 to 1. This means finding the max value and the use it as normalization value.
  - Print and log the data average before and after normalization.
  - Print and log the size of the data.
- Configure logging to write to a shared log file, while ensuring thread safety.

(Use the provided json files: `./sample-03-xx-json.json`)

4. API Restful Design:

Create a Django-based RESTful API that performs CRUD (Create, Read, Update, Delete) operations for managing medical image processing results, which will be stored in a PostgreSQL database.

Requirements:

1. Models:
    - Design a Django model to represent each JSON element. The model should include fields for id, device_name, average before normalization, average after normalization, data size, and raw data (can be stored as a JSON field).
2. Endpoints:
    - Create (POST): Accept JSON payload (similar to the previous question) and store it in the PostgreSQL database. The data should be validated before storing it.
    - Read (GET): List all existing entries or retrieve a single entry by its ID.
    - Update (PUT/PATCH): Allow users to update the device_name and/or the id of an existing entry.
    - Delete (DELETE): Delete an entry by its ID.
3. Database:
    - Use PostgreSQL as the backend database. Ensure proper configuration and migrations.
4. Validation:
    - Ensure that the input data is validated and that any invalid data results in an appropriate HTTP error code.
5. Logging:
    - Log all API requests and responses, as well as any errors encountered.

Example API Endpoints:

- POST /api/elements/
    - Payload: JSON data as described above.
    - Action: Create a new entry in the database.
- GET /api/elements/
    - Action: List all entries.
- GET /api/elements/<id>/

- Action: Retrieve a specific entry by ID.

- PUT /api/elements/<id>/
  - Payload: { "device_name": "new_device_name" }
  - Action: Update the device_name or id of an existing entry.
- DELETE /api/elements/<id>/
  - Action: Delete an entry by ID.

(You can use the provided json file: `./sample-04-json.json`)

5. Distributed DICOM Processing System (Design and Architecture)

You are tasked with designing a distributed system that processes large batches of medical images (e.g., DICOM images) across multiple machines. Your design should ensure efficient task distribution, fault tolerance, logging, and scalability. The system must handle multiple concurrent image uploads, perform distributed processing, and provide a way for users to retrieve the results.

This question focuses on architecture and design rather than implementation.

- **Main Entry Point:**
  - Design the main entry point where medical images (DICOM files) are uploaded by users.
  - This entry point should accept large files, validate them, and enqueue them for processing.
- **Distributed Processing System:**
  - Design a group of servers (workers) responsible for processing the uploaded medical images.
  - Consider the load balancing and distribution of tasks among servers based on workload and usage limits (e.g., CPU, memory).
  - Ensure scalability, allowing the addition or removal of servers based on system demands.
- **Usage Limits and Resource Management:**
  - Explain how the system will monitor and manage the usage of each server to prevent overloading.
  - Propose an algorithm or strategy to efficiently assign tasks to servers based on their current load and capabilities.
- **Database Integration:**
  - Each processing server must store the results of its medical image analysis in a distributed database.

- Design how the servers will report their results to the corresponding database.
- Ensure that the database can handle large volumes of data and concurrent writes from multiple servers.
- Include a mechanism for logging all processing activities, including failures and successes, to track the system's performance.

- **Information System for Users:**
  - Users should be able to log into an information system (e.g., a web portal) to check the status of their submitted images and retrieve the results.
  - Design how the information system will interact with the distributed database to show users their processed data in real-time.
  - Include design considerations for user authentication, access control, and the UI/UX for displaying processing results.
- **Additional Considerations:**
  - How will the system ensure the security and privacy of medical data, particularly considering HIPAA or GDPR compliance?
  - What encryption methods would you use for data transmission and storage?
  - Describe how logging will be implemented across the system for monitoring the health and status of each component (e.g., task queue, worker servers, database).
  - How will the system handle errors or downtime in individual servers while maintaining overall availability?