Academic Journal of Science and Technology
ISSN: 2771-3032 | Vol. 10, No. 3, 2024

Analysis and Implementation of a Simple Database Management Software Based on Python and Excel

Shanwen Lei

Philippine Christian University, 1004, Philippines

Abstract: This paper aims to address the challenges faced by small and medium-sized enterprises (SMEs) and individual users in database management. It adopts a combination of Python and Excel to simplify the operational process and enhance the user interface, thereby reducing the learning curve and technical requirements for managing databases. Users can effortlessly manage their databases, leading to improved work efficiency. This project breaks through the limitations of traditional database management software, offering a simple, efficient, and low-resource solution.

Keywords: Database management software, Python, Data export, Data import.

1. Introduction

In the current era of rapid information technology development, effective management of data information in databases has become a crucial issue. For large-scale enterprises, they usually can allocate a large amount of manpower and resources for data management or hire professional teams to tackle the complexity and maintenance challenges of data management.

However, for small and medium-sized enterprises (SMEs) and individual users, there is often insufficient resources to establish a scalable database management team. Although existing database maintenance software can simplify the management process, users still need to have a certain level of database management knowledge and operational skills to use them proficiently, which increases the learning curve and usage threshold for users.

Therefore, a simple, efficient, and low-resource database management solution is more important and necessary for these users.

2. Functional Implementation Analysis

To achieve a simple, efficient, and low-resource database management solution, researchers feel that the following issues need to be addressed, combined with existing database management software:

First, different types of database management software have different functions and operation interfaces, and often require the use of SQL language for operation, which requires users to have a certain level of database management knowledge and operational experience to use proficiently, increasing the learning and usage threshold for users.

Second, database management software usually consumes a lot of system resources, and most database management software is usually client-server architecture, requiring connection to the database server for management and operation.

Third, the cross-platform compatibility of database management software is poor, and many commercial database management software only supports specific operating systems, which increases the complexity and cost of use for users who use other incompatible operating systems.

Fourth, commercial database management software often requires payment or subscription, which will be a significant investment for users with limited budgets.

How to solve these problems? Researchers analyze from the perspectives of simplifying database management operations, reducing the learning curve, and improving work efficiency, and decide to use the combination of Python+Excel to solve these problems faced by small and medium-sized enterprises or individual users in the database management process.

First, through Python scripts to achieve database connection, data query, and processing operations. This is attributed to Python's rich third-party libraries and tools resources. Using these libraries and tools can help users simplify the database management process and reduce the complexity of operations.

Second, using Excel as the main tool for data display, analysis, and processing. It has an intuitive and friendly user interface and powerful data processing capabilities. Users can perform operations such as data entry, editing, and viewing without mastering complex database management techniques, thereby improving the convenience and flexibility of operations.

This combination solution not only meets various needs of users for database data management but also reduces the difficulty for users in learning and usage, improves work efficiency, and is a simple, efficient, and low-resource database management solution.

According to the proposed solution, to meet the needs of users, the software should mainly implement the following functions:

Database connection function:Allow users to connect mainstream database resources easily through configuration parameters to ensure the universality and applicability of the software.

Database export to Excel function:Allow users to select specified data tables from the database, export the data tables to Excel files, and save them in the specified file path specified by the user.

Excel import to database function: Allow users to import data from their specified Excel files into the database and save them in the specified data table.

Simple user interface: Design an intuitive and concise user interface to simplify the user operation process and reduce the learning and usage threshold for users.

With the above functions, combined with Excel's powerful data processing capabilities, such functions are sufficient to meet the basic needs of users for database management.

3. Functional Implementation Method

This chapter will guide readers step by step to build a complete Python-based data management software, which can effectively manage Access databases, and realize the functional requirements analysis of researchers for software functions. At the same time, it also deepens readers' understanding of the Python language through practice, and realizes more flexible application in reality.

3.1. Framework Analysis

According to the functional requirement analysis, the program mainly consists of two major functional categories: database export and database import. To make the code clearer, researchers divide the code into two classes, ExportApp and ImportApp, respectively. Encapsulating different functional implementation codes in different classes can help improve the overall structure of the program and enhance its capabilities in readability, maintainability, and extensibility.

```
class ExportApp:


class ImportApp:



if __name__ == "__main__":
    root = Tk()
    root.title("Database Management")
    master = Notebook(root)
    export_tab = Frame(master)
    import_tab = Frame(master)
    master.add(export_tab, text="Export")
    master.add(import_tab, text="Import")
    master.pack(expand=1, fill="both")

    export_app = ExportApp(export_tab)
    import_app = ImportApp(import_tab)
    root.mainloop()
```

Figure 1. Software Framework Code

Here, by defining ExportApp and ImportApp as classes, they become two classes, and then using the Tkinter library in the main function of the program, a GUI interface named "Database Management" is established. At the same time, two tabs are set in the interface, one for export and one for import, and the ExportApp and ImportApp classes previously defined are respectively connected to these two tabs. This achieves the function of switching between functional page tabs in the GUI interface. As shown in the figure below.

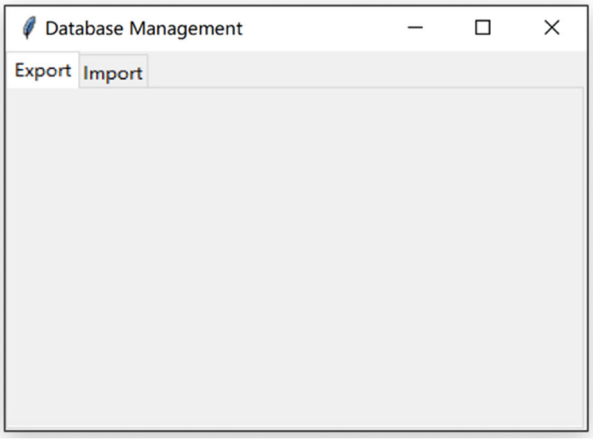

Figure 2. Framework GUI Display

3.2. Export Function Implementation:

After building the overall framework of the program, researchers need to design the specific function implementation. Let's start with designing the code in the ExportApp class.

First, we analyze what operations the user needs to perform to export data from the database. First, the user needs to specify the path of the database to be exported, then select which table to export from the database, and finally select where to store the exported Excel file.

Combining the user's execution steps, we analyze how to implement the function. The first step is to design a function that allows users to specify the path of the database to be exported. The second step is to design a function to connect to the selected database and display the tables in the database for user selection. The third step is to design a function to provide users with the option to select the export file storage

location. The fourth step is to export the specified table from the database and store it at the specified path.

Combining the above requirements, we design functions for each step to implement the functionality.

```
class ExportApp:
    def __init__(self, tab):
        self.export_db_path = ""
        self.export_file_path = ""
        self.create_widgets(tab)

    def create_widgets(self, tab):
        self.export_db_label_text = "Select Database Path:"
        self.export_file_label_text = "Select Export Path:"
        self.export_table_label_text = "Select Table to Export:"

        self.export_db_label = Label(tab, text=self.export_db_label_text)
        self.export_db_label.grid(row=0, column=0, padx=10, pady=5, sticky="w")

        self.export_db_path_entry = Entry(tab, width=50)
        self.export_db_path_entry.grid(row=1, column=0, padx=10, pady=5, sticky="w")

                            self.export_db_button      =      Button(tab,      text="Browse",
command=self.select_export_database)
        self.export_db_button.grid(row=1, column=1, padx=10, pady=5, sticky="w")

        self.export_file_label = Label(tab, text=self.export_file_label_text)
        self.export_file_label.grid(row=2, column=0, padx=10, pady=5, sticky="w")

        self.export_file_path_entry = Entry(tab, width=50)
        self.export_file_path_entry.grid(row=3, column=0, padx=10, pady=5, sticky="w")

                           self.export_file_button      =      Button(tab,      text="Browse",
command=self.select_export_file)
        self.export_file_button.grid(row=3, column=1, padx=10, pady=5, sticky="w")

        self.export_table_label = Label(tab, text=self.export_table_label_text)
        self.export_table_label.grid(row=4, column=0, padx=10, pady=5, sticky="w")

        self.export_selected_table = StringVar(tab)
        self.export_selected_table.set("Select")
                                    self.export_table_dropdown        =        Combobox(tab,
textvariable=self.export_selected_table, width=47)
            self.export_table_dropdown.grid(row=5,  column=0,  columnspan=2,  padx=(10,  5),
pady=5, sticky="w")

        self.export_button = Button(tab, text="Export", command=self.export_to_excel)
        self.export_button.grid(row=5, column=1, padx=10, pady=5, sticky="w")

        self.export_status_label = Text(tab, height=10, width=50)
           self.export_status_label.grid(row=6,  column=0,  columnspan=2,  padx=10,  pady=5,
sticky="w")

    def select_export_database(self):
            self.export_db_path  =  filedialog.askopenfilename(filetypes=[("Access  Database
Files", "*.mdb;*.accdb")])
        self.export_db_path_entry.delete(0, "end")
        self.export_db_path_entry.insert(0, self.export_db_path)
        if self.export_db_path:
            self.connect_to_export_database()

    def connect_to_export_database(self):
                        conn_str     =     f"DRIVER={{Microsoft     Access     Driver     (*.mdb,
*.accdb)}};DBQ={self.export_db_path};"
        conn = pyodbc.connect(conn_str)
        cursor = conn.cursor()
                         self.export_tables     =     [table.table_name     for     table     in
cursor.tables(tableType='TABLE')]
        self.export_table_dropdown["values"] = self.export_tables
        conn.close()

    def select_export_file(self):
        self.export_file_path = filedialog.askdirectory()
        self.export_file_path_entry.delete(0, "end")
        self.export_file_path_entry.insert(0, self.export_file_path)
```

```
    def export_to_excel(self):
        if not self.export_db_path:
            self.export_status_label.delete(1.0, "end")
            self.export_status_label.insert("end", "Please select database path first")
            return
        if not self.export_file_path:
            self.export_status_label.delete(1.0, "end")
            self.export_status_label.insert("end", "Please select export file path first")
            return
          if not self.export_selected_table.get() or self.export_selected_table.get() ==
"Select":
            self.export_status_label.delete(1.0, "end")
            self.export_status_label.insert("end", "Please select table to export")
            return

        table_name = self.export_selected_table.get()
                    conn_str    =    f"DRIVER={{Microsoft    Access    Driver    (*.mdb,
*.accdb)}};DBQ={self.export_db_path};"
        conn = pyodbc.connect(conn_str)
        timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
        excel_path = os.path.join(self.export_file_path, f"{table_name}_{timestamp}.xlsx")
        query = f"SELECT * FROM {table_name}"
        df = pd.read_sql(query, conn)
        df.to_excel(excel_path, index=False)
        timestamp = datetime.now().strftime("%H:%M:%S")
        self.export_status_label.delete(1.0, "end")
            self.export_status_label.insert("end",  f"{timestamp}:  Table  '{table_name}'
successfully exported to '{self.export_file_path}/{table_name}_{timestamp}.xlsx'")
        conn.close()
```

Figure 3. Export Function Implementation Code

As shown in the code, we first initialize the ExportApp class and design a simple GUI display interface according to the functional requirements.

Next, we create the select_export_database function, which allows the user to choose the database path. Since this software is primarily used to access Access databases, we limit the file names in the function to only import *.mdb or *.accdb Access databases.

Then, we create the connect_to_export_database function, which uses the API in the pyodbc library to establish a connection with the Access database and retrieve the names of all tables in the database. These table names are displayed in a dropdown list for the user to select. This function is also called in the select_export_database function, so when the user selects an Access database through select_export_database, the software automatically executes the connect_to_export_database function to display all tables in the Access database in the dropdown list, awaiting user selection.

Next, we create the select_export_file function, which allows the user to choose the export file storage path.

Finally, we create the export_to_excel function, which, after the user executes all the above functions, uses SQL query language corresponding to Access to retrieve data from the user-selected table. The retrieved data is then saved in an Excel file at the specified path using the API in the pandas library.

To improve interaction between the program and the user, we add a detection mechanism in the export_to_excel function to check whether the previous steps have been completed. If not, the user is prompted to complete the necessary steps first. Additionally, for the exported Excel file, we name it consistently with the table names in the database and add a timestamp of the current time. This allows the exported file to serve as a valid backup file for the database table, enhancing the convenience of database backup operations. Finally, when the program successfully completes the export, the program status display window notifies the user that the data has been successfully exported.

With this, all the functional requirements of the ExportApp class are completed, and the specified tables in the database are successfully exported to Excel files, saved in the user-specified path. The advantages of using the Python language are also demonstrated, as many important functionalities can be achieved using APIs in third-party libraries without requiring users to analyze how to implement them, truly achieving the convenience of "plug and play".

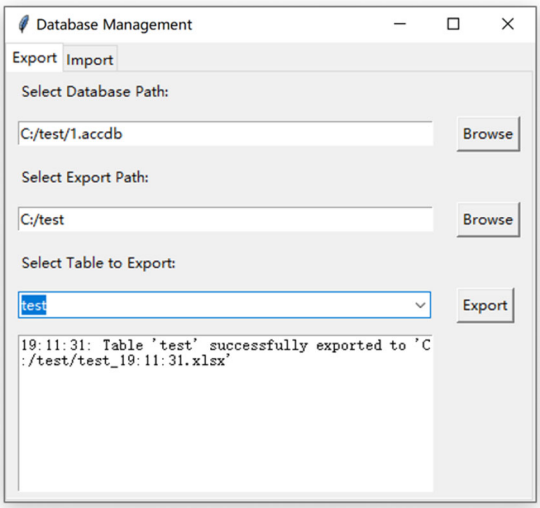

Figure 4. Export GUI Display

3.3. Implementation of Import Functionality:

After implementing the functionality of the ExportApp class, the researchers proceeded to build the code for the ImportApp class.

With the foundation laid by the previous analysis of export functionality requirements, we now only need to address the user's needs in reverse. Firstly, users need to specify the Excel file and its path for import. Then, they select the database path for import, followed by choosing the table within the database for import, and finally executing the import operation.

Considering the user's execution steps, we analyze how to implement each functionality. The first step involves designing a function for the user to specify the import file and its path. The second step is to provide the user with the option to choose the database import path. The third step requires designing a function to establish a connection to the selected database and display the tables within the database for user selection. The fourth step involves importing the data from the Excel file into the specified table within the database.

Based on these requirements, we proceed to design individual functions for each step to implement the desired functionality.

```
  class ImportApp:
      def __init__(self, tab):
          self.import_file_path = ""
          self.import_db_path = ""
          self.create_widgets(tab)

      def create_widgets(self, tab):
          self.import_file_label_text = "Select Import File:"
          self.import_db_label_text = "Select Import Database:"
          self.import_table_label_text = "Select Table to Import:"

          self.import_file_label = Label(tab, text=self.import_file_label_text)
          self.import_file_label.grid(row=0, column=0, padx=10, pady=5, sticky="w")

          self.import_file_path_entry = Entry(tab, width=50)
          self.import_file_path_entry.grid(row=1, column=0, padx=10, pady=5, sticky="w")

                              self.import_file_button      =      Button(tab,      text="Browse",
command=self.select_import_file)
          self.import_file_button.grid(row=1, column=1, padx=10, pady=5, sticky="w")

          self.import_db_label = Label(tab, text=self.import_db_label_text)
          self.import_db_label.grid(row=2, column=0, padx=10, pady=5, sticky="w")

          self.import_db_path_entry = Entry(tab, width=50)
          self.import_db_path_entry.grid(row=3, column=0, padx=10, pady=5, sticky="w")

                               self.import_db_button      =      Button(tab,      text="Browse",
command=self.select_import_database)
          self.import_db_button.grid(row=3, column=1, padx=10, pady=5, sticky="w")

          self.import_table_label = Label(tab, text=self.import_table_label_text)
          self.import_table_label.grid(row=4, column=0, padx=10, pady=5, sticky="w")

          self.import_selected_table = StringVar(tab)
          self.import_selected_table.set("Select")
                                       self.import_table_dropdown        =        Combobox(tab,
textvariable=self.import_selected_table, width=47)
              self.import_table_dropdown.grid(row=5,  column=0,  columnspan=2,  padx=(10,  5),
pady=5, sticky="w")

          self.import_button = Button(tab, text="Import", command=self.import_from_excel)
          self.import_button.grid(row=5, column=1, padx=10, pady=5, sticky="w")

          self.import_status_label = Text(tab, height=10, width=50)
            self.import_status_label.grid(row=6,  column=0,  columnspan=2,  padx=10,  pady=5,
sticky="w")

      def select_import_file(self):
              self.import_file_path  =  filedialog.askopenfilename(filetypes=[("Excel  Files",
"*.xlsx;*.xls")])
          self.import_file_path_entry.delete(0, "end")
          self.import_file_path_entry.insert(0, self.import_file_path)

      def select_import_database(self):
              self.import_db_path  =  filedialog.askopenfilename(filetypes=[("Access  Database
Files", "*.mdb;*.accdb")])
          self.import_db_path_entry.delete(0, "end")
          self.import_db_path_entry.insert(0, self.import_db_path)
          if self.import_db_path:
              self.connect_to_import_database()
```

```
    def connect_to_import_database(self):
                    conn_str    =    f"DRIVER={{Microsoft    Access    Driver    (*.mdb,
*.accdb)}};DBQ={self.import_db_path};"
        conn = pyodbc.connect(conn_str)
        cursor = conn.cursor()
                    self.import_tables    =    [table.table_name    for    table    in
cursor.tables(tableType='TABLE')]
        self.import_table_dropdown["values"] = self.import_tables
        conn.close()

    def import_from_excel(self):
        if not self.import_file_path:
            self.import_status_label.delete(1.0, "end")
            self.import_status_label.insert("end", "Please select import file path first")
            return
        if not self.import_db_path:
            self.import_status_label.delete(1.0, "end")
              self.import_status_label.insert("end", "Please select import database path
first")
            return
          if not self.import_selected_table.get() or self.import_selected_table.get() ==
"Select":
            self.import_status_label.delete(1.0, "end")
            self.import_status_label.insert("end", "Please select table to import")
            return

        table_name = self.import_selected_table.get()
                    conn_str    =    f"DRIVER={{Microsoft    Access    Driver    (*.mdb,
*.accdb)}};DBQ={self.import_db_path};"
        conn = pyodbc.connect(conn_str)
        cursor = conn.cursor()
        cursor.execute(f"DELETE FROM {table_name}")

        df = pd.read_excel(self.import_file_path)
        for index, row in df.iterrows():
            values_str = ', '.join([f"'{value}'" if isinstance(value, str) else str(value)
for value in row])
            sql = f"INSERT INTO {table_name} VALUES ({values_str})"
            cursor.execute(sql)
        cursor.connection.commit()

        timestamp = datetime.now().strftime("%H:%M:%S")
        self.import_status_label.delete(1.0, "end")
                    self.import_status_label.insert("end",    f"{timestamp}:    File
'{self.import_file_path}' successfully imported to table '{table_name}'")
        conn.close()
```

Figure 5. ImportApp Class Function Code

As shown in the code, most of the operations are similar to when writing the ExportApp class code, with the main difference being the use of "import" in function naming for readability. Additionally, the functions created, such as select_import_file, select_import_database, and connect_to_import_database, have functionalities similar to their counterparts in the ExportApp class.

The key distinction lies in the creation of the import_from_excel function. After the user executes the aforementioned functions, this function utilizes the pandas library's API to retrieve data from the specified Excel file. It then generates SQL input language suitable for the Access database, executes the SQL input language, and imports the data into the table selected by the user.

Similarly, to enhance interaction with the user, we've incorporated similar interactive displays in the import_from_excel function, including a detection mechanism and prompts upon completion of the import process.

With this, all the functionality requirements of the ImportApp class have been fulfilled, and the data from the specified Excel file has been successfully imported into the table within the user's specified database.

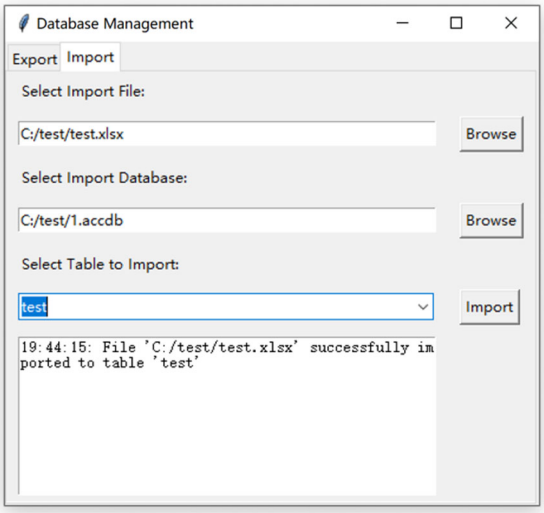

Figure 6. Export GUI Display

3.4. Program Integration

Finally, combine the codes of ExportApp and ImportApp classes with the previous main function, ensuring to import all the necessary libraries. With this, the entire program design is complete. The program can be executed by running the Python file, or the code can be packaged into an *.exe file for execution.

```
import os
import pyodbc
import pandas as pd
from tkinter import filedialog, Tk, Label, Button, StringVar, Entry, Text
from tkinter.ttk import Combobox, Notebook, Frame
from datetime import datetime
```

Figure 7. Dependency on Libraries in the Import Program

4. Conclusion

This project implements a simple yet fully functional database management process using Python programs and Excel software. It greatly reduces the learning curve and technical requirements for users in database management. Additionally, it provides readers with an understanding of Python language programming and its rich array of third-party libraries and tools, making it particularly suitable for small and medium-sized enterprises and individual users with limited budgets.

However, there are areas where the program can be improved and enhanced, such as supporting operations on more database types and optimizing user interface design.

References

[1] Lemahieu, W., vanden Broucke, S., & Baesens, B. (2018). Principles of database management: the practical guide to storing, managing and Analyzing big and small Data. Cambridge University Press.

[2] Ghiani, G., Laporte, G., & Musmanno, R. (2022). Introduction to Logistics Systems Management: With Microsoft Excel and Python Examples. John Wiley & Sons.

[3] Suraya, S., & Sholeh, M. (2022). Designing and implementing a database for thesis data management by using the python Flask Framework. International Journal of Engineering, Science and Information Technology, 2(1), 9-14.

[4] Toman, S. (2021). Web application for Data Import from XLSX into a Relational Database.

[5] Lindstrom, G. (2005). Programming with python. IT professional, 7(5), 10-16.