Building a File-Based CRUD Application in Python: A Step-by-Step Guide

1.0 Introduction

This document provides a technical breakdown of the four core CRUD (Create, Read, Update, Delete) functions and their supporting utility functions from a Python application that uses a local text file for data persistence. The analysis covers the logic, file handling methods, and error handling strategies employed in each function.

2.0 Foundational Utility Functions

The core CRUD operations rely on two essential utility functions for interacting with the data file (`cities.txt`): `read_all_records` and `write_all_records`. These functions abstract the file I/O operations, ensuring data is read and written consistently.

2.1 `read_all_records()`

- **Objective:**

  This function is designed to read all records from the `cities.txt` file and return them as a list of strings.

- **File Handling:**

  It uses the `with open(file_name, 'r')` statement to safely open the file in read-only mode ('r'). This approach ensures the file is automatically closed even if errors occur.

- **Data Processing:**

  A list comprehension (`[line.strip() for line in file if line.strip()]`) is employed for efficient data processing. It iterates through each line in the file object, first using `.strip()` to remove any leading or trailing whitespace. A conditional check (`if line.strip()`) is then performed on the stripped line, ensuring that both empty

lines and lines containing only whitespace are filtered out and excluded from the final `records` list.

- **Error Handling:**

  The file-opening logic is wrapped in a `try...except` block. If a `FileNotFoundError` occurs (meaning the data file does not yet exist), the function gracefully handles it by returning an empty list, preventing the application from crashing on its first run.

2.2 `write_all_records()`

- **Objective:**

  This function is designed to write an entire list of records to the data file, truncating the file before writing the new content.

- **File Handling:**

  It utilizes the `with open(file_name, 'w')` statement. The write mode ('w') is crucial here, as it erases the file's existing contents before writing the new data, ensuring the old dataset is completely replaced.

- **Data Processing:**

  The function iterates through the `records` list provided as an argument. For each `record`, it performs a single `file.write()` operation, writing the `record` string concatenated with a newline character (`\n`). This process ensures each record is written on its own line, maintaining the file's structure.

- **Interdependency:**

  This function is critical for the `update_record` and `delete_record` operations. By overwriting the entire file in one operation, this design ensures that changes are atomic from the file's perspective. The file is either in its old state or its new state, preventing data corruption that could arise from partial writes.

3.0 Core CRUD Function Analysis

This section provides a detailed analysis of the four primary functions that implement the Create, Read, Update, and Delete operations.

The **Update** and **Delete** functions operate on a *"Read-Modify-Write"* pattern: they first read the entire file into an in-memory list, perform the modification on

that list, and then write the entire modified list back to the file, overwriting the original.

3.1 Create Function

- **Logic:**

  This function is responsible for adding a single new record to the data file.

- **File Handling:**

  It uses the `with open(file_name, 'a')` statement to open the file. The append mode ('a') is specifically chosen to ensure that the new record is added to the end of the file without erasing any existing data.

- **Implementation:**

  The process involves a single `file.write()` call that writes the `new_record` string concatenated with a newline character (`\n`) to the file. This maintains proper line-by-line formatting for all records.

  This use of append mode makes adding new records highly efficient, as it avoids the need to read and rewrite the entire file—a key difference from the update and delete operations.

3.2 Read Function

- **Logic:**

  This function reads and displays all current records to the user in a formatted, numbered list, making the data human-readable and easy to reference.

- **Interaction:**

  The function's first step is to call the `read_all_records()` utility to retrieve all data from the file into a local list variable named `records`.

- **User Interface Logic:**

  It uses a `for` loop with the `enumerate` function to iterate over the `records` list. This provides both the index and the value for each item in the list.

- **Numbering Detail:**

  Because `enumerate` produces a zero-based index, the function adds one to the index (`index + 1`) within the `print` statement. This presents a user-friendly,

1-based list (e.g., "1. Bhopal", "2. Indore"), which is more intuitive for selecting records to update or delete.

- **Return Value:**

  The function returns the `records` list. This return value is a crucial part of the application's workflow, as it provides the `update` and `delete` functions with the exact, up-to-date list of records that the user is currently viewing, ensuring data consistency for the subsequent operation.

3.3 Update Function

- **Logic:**

  Adhering to the "Read-Modify-Write" pattern, the update process involves loading the full dataset into memory, modifying a specific item, and then persisting the entire modified dataset back to the file.

  1. **Load Records into Memory:**

     The function first calls `read_all_records()` to load all current records from the file into a list.

  2. **Index Adjustment:**

     The `display_records` function presents a 1-based numbered list to the user for intuitive selection (e.g., '1. Bhopal'). However, Python lists are 0-based. This step is critical to translate the user's input (e.g., `index = 1`) into the correct internal list index (`idx = 0`) by subtracting one. All subsequent list operations use this adjusted `idx`.

  3. **Validation:**

     An `if` condition (`0 <= idx < len(records)`) validates that the adjusted `idx` is a valid index within the bounds of the `records` list.

  4. **In-Memory Update:**

     If the index is valid, the record at that position is replaced with the `new_value` using the assignment `records[idx] = new_value`.

  5. **Persist Changes:**

     Finally, the function calls `write_all_records(records)`, passing the modified list to overwrite the data file with the updated information.

- **Error Handling:**

If the user-provided index is outside the valid range, the `if` condition fails, and the function prints an "Invalid record number" error to the console.

3.4 Delete Function

- **Logic:**

  The deletion process follows the same "Read-Modify-Write" pattern as the update logic, involving an in-memory modification followed by a complete file rewrite.

  1. **Load Records into Memory:**

     Like the update function, it begins by calling `read_all_records()` to get the current list of records.

  2. **Index Adjustment:**

     It converts the user's 1-based `index` to a 0-based `idx` by subtracting one (`idx = index - 1`). This translation is necessary to align the user-facing number with Python's internal list indexing.

  3. **Validation:**

     An identical `if` condition is used to confirm that `idx` is a valid index for the current `records` list.

  4. **In-Memory Deletion:**

     If the index is valid, the record at that position is removed from the list using the `records.pop(idx)` method.

  5. **Persist Changes:**

     The final step is to call `write_all_records(records)` with the modified (now smaller) list. This overwrites the file with the remaining records, effectively deleting the selected one.

- **Error Handling:**

  If the provided index is out of bounds, the function prints an "Invalid record number" error to the console.

4.0 Building the Complete Application

4.1 Understanding CRUD

The CRUD acronym represents the four basic functions necessary to create and manage persistent data.

- **C - Create:** Add or create new records.
- **R - Read:** Retrieve or view existing data.
- **U - Update:** Modify or edit existing records.
- **D - Delete:** Remove existing records.

These four operations form the foundation of all database and data-handling applications.

4.2 Designing the Application Architecture

The application's architecture includes:

1. **Command-Line Interface (CLI)** for user interaction.
2. **Central Loop** using a `while True:` structure to maintain interactivity.
3. **Menu-Driven Choices** offering five actions (Add, View, Update, Delete, Exit).
4. **File-Based Storage** using a simple text file `cities.txt` for data persistence.

4.3 Core File I/O Functions

The file I/O functions ensure data persistence across sessions by reading and writing to `cities.txt`.

Example Implementation:

```
FILE_NAME = "cities.txt"
```

Reading Records

```
def read_all_records():
    try:
        with open(FILE_NAME, 'r') as file:
            records = [line.strip() for line in file if line.strip()]
            return records
    except FileNotFoundError:
```

```
        return []
    except Exception:
        return []
```

Writing Records

```
def write_all_records(records):
    try:
        with open(FILE_NAME, 'w') as file:
            for record in records:
                file.write(record + '\n')
    except Exception as e:
        print(f"An error occurred while writing to the file: {e}")
```

4.4 CRUD Function Implementations

Create

```
def create_record(new_record):
    try:
        with open(FILE_NAME, 'a') as file:
            file.write(new_record + '\n')
        print("Record added successfully.")
    except Exception:
        print("Could not create a new record.")
```

Read

```
def display_records():
    records = read_all_records()
    if not records:
        print("No records found.")
        return []

    print("\n--- Current Records ---")
    for index, record in enumerate(records):
        print(f"{index + 1}. {record}")
```

```
    print("-----------------------")
    return records
```

Update

```
def update_record(index, new_value):
    records = read_all_records()
    idx = index - 1

    if 0 <= idx < len(records):
        records[idx] = new_value
        write_all_records(records)
        print("Record updated successfully.")
    else:
        print("Error: Invalid record number.")
```

Delete

```
def delete_record(index):
    records = read_all_records()
    idx = index - 1

    if 0 <= idx < len(records):
        records.pop(idx)
        write_all_records(records)
        print("Record deleted successfully.")
    else:
        print("Error: Invalid record number.")
```

4.5 Application Control Flow (Main Function)

```
def main():
    while True:
        print("\n--- City Records ---")
        print("1: Add New Record")
        print("2: View All Records")
```

```
        print("3: Update a Record")
        print("4: Delete a Record")
        print("5: Exit")

        try:
            choice = int(input("Enter your choice: "))
        except ValueError:
            print("Invalid input. Please enter a number.")
            continue

        match choice:
            case 1:
                new_item = input("Enter the new city name: ")
                if new_item:
                    create_record(new_item)
                else:
                    print("Warning: Record cannot be empty.")

            case 2:
                display_records()

            case 3:
                records = display_records()
                if records:
                    try:
                        record_num = int(input("Enter the record number to update:
"))
                        new_value = input("Enter the new city name: ")
                        if new_value:
                            update_record(record_num, new_value)
                        else:
                            print("Warning: Record cannot be empty.")
                    except ValueError:
                        print("Invalid input. Please enter a number for the record.")

            case 4:
                records = display_records()
                if records:
```

```
                try:
                    record_num = int(input("Enter the record number to delete:
"))
                    delete_record(record_num)
                except ValueError:
                    print("Invalid input. Please enter a number for the record.")

            case 5:
                print("Thank you, Goodbye!")
                break

            case _:
                print("Invalid choice. Please retry.")
```

4.6 Entry Point

```
if __name__ == "__main__":
    main()
```

Explanation:

This ensures that the `main()` function only executes when the script runs directly, not when imported into another module. It's a Python best practice for modular and reusable code design.

Would you like me to convert this structured version into a **formatted PDF report** (with proper headings, fonts, and indentation) for documentation or presentation purposes?