**Python Data Persistence**

## **Introduction**

Most applications today accept data from users, process it, and perform specific tasks. Some applications, however, require **data persistence** to function properly. Python provides various modules and methods for achieving data persistence. But first, what does *data persistence* mean?

## **Data Persistence**

Persistence refers to the continuation of an effect even after its cause is removed. For example, a fire may leave behind lasting effects even after the flame is extinguished. Similarly, **data persistence** ensures that data continues to exist even after an application is terminated. This persistent data can then be reused by the same or other applications.

To achieve data persistence, the data must be saved in **non-volatile storage** (e.g., hard drives or SSDs). In Python, there are several ways to do this, starting with basic **file handling methods**, also known as **File APIs**.

## **File API**

Python allows storing data in simple .txt files, which can be handled using the following:

1. **Opening Files**
   * The open() function is used to open a file in read ('r'), write ('w'), append ('a'), or read/write ('r+') mode.
2. **Reading Files**
   * Data can be read line by line using readline() or all at once as a list of strings using readlines().
3. **Writing to Files**
   * Data can be written line by line using writeline() or all at once with writelines().
4. **Closing Files**
   * After performing operations, files are closed using the close() method to release system resources.

While these methods are simple and effective for basic data storage, they are not well-suited for handling **complex data structures** (e.g., arrays, dictionaries).

**Bytes Handling**

The File API can also store basic data types like integers, strings, and floats in **byte format** for efficient storage and retrieval. However, storing complex data structures directly (e.g., dictionaries or lists) requires additional steps, as the File API does not support this natively.

**Simultaneous Read/Write Operations**

The File API allows **simultaneous reading and writing** on the same file object using modes like 'r+'. The seek() function can be used to reset the file pointer to the beginning, allowing sequential read/write operations.

However, the File API is inherently **sequential** and does not support **random access** efficiently, making it unsuitable for applications requiring rapid or indexed data retrieval.

**Limitations of File API**

While the File API provides a simple way to achieve data persistence, it is not recommended for complex applications due to:

* Limited support for complex data structures.
* Inefficient handling of large datasets.
* Sequential-only access, which restricts flexibility for advanced use cases.

This was first method however there are other methods that overcome most of this limitation but let us continue with methods that are closely related to this one. One of them is using os module which is almost the same as File API so let’s explore it.

## **OS Module**

The os module is a built-in Python library that provides low-level file handling functionality, offering tighter integration with the operating system. While it shares similarities with the File API, it introduces key differences:

**Key Features of the OS Module**

1. **File Descriptors**
   * Unlike the File API, where open() returns a file object, the os.open() method returns a file descriptor. A file descriptor is an integer representing the state of the file, which can be used for further operations.
2. **File Operations**
   * The os module offers functions such as os.read() for reading, os.write() for writing, and os.lseek() for adjusting the file pointer. These operations are directly tied to the file descriptor.
3. **Low-Level Access**
   * The OS module enables low-level file operations that are closer to the operating system, making it suitable for system-level tasks.

**Comparison with the File API**

* **File API:**
  + Object-oriented, high-level, and suitable for most applications.
  + Methods are called on the file object (e.g., file.read()).
* **OS Module:**
  + Procedural, low-level, and better for system-specific tasks.
  + Functions are called using file descriptors (e.g., os.read(fd)).

While the OS module provides more control, it is generally less user-friendly and used only for specific needs.

**Object Serialization**  
The File API and OS module share a limitation: they cannot directly store complex Python objects (e.g., dictionaries, lists). Instead, data must be converted to strings or bytes before storage, making retrieval cumbersome. Serialization resolves this issue.

**What is Serialization?**  
Serialization refers to converting a Python object into a byte stream that can be stored or transmitted. Deserialization is the process of reconstructing the original object from the byte stream. Serialization simplifies the storage and retrieval of complex objects without manual conversion.

## **Pickle Module**

The pickle module in Python enables serialization and deserialization but uses a Python-specific format, making it unsuitable for cross-platform or non-Python applications.

**Key Functions in Pickle**

1. **In-Memory Serialization**
   * dumps(obj) converts a Python object into a byte string.
   * loads(byte\_string) reconstructs the object from the byte string.
2. **File-Based Serialization**
   * dump(obj, file) serializes an object and writes it to a file.
   * load(file) reads a serialized object from a file and reconstructs it.
3. **Advanced Serialization**
   * The Pickler and Unpickler classes allow custom serialization processes.

**Limitations of Pickle**

* **Python-Specific Format:** Data cannot be deserialized by non-Python applications.
* **Security Risks:** Loading data from untrusted sources can execute arbitrary code.

While Pickle is a powerful tool for Python applications, it is not ideal for scenarios requiring interoperability or enhanced security.

## **Marshal Module**

The marshal module is a specialized, code-centric module in Python. Unlike general-purpose data storage solutions, it focuses on performing read/write operations on Python's compiled modules (.pyc files). However, it has some limitations, including incompatibility across Python versions, which restricts its use for long-term data storage.

**Key Features of the marshal Module**

1. **Purpose**:
   * Primarily designed for handling Python-specific data like compiled code objects. It is not intended for general-purpose data persistence.
2. **Methods Overview**:
   * Similar to the pickle module, it uses dump() and load() methods for serialization and deserialization.
   * Additionally, it introduces the compile() method, which can convert Python source code into code objects.
3. **Functionality**:
   * The marshal module leverages Python’s built-in functionality to compile source strings into code objects.
   * These code objects can then be serialized using marshal.dump() into a file, enabling storage and later retrieval for execution.

**Example Workflow**

1. **Compilation**:
   * The compile() function translates Python source code into a code object that embeds Python instructions.
2. **Serialization**:
   * The resulting code object can be serialized to a file using marshal.dump().
3. **Execution**:
   * The serialized object can be deserialized using marshal.load() and executed, preserving the behavior of the original code.

**Limitations**

* **Version Dependency**:
  + marshal-based data is tightly coupled to the Python version in use, meaning data serialized in one version may not be readable in another.
* **Specialized Usage**:
  + The module is not suitable for storing general-purpose data or application-level objects.

**Use Case**

The marshal module is best suited for scenarios where temporary storage and retrieval of compiled Python code are required, such as internal tasks related to Python interpreters or quick execution of dynamically generated code.

## **Shelve Module**

The **Shelve module** simplifies data handling by allowing files to function like Python dictionaries. It provides a convenient way to store and retrieve data using keys and values, making it suitable for small to medium-scale applications requiring simple data storage.

* **Keys and Values:**
  + Keys must be strings.
  + Values can be any object supported by Python, including custom objects, pre-defined objects, or lists.
* **Functionality:**
  + The module uses a dictionary-like interface, allowing operations such as adding, updating, and deleting data to be performed with ease.
* **Dependency on Pickle:**
  + Shelve relies on the Pickle module to serialize and deserialize data.
  + As a result, shelved files are Python-specific and cannot be shared directly with non-Python applications.
* **Limitations:**
  + Shelve does not support concurrent write operations without proper locking mechanisms.
  + It is not ideal for high-performance or large-scale data storage but works well for lightweight use cases.

## **Database Manager (DBM)**

The **Database Manager (DBM)** module provides a way to store key-value pairs in a persistent, dictionary-like structure. It is suitable for small-scale data storage needs and offers the following features:

* **Keys and Values:**
  + Both keys and values are stored as binary-coded strings.
  + This format allows for efficient storage and retrieval but may require encoding and decoding for non-string data.
* **Dictionary-Like Functionality:**
  + The DBM object behaves like a Python dictionary.
  + Common dictionary operations such as adding, updating, and deleting data are supported.
* **Persistence:**
  + Unlike standard dictionaries, DBM stores data persistently on disk, enabling the data to persist between program runs.
* **Use Cases:**
  + DBM is ideal for lightweight applications requiring simple key-value storage.
* **Limitations:**
  + It is not suitable for large-scale or complex data storage needs.
  + It lacks advanced features like relationships or indexing found in full-fledged database systems.

## **CSV Module**

The CSV (Comma-Separated Values) module simplifies handling tabular data, commonly used for importing and exporting files to and from spreadsheets. It supports reading and writing operations, especially with Excel spreadsheets, and handles data using different dialects (standards for formatting).

#### Key Features

1. **Reading and Writing:**
   * The Writer() class writes rows to a CSV file.
   * To prevent extra blank lines, the newline='' parameter is used when opening files.
2. **Reading CSV Files:**
   * The reader() function reads CSV files and returns an object, which can be converted into a list for easier handling.
3. **Default Dialect:**
   * The module uses Excel as its default dialect but allows customization.
   * Dialects are predefined standards for CSV formatting, and available dialects can be listed using list\_dialects().
4. **Dictionary Support:**
   * The module supports writing dictionaries to CSV files, similar to the shelve module, but does not support saving custom objects directly.

JSON Module:

This module is the one we are gonna use for many of its advantages lets first get to know what does it stand for. JSON stands for JavaScript Object Notation . lets see what are its advantages:

It is lightweighted format suitable for simple applications like ours.

It is independent of programming languages allowing cross platform.

It is also used in web development making it suitable for web development feature that we will add on later.

It is similar to pickle but not python specific making it easier to work with.

Implemented by many languages making it universal standards.