

## EXPT-4: Data Storage for Edge Devices

**Objective:** To understand process of data storage for edge computing devices.

We will do hands-on and see how Python manages data storage **locally** using a simple file IO and databases like SQLite. The data can be stored **remotely** using databases like MongoDB, and a **time-series** database like QuestDB.

### A. Local data storage

Data can be stored permanently on the same device (the secondary storage or hard disk and so on.) where the application is running. Local data storage has some advantages like fast insert, retrieval, and easy handling. The main disadvantage is that the size and performance are dependent on the capacity of the device. The data can be stored locally either using files like CSV, JSON, XML, and so on., or databases like SQLite. Files are better when data is in a small size like in a few KBs and data insertion and retrieval is less frequent, but data-bases are good for large data sizes like in MBs or GBs with frequent insert and retrieval activities.

**Basics of file storage handling** Files can either be text or binary. Text files are directly human-readable like TXT, CSV, JSON, XML, and so on. Whereas binary files are not directly human-readable like images or video. Generally, special tools are provided for binary files like image viewers or video players. Executable files are also binarily formatted (also referred to as compiled binaries). File handling is supported by Python in a very simple and easy way, and it allows developers to handle file operations, that is, to read and write files, along with many other file handling options. Python also provides special handling for binary files like images or video via special libraries like open CV (briefly covered in the previous chapter, under section Video streaming and image processing). The following are the basic file operations as supported by Python:

**Open file** This is the main operation; the file shall be opened first before doing any reading or writing operations. In Python, there is a function **open()**, which takes two parameters: filename, and mode. There are four types of modes for opening a file as shown as follows:

- **“r” – Read:** Default mode. Opening a file only for reading and writing is not permitted. Throws error if the file does not exist.
- **“w” – Write:** Opens a file only for writing. It overwrites the existing data with new data if the file already exists, otherwise it creates the file if it does not exist.
- **“a” – Append:** Opens a file for appending, new data is appended with existing data. The file is created if it does not exist.
- **“x” - Create:** Creates the specified file, - **returns** an error if the file exists. Optionally, we can also specify the file-handling mode as either binary or text:
  - **“t” - Text** - Default value.
  - **“b” - Binary** (for example, images).

The following is the syntax for opening a file: `file_object = open("mytextfile.txt")` The above code will open the existing file in reading and text mode by default, hence, no mode is specified. The function **open()** returns the file object, if the file does not exist, then the **File Not Found** error will be shown. The preceding code can also be written as follows: `file_object = open("mytextfile.txt", "rt")` By default, the file should exist at

the same place as the Python program otherwise the file location can be mentioned as shown as follows: `file_object = open("D:/python/project2/mytextfile.txt", "rt")` **Here we are using Unix style / as a path separator, but if you want to use Windows path style then use \\ as a separator because single \ can be treated as escape sequence which can cause an error.**

**Reading the content of the file** Let us try to open and read the contents of the file using our program and print it on the terminal. To do this, first, create a new file in Notepad, add some contents and save it . Now, create a new Python file in VS Code and add the following code: `file_object = open("mytextfile.txt") print(file_object.read()) file_object.close()` Run the preceding code in the terminal, the output should be as shown in the following figure:

```
(.venv) PS D:\python\project2> py .\filehandling.py
Edge Computing with Python!
This is my local text file.
```

*Figure : Output of the Reading File Program*

The **file** object has a **read()** function which by default returns the whole file content. You can also specify how many characters to return. Change line 2 of the program to the following:

`print(file_object.read(15))` Run the program and observe the output on the terminal, **Edge Co**  
**mputing** should be printed. We should also close the file after performing our  
opera- tions by calling the **close()** function of the file object. **The file object also provides functions for reading the contents line by**

**line, that is, `readline()` and `readlines()`. The `readline()` returns one line at a time, so in our program, we need to call it two times to print both the lines of our text file. The `readlines()` function by default returns all the lines from the file, however, we can also pass the number of lines to read from the file as `file_object.readlines(1)`.**

### **Writing to the file**

A file should be opened with either **a** or **w** parameters. Try the following code to observe the behavior of **a**:

```
01. file_object = open("mytextfile.txt", "a")
02. file_object.write("This line is appended at the end!")
03. file_object.close()
04. file_object = open("mytextfile.txt", "r")
05. print(file_object.read())
06. file_object.close()
```

**Code** The following figure shows the output after running the preceding code:

```
(.venv) PS D:\python\project2> py .\filehandling.py
Edge Computing with Python!
This is my local text file.This line is appended at the end!
```

**Figure : Output with Append Mode** The following code is for showing the behavior of **w**:

```
01. file_object = open("mytextfile.txt", "w")
02. file_object.write("This line overwrites previous content!")
03. file_object.close()
04. file_object = open("mytextfile.txt", "r")
05. print(file_object.read())
06. file_object.close()

(.venv) PS D:\python\project2> py .\filehandling.py
This line overwrites previous content!
```

*Figure : Output with Write Mode*

As already mentioned, both `a` and `w` create new files if files do not exist. The readers can also try to open a file with `x` mode which is used only for creating new files. Also, notice the usage of `close()` function in both the preceding programs.

### **Deleting, renaming, and other operations on the file**

For these operations we can use functions of Python's `os` library, as shown as follows:

#### **•Deleting a file**

```
import os
os.remove("mytextfile.txt")
```

#### **•Renaming a file**

```
import os
os.rename("mytextfile.txt", "renamed_mytextfile.txt")
```

#### **•Check if the file exists and then delete it**

```
import os
if os.path.exists("mytextfile.txt"):
    os.remove("mytextfile.txt")
else:
    print("Invalid file!")
```

**Using file storage in the program** Let us make use of the file storage for enhancing our weather-

monitoring program. Create a new file in VS Code, name it like '**config.ini**', in editor enter the base **url**, **api key**, and **units** as shown as follows:

```
BaseURL #
http://api.openweathermap.org/data/2.5/weather APIKey #
"xxxxxxxxXXXXXXXXXXXXXXXXXXXXXXXXxxxxx" Units #
```

metric Now, open the **weather-status.py** program in VS Code and do the code updates as shown as follows:

```
06. BASE_API_URL = ""
07. APIKEY = ""
08. UNITS = ""
09.
10. def init_config():
11.     global BASE_API_URL
12.     global APIKEY
13.     global UNITS
14.
15.     file_object = open("config.ini", "r")
16.     line_content = file_object.readline()
17.     BASE_API_URL = (line_content.split('#')[1]).removesuffix('\n').strip()
18.     line_content = file_object.readline()
19.     APIKEY = (line_content.split('#')[1]).removesuffix('\n').strip()
20.     line_content = file_object.readline()
21.     UNITS = (line_content.split('#')[1]).removesuffix('\n').strip()
22.     file_object.close()
23.     return
```

We have added three global variables in lines 6, 7, and 8. A new function **init\_config()** is defined from lines 10, to 23. This function opens the **config.ini** file and parses all the lines to initialize the respective global variables. Also, update the function **make\_api\_url()** as follows:

```
34. def make_api_url(city_name):
35.     api_key = APIKEY
36.     return (f"(BASE_API_URL)?q={city_name}&units={UNITS}&appid={api_key}")
--
```

You can notice how we have used all three global variables here. You also need to call **init\_config()** function before any other function and then run the program in the terminal.

**Advantage of the file** You can notice that it will be easy to update those three variables in future without changing the actual code, thus, reducing the maintenance efforts. We can also put more information in this file to enhance our code, for example, the help or error text that is being used as plain string can also be included in the file and used in the code via separate variables. But, as we are using a plain text file, there is a problem, we need to take care of complex parsing each line by splitting, removing unwanted suffixes or characters and so on, so the chances of errors are higher. To reduce these complex parsing efforts and errors we can make use of formatted text like JSON, XML and so on. The JSON is now widely used by web-based applications, also popular for Edge apps, and covered in the next section.

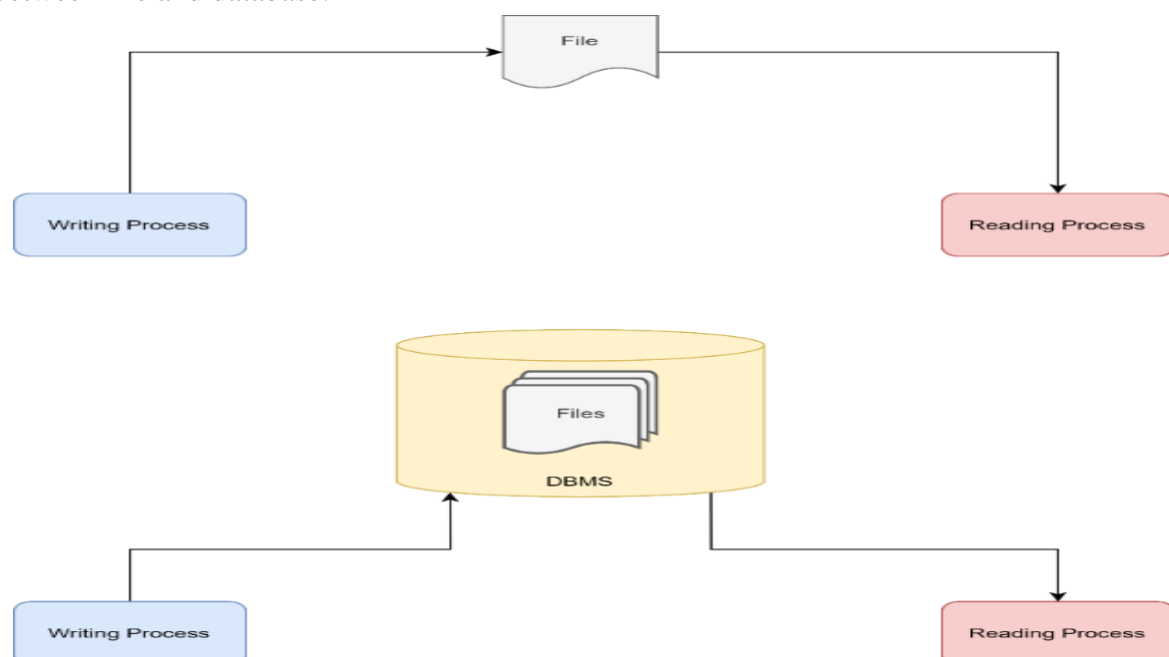
**JSON files: JavaScript Object Notation (JSON)** is a text format for storing and transporting data between computers. It is *self-described*, easy to understand and language independent. As the format is text only, the data can easily be sent between computers, and used by any programming language. A JSON object starts with { and ends with }, and the data is represented as key/value pair inside the braces. The following is an example of JSON object: {"name": "Harry", "height":172, "car": null} The previous object has three properties separated by comma; name, height, car, and each property has a value. The value can be simple like a string, a number, a Boolean, null or complex like an object, an array, a function, or date and so on. As JSON is represented in simple text or string, it can be easily stored in text files, and we can save these text files with **.json** extension for better identification, however this extension is not mandatory. The **config.ini** file which we created in previous section can be written as follows in JSON format: { "BaseURL": "http://api.openweathermap.org/data/2.5/weather", "APIKey": "xxxxxxxxxxxxxxxxxxxxxxxxxxxx", "Units": "metric" } Python provides **json** library for easy handling with JSON formats. The following is the updated function **init\_config()**:

```
04. import json
05.
06. BASE_API_URL = ""
07. APIKEY = ""
08. UNITS = ""
09.
10. def init_config():
11.     global BASE_API_URL
12.     global APIKEY
13.     global UNITS
14.
15.     file_object = open("config.ini", "r")
16.     file_content = file_object.read()
17.     config_json = json.loads(file_content)
18.     BASE_API_URL = config_json["BaseURL"]
19.     APIKEY = config_json["APIKey"]
20.     UNITS = config_json["Units"]
21.     file_object.close()
22.     return
```

In line 17, the whole content is converted as JSON object and then value of each property is being accessed via property name in lines 18, 19 and 20. With this, our program becomes simpler. You can also notice that the weather data returned by the API is also in JSON format, and we are accessing its various properties in the same manner as shown earlier. Look at these functions; `acquire_weather_data()`, and `process_and_output_weather_info()`. It is also possible to convert Python dictionary into JSON object. The dictionary is passed as parameter to `json.dumps()` function which returns the string representation of JSON, this string can further be converted into JSON object by calling `json.loads()` function. There is another function `json.dump()`, it takes two parameters, first is dictionary and second is file object, it converts the dictionary into JSON representation and writes to the respective file.

### ***B. Database (DB) storage***

Till now, we have seen file storage that is good for storing small amounts of data like configuration. But, when it comes to storing process related data which needs to be stored and retrieved on a continuous basis, then files are not the right solution. As we know that with files, we can either read or write at a time and not do both simultaneously and efficiently. For example, imagine the situation where one process gets the weather data every second and stores it in a file in time-series manner, and there is another process which wants to read this time-series weather data every second and plot on the trend view. There will be too many read/write operations happening on the same file. When simultaneous read/write operations are required on the file, and the size of file also increases over time, then its read/write performance goes down. Even if we try to handle it using threads or concurrent programming and complex logic, it is like re-inventing the wheel because databases are designed to handle this type of data and operations (also referred as *transactions*) in an efficient manner. The following figure 3.2.2 shows the difference between file and database:



***Figure : File Vs Database***



An organized collection of structured data is referred to as database, which is stored in a computer system. A **Database Management System (DBMS)** or also referred to as database engine) is used to control the database.

Let us start with SQLite DB which is specially designed for low powered devices and acts as a local data storage.

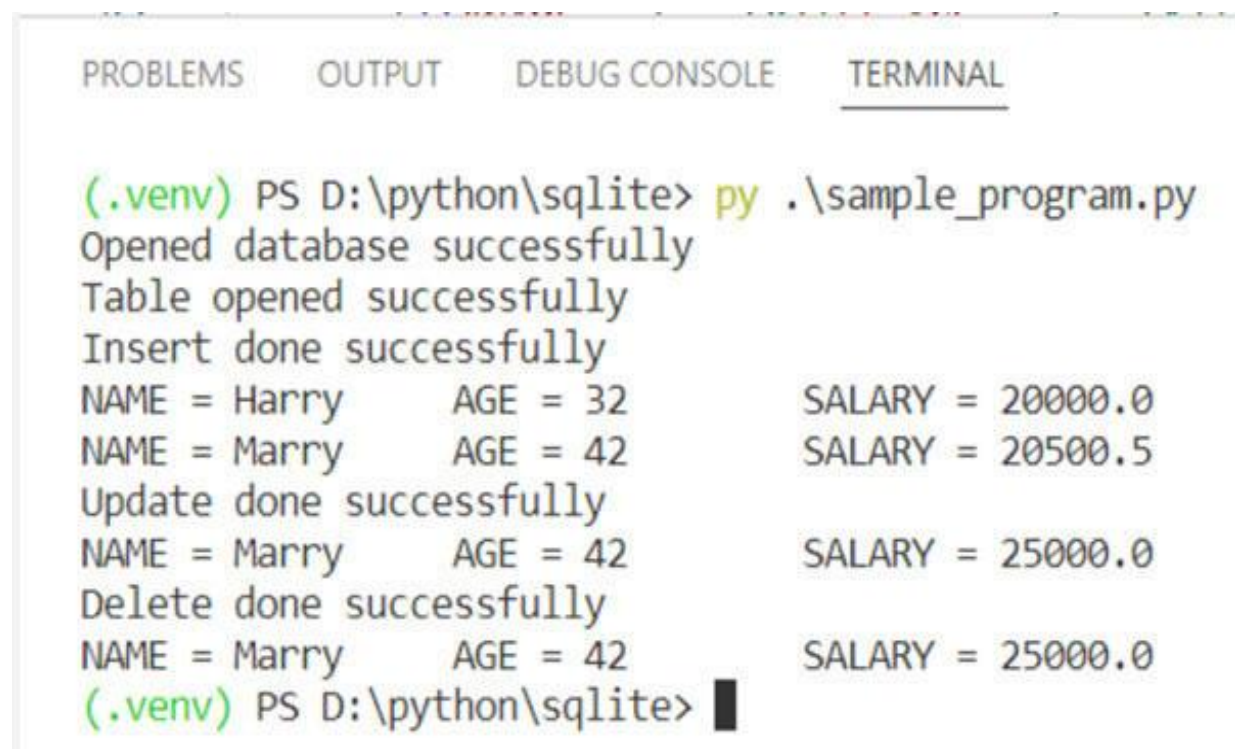
**SQLite DB in Python:** SQLite is a library that provides a small, fast, reliable, serverless, zero-configuration, and transactional SQL database engine. It is the most widely used database engine in the world and it is open source. More detailed information can be found from the official

link <https://www.sqlite.org/about.html>. Python has **sqlite3** library which provides an SQL interface for handling DB operations. This library is available by default with Python version 2.5.x onwards, as we are using Python version 3.10.2, hence **sqlite3** is already available. Let us write a simple program which demonstrates key operations of database using **sqlite3**. Open VS Code, create a new file, name it like **sqlite\_example.py**, and enter the following code:

```
01. import sqlite3
02.
03. conn = sqlite3.connect('my_database.db')
04. print("Opened database successfully")
05.
06. conn.execute("CREATE TABLE IF NOT EXISTS Person \
07. (NAME TEXT, AGE INT, SALARY REAL)")
08. print("Table opened successfully")
09.
10. conn.execute("INSERT INTO Person (NAME, AGE, SALARY) \
11. VALUES ('Harry', 32, 20000.00)")
12. conn.execute("INSERT INTO Person (NAME, AGE, SALARY) \
13. VALUES ('Marry', 42, 20500.50)")
14. print("Insert done successfully")
15.
16. cursor = conn.execute("SELECT name, age, salary from Person")
17. for row in cursor:
18.     print(f"NAME = {row[0]}\t AGE = {row[1]}\t SALARY = {row[2]}")
19.
20. conn.execute("UPDATE Person set SALARY = 25000.00 where NAME = 'Marry'")
21. conn.commit()
22. print(f"Update done successfully")
23.
24. cursor = conn.execute("SELECT name, age, salary from Person \
25. where NAME = 'Marry'")
26. for row in cursor:
27.     print(f"NAME = {row[0]}\t AGE = {row[1]}\t SALARY = {row[2]}")
28.
29. conn.execute("DELETE from Person where NAME = 'Harry'")
30. conn.commit()
31. print(f"Delete done successfully")
32.
33. cursor = conn.execute("SELECT name, age, salary from Person")
34. for row in cursor:
35.     print(f"NAME = {row[0]}\t AGE = {row[1]}\t SALARY = {row[2]}")
36.
37. conn.close()
```

The program is explained as follows:

- Line 3:** The **connect()** function either opens an existing database or creates a new one. A file named **my\_database.db** will be created in the same folder where the program exists.
- Line 6:** Executing SQL **CREATE** statement to create a table with name **Person** if it does not exist (otherwise, nothing will be done) with three columns **NAME**, **AGE**, **SALARY** with respective data types mentioned, that is, **TEXT**, **INT**, **REAL**.
- Line 10, 12:** Executing SQL **INSERT** statements for inserting rows (or records) with values of respective columns of the table.
- Line 16 to 18:** Executing SQL **SELECT** statement which returns collection of all rows into the cursor object. Then iterating the rows collection via cursor and printing each column's value (the **row[index]** represents the column value of the respective record).
- Line 20, 21:** Executing SQL **UPDATE** statement to modify the **SALARY** for the person **Marry**. The call to **commit()** function makes sure that the updates (or transactions) are permanently written to database. If you close the database without committing, then the updates will be lost.
- Line 24 to 27:** Just query the database to get the record for person **Marry** and print to see the updated value.
- Line 29:** Executing the SQL **DELETE** statement to remove the record for the person **Harry**
- Line 33 to 35:** Again, just query all the records and print values to see the effect of **DELETE**.
- Line 37:** Closing the database to stop any other operations just like we do for files. When you run this program in terminal the output should be as shown in following figure:

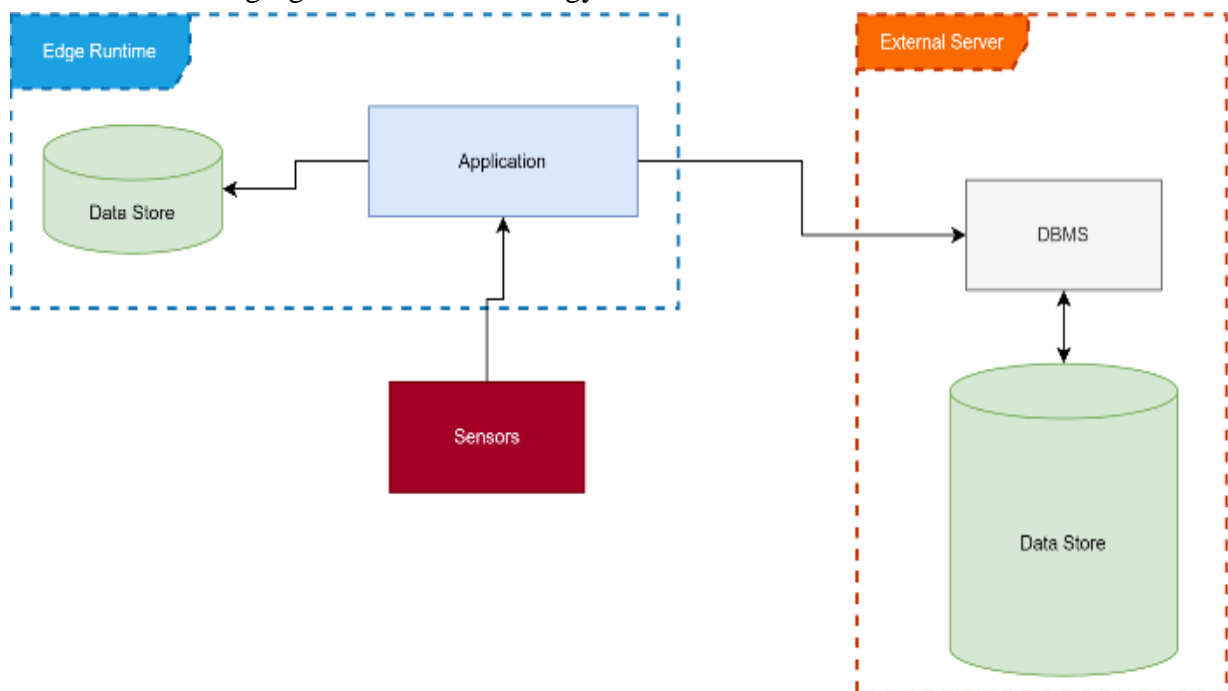


```
(.venv) PS D:\python\sqlite> py .\sample_program.py
Opened database successfully
Table opened successfully
Insert done successfully
NAME = Harry      AGE = 32          SALARY = 20000.0
NAME = Marry      AGE = 42          SALARY = 20500.5
Update done successfully
NAME = Marry      AGE = 42          SALARY = 25000.0
Delete done successfully
NAME = Marry      AGE = 42          SALARY = 25000.0
(.venv) PS D:\python\sqlite>
```

**Figure :** Output of SQLite Program

### C. Remote data storage

When the size of the data increases over time, the local storage runs out of space. Mostly it is not feasible to upgrade the storage space of the Edge device. So, the data is stored outside the device on an external storage which is generally hosted on a dedicated server machine. These servers are equipped with larger storage spaces in TBs and are also easily upgradable. So, the Edge devices store short term data into their local storage and the long-term data is transferred to the remote storage. For example, let us assume we have an Edge device with a 30 GB hard disk (consider this as a free space only for application data) and lots of connected sensors. These sensors generate 1 GB of data per day, and for analytics and business reasons we need to hold the data for 1 year. This Edge device can hold data of max 30 days or 1 month, but we need storage for 12 months, that is, more than 365 GBs. In this case, the strategy is to keep only the latest 30 days of data on Edge device and use external storage for storing the 1-year data. The following figure shows this strategy.



**Figure:** Output of SQLite Program

The database engine which is hosted on external machine provides a communication layer so that remote clients can connect over network protocols like TCP, HTTP, Web Sockets and so on. and access the database. Mostly all the enterprise grade databases engines in the market are meant for re- mote access. For example, Oracle, MySQL, MS-SQL, MongoDB, QuestDB, InfluxDB, CrateDB and so on. However, SQLite does not provide any communication layer for remote ac- cess, but we can develop a REST API based communication layer over HTTP for SQLite.