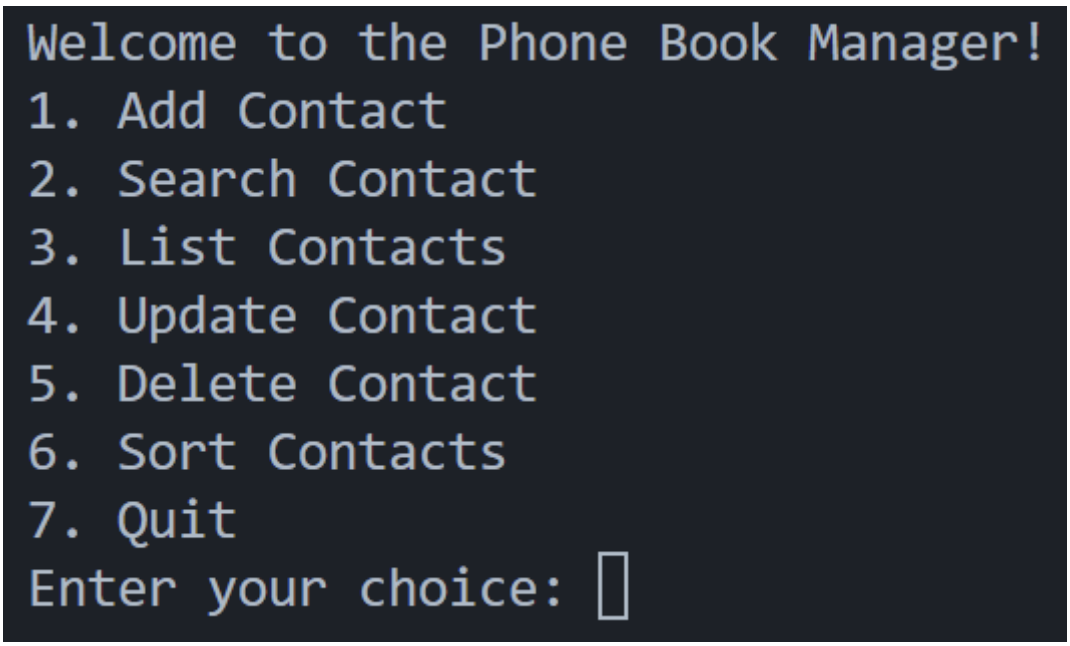
Phone Book Management Report

Shiqi Cui

1. Introduction

The Phone Book Management System is a Python-based command-line application designed to handle various contact management tasks efficiently. It allows users to create, update, delete, and search for contacts, offering both individual and batch operations. The application also integrates advanced functionalities such as sorting, grouping, and logging, ensuring smooth operation while maintaining a clear history of changes. Input validation and error handling are applied throughout the system to ensure data integrity and user-friendliness.

After running the program, user will see the main menu :



2. Data Structures

The core data structures used in the application include:

- **Contact Class:** Represents individual contact entries.

  Each contact is defined by attributes such as first name, last name, phone number, email address (optional), and physical address (optional). Additionally, each contact stores timestamps for creation and last update times, ensuring clear auditing.

- **PhoneBook Class:** Manages the list of contacts and provides CRUD (Create, Read, Update, Delete) functionality.

  Contacts are stored as instances of the Contact class in a list, making it simple to iterate, search, and manipulate data. The PhoneBook class also implements sorting, searching (with wildcard support), and grouping functionalities.

- **PhoneBookCLI Class:** Acts as the user interface, handling input/output operations. It interacts with the PhoneBook class to execute commands based on user input and ensures a smooth user experience.

Throughout and after the developing process, unittests are used to test the effect and help debug. Include:

- **Test_PhoneBook Class**
- **Test_Contact Class**
- **Test_PhoneBookCLI Class**

3. Functionalities Integrated

The system incorporates a wide range of functionalities, ensuring that it meets advanced requirements for contact management:

3.1 Basic CRUD Operations:

3.1.1 Create

- **Add individual contact manually:**

Add new contacts manually. After successfully adding a new contact, the system asks user whether to add another contact.

Code:

```
#phone_book.py
def add_contact(self, first_name, last_name, phone_number, email=None,
address=None):
        '''
            Create a new contact with the provided attributes.
            Raise a ValueError if the contact cannot be created due to missing or
invalid attributes.
        '''
        try:
            new_contact = Contact(first_name, last_name, phone_number, email,
address)
            self.contacts.append(new_contact)
        except ValueError as ve:
            # Log the error and re-raise it to be handled by CLI
            logging.error(f"Failed to add new contact: {ve}")
            raise ve
```

Test:

```
#phone_book.py
def test_add_contact(self):
    contact = Contact("John", "Doe", "1234567890", "john@gmail.com", "123 Main
St")
    self.phonebook.add_contact(contact)  # Add a contact
    self.assertEqual(len(self.phonebook.contacts), 1)
    self.assertEqual(self.phonebook.contacts[0].first_name, "John")
    self.assertEqual(self.phonebook.contacts[0].last_name, "Doe")
    self.assertEqual(self.phonebook.contacts[0].phone_number, "1234567890")
    self.assertEqual(self.phonebook.contacts[0].email, "john@gmail.com")
    self.assertEqual(self.phonebook.contacts[0].address, "123 Main St")
```

Output:

```
1. Add Contact manually
2. Load contacts from CSV file
Enter your choice: 1
First Name (Required): Lily
Last Name (Required): May
Phone Number (Required, Format (###) ###-####): (123) 456-7890
Email (Optional):
Address (Optional):
Contact added successfully.
New contact: Lily May - (123) 456-7890 - None - None (Created: 2024-09-19 10:57:44, Updated: 2024-09-19 10:57:44)
Add another contact? (y/n): N
```

- **Import contacts via CSV file:**

Input the path of CSV file. After importing, the system asks user if user needs to import another CSV file.

The first row of the CSV file should be the header row.

If one row of contact information has invalid data, the program will log the error and continue to the next row.

Code:

```
#phone_book.py
def import_contacts(self, csv_file):
        '''
            Read contacts from a CSV file and add them to the phone book.
            The first row of the CSV file should be the header row.
            Raise a ValueError if the file is not found, cannot be read, or has
invalid CSV format.
        '''
        try:
            with open(csv_file, 'r') as file:
                reader = csv.DictReader(file)
                for row in reader:
                    try:
                        first_name = row['first_name']
                        last_name = row['last_name']
                        phone_number = row['phone_number']
                        email = row.get('email', None)
                        address = row.get('address', None) # Using dict.get() to
handle missing 'address' key
                        # Try to add a new contact. This will raise a ValueError
if data is invalid.
                        self.add_contact(first_name, last_name, phone_number,
email, address)
                    except ValueError as ve:
                        # Log the error and continue to the next row
                        logging.error(f"Error in row{reader.line_num}: {ve}")
                        print(f"Skipping invalid row{reader.line_num}: {ve}")
                        continue

        except FileNotFoundError:
            logging.error(f"File not found: {csv_file}")
            print(f"File not found: {csv_file}")
        except IOError as ioe:
            logging.error(f"Error reading file {csv_file}: {ioe}")
            print(f"Error reading file {csv_file}: {ioe}")
        except csv.Error as cve:
            logging.error(f"CSV parsing error in file {csv_file} : {cve}")
            print(f"Invalid CSV format in file {csv_file}: {cve}")
```

Test: Importing `data.csv` generates the following message.

```
#data.csv
first_name,last_name,phone_number,email,address
John,Smith,(123) 456-7890,john@example.com,456 Edward St
Johnson,Doe,(999) 999-9999,johnson@example.com,80 Prince Rd
Lily,Smith,(343) 343-7890,lilys@example.com,76 Laurier St
Emily,Ying,(555) 455-9999,
Ming,Li,(***),
Li,Chen,(999) 123-4567,lichen123
,Smith,(123) 456-7890,
Lily,,(343) 343-7890,
```

Output:

```
1. Add Contact
2. Search Contact
3. List Contacts
4. Update Contact
5. Delete Contact
6. Sort Contacts
7. Quit
Enter your choice: 1
1. Add contacts manually
2. Load contacts from CSV file
Enter any other key to return to the main menu
Enter your choice: 2
Enter CSV file path: data.csv
Skipping invalid row6: Phone number must be in the format (###) ###-####
Skipping invalid row7: Invalid email address
Skipping invalid row8: First name is required.
Skipping invalid row9: Last name is required.
Contacts imported successfully.
Import another CSV file? (y/n): n
```

3.1.2 Read

Retrieve contacts by different criteria, including first name, last name, or phone number.

- **List all contacts**

Code:

```
#phone_book.py
def list_contacts(self):
        '''
            List all contacts in the phone book.
            Return a list of all contacts.
        '''
        logging.info("Listing all contacts.")
        return self.contacts
```

- **List searched contacts**

Code:

```
#phone_book.py
def sort_contacts(self, key='first_name', reverse=False):
        '''
            Sort contacts by the specified key.
            Return a list of sorted contacts.
            The key parameter specifies the attribute to sort by (e.g.,
'first_name', 'last_name', 'phone_number', 'email', 'address').
            By default, contacts are sorted by first name in ascending order.
            Raise a ValueError if the key is not a valid attribute of the Contact
class.
        '''
        # Sort contacts by the specified key
        self.contacts.sort(key=lambda x: getattr(x, key), reverse=reverse)
        logging.info(f"Contacts sorted by {key} in {ORDER_DICT_LOGGING[reverse]}
order.")
        return self.contacts
```

3.1.3 Update

Modify contact details after performing a search.

Code:

```
#phone_book.py
def update_contact(self, contact, **kwargs):
        '''
            Update a contact with the provided keyword arguments.
            The **kwargs parameter is a special syntax in Python that allows the
method to accept an arbitrary number of keyword arguments.
            These keyword arguments are passed as a dictionary, where the keys
are the argument names and the values are the corresponding values.
            Raise a ValueError if the contact cannot be updated due to missing or
invalid attributes.
        '''
        try:
            contact.update_contact(**kwargs)
        except ValueError as ve:
            # Log the error and re-raise it to be handled by CLI
            logging.error(f"Failed to update contact: {ve}")
```

```
#phone_book_CLI.py
def update_contact(self):
        # Search cantact using keyword
        keyword = input("Search by name or phone number: ")
        results = self.phonebook.search_contact(keyword)
        # If no contacts found, print message and return
        if not results:
            print("No contacts found.")
            return
        # Print the search results
```

```
        for idx, contact in enumerate(results):
            print(f"{idx}: {contact}")
        # Select contact to update
        contact_index = int(input("Enter contact index to update: "))
        # If the contact index is invalid, print message and return
        if contact_index < 0 or contact_index >= len(results):
            print("Invalid contact index.")
            return

        contact = results[contact_index]

        # Update the contact
        first_name = input("First Name (leave blank to keep unchanged): ")
        last_name = input("Last Name (leave blank to keep unchanged): ")
        phone_number = input("Phone Number (leave blank to keep unchanged): ")
        email = input("Email (Optional, leave blank to keep unchanged): ")
        address = input("Address (Optional, leave blank to keep unchanged): ")

        try:
            self.phonebook.update_contact(contact, first_name=first_name,
last_name=last_name, phone_number=phone_number, email=email, address=address)
            print("Contact updated successfully.")
            print(f"Updated contact: {contact}")
        except ValueError as ve:
            print(f"Error: {ve}")
            return
```

3.1.4 Delete

Remove contacts individually or in batch mode.

- **Delete contacts manually after search**

After searching, choose one or more contacts to delete.

Code:

```
#phone_book.py
def delete_contact(self, contact):
        '''
            Delete a contact from the phone book.
        '''
        self.contacts.remove(contact)
        logging.info(f"Contact deleted: {contact}")
```

```
#phone_book_CLI.py
def delete_contact_by_search(self):
        # Search cantact using keyword
        keyword = input("Search by name or phone number: ")
        results = self.phonebook.search_contact(keyword)
        # If no contacts found, print message and return
        if not results:
            print("No contacts found.")
            return
        # Print the search results
```

```
        for idx, contact in enumerate(results):
            print(f"{idx}: {contact}")
        # Select contact to delete
        contact_index = input("Enter contact index to delete (Seperate multiple
choices with ','): ")

        contact_indices = [int(idx.strip()) for idx in contact_index.split(',')]

        for idx in contact_indices:

            # If the contact index is invalid, print message and return
            if idx < 0 or idx >= len(results):
                print(f"Contact index {idx} is invalid.")
                continue

            contact = results[idx]

            # Delete the contact
            self.phonebook.delete_contact(contact)

            # Print the deleted contact
            print(f"Contact deleted:{contact}")
```

- **Delete all contacts**

Code:

```
#phone_book.py
def delete_all_contacts(self):
        '''
            Delete all contacts from the phone book.
        '''
        self.contacts = []
        logging.info("All contacts deleted.")
```

3.2 Advanced Search

3.2.1 Search by names and phone numbers

The search function supports wild-card searches, allowing partial matches in both names and phone numbers.

Code:

```
#phone_book.py
def search_contact(self, keyword):
        '''
            Search for contacts by keyword (name or phone number).
            Return a list of contacts that match the keyword.
            Return an empty list if no contacts are found.
        '''
        results = [contact for contact in self.contacts if keyword.lower() in
contact.first_name.lower() or keyword.lower() in contact.last_name.lower() or
keyword in contact.phone_number]
        logging.info(f"Search results for '{keyword}': {results}")
        return results
```

3.2.2 Search contacts within time frame

The system can filter contacts based on a specific time frame, providing flexible search options.

Code:

```
#phone_book.py
def search_contact_by_updated_time(self, start_time, end_time):
        '''
            Search for contacts updated within a specific time range.
            Raise a ValueError if the start time is greater than the end time.
            Return a list of contacts that were updated within the specified time
range.
            Return an empty list if no contacts are found.
        '''
        results = [contact for contact in self.contacts if start_time <=
contact.updated_at <= end_time]
        logging.info(f"Search results for contacts updated between {start_time}
and {end_time}: {results}")
        return results

    def search_contact_by_created_time(self, start_time, end_time):
        '''
            Search for contacts created within a specific time range.
            Raise a ValueError if the start time is greater than the end time.
            Return a list of contacts that were created within the specified time
range.
            Return an empty list if no contacts are found.
        '''
        results = [contact for contact in self.contacts if start_time <=
contact.created_at <= end_time]
        logging.info(f"Search results for contacts created between {start_time}
and {end_time}: {results}")
        return results
```

3.3 Sorting

The system can sort contacts based on four attributes: first name, last name, created time and updated time. User can choose to sort in ascending or descending order.

Code:

```
#phone_book.py
def sort_contacts(self, key='first_name', reverse=False):
        '''
            Sort contacts by the specified key.
            Return a list of sorted contacts.
            The key parameter specifies the attribute to sort by (e.g.,
'first_name', 'last_name', 'phone_number', 'email', 'address').
            By default, contacts are sorted by first name in ascending order.
            Raise a ValueError if the key is not a valid attribute of the Contact
class.
        '''
        # Sort contacts by the specified key
        self.contacts.sort(key=lambda x: getattr(x, key), reverse=reverse)
        logging.info(f"Contacts sorted by {key} in {ORDER_DICT_LOGGING[reverse]}
order.")
        return self.contacts
```

```
#phone_book_CLI.py
def group_contacts_by_initial_letter(self, key):
        contacts = self.phonebook.group_contacts_by_initial_letter(key)
        for initial, contact_list in contacts.items():
            print(f"{len(contact_list)} contacts with {key} starting with
'{initial}':")
            for contact in contact_list:
                print(contact)

def group_contacts_by_area_code(self):
        contacts = self.phonebook.group_contacts_by_area_code()
        for area_code, contact_list in contacts.items():
            print(f"{len(contact_list)} contacts with phone number area code
'{area_code}':")
            for contact in contact_list:
                print(contact)
```

3.4 Grouping

Contacts can be grouped based on the initial letter of the first name, last name or phone area code.

- **Group by the initial letter of names**

Code:

```
#phone_book.py
def group_contacts_by_initial_letter(self, key):
        '''
            Group contacts by the initial letter of the specified key.
            Return a dictionary where the keys are the initial letters and the
values are lists of contacts.
            The key parameter specifies the attribute to group by (e.g.,
'first_name', 'last_name', 'phone_number', 'email', 'address').
```

```
            Raise a ValueError if the key is not a valid attribute of the Contact
class.
        '''
        # Group contacts by the initial letter of the specified key
        groups = {}
        for contact in self.contacts:
            initial = getattr(contact, key)[0].upper()
            if initial not in groups:
                groups[initial] = []
            groups[initial].append(contact)
        groups = dict(sorted(groups.items())) # Sort the dictionary by key
        logging.info(f"Contacts grouped by initial letter of {key}: {groups}")
        return groups
```

- **Group by phone area codes**

Code:

```
#phone_book.py
def group_contacts_by_area_code(self):
        '''
            Group contacts by the area code of their phone numbers.
            Return a dictionary where the keys are the area codes and the values
are lists of contacts.
            Return an empty dictionary if no contacts have phone numbers.
        '''
        # Group contacts by the area code of their phone numbers
        groups = {}
        for contact in self.contacts:
            area_code = contact.phone_number[1:4]
            if area_code not in groups:
                groups[area_code] = []
            groups[area_code].append(contact)
        groups = dict(sorted(groups.items())) # Sort the dictionary by key
        logging.info(f"Contacts grouped by area code: {groups}")
        return groups
```

3.5 Logging

Every operation, including create, update, and delete actions, is logged with a timestamp for future reference. This logging feature ensures that all actions are transparent and traceable.

The system implements logging using `logging` module.

Example of logging:

```
#phonebook.log
2024-09-18 15:49:29,225 - Failed to add new contact: Invalid email address
2024-09-18 16:22:18,038 - Listing all contacts.
2024-09-18 16:24:48,667 - Search results for contacts created or updated between
2021-01-01 00:00:00 and 2021-01-31 00:00:00: []
2024-09-18 16:27:45,971 - Search results for contacts created or updated between
2021-01-01 00:00:00 and 2021-01-31 00:00:00: []
```

```
2024-09-18 16:27:45,972 - Search results for contacts created or updated between
2024-01-01 00:00:00 and 2024-12-31 00:00:00: [<contact.Contact object at
0x0000019849153DA0>, <contact.Contact object at 0x0000019849153F80>,
<contact.Contact object at 0x0000019849153E00>, <contact.Contact object at
0x00000198491840B0>]
2024-09-18 17:04:00,736 - Search results for contacts created or updated between
2020-01-01 00:00:00 and 2024-12-31 00:00:00: [<contact.Contact object at
0x00000234A4F35E80>, <contact.Contact object at 0x00000234A4F36060>,
<contact.Contact object at 0x00000234A4F35EE0>, <contact.Contact object at
0x00000234A4F36150>]
2024-09-18 18:39:53,240 - Listing all contacts.
2024-09-18 18:39:59,219 - Listing all contacts.
2024-09-18 18:40:33,248 - Listing all contacts.
2024-09-18 18:41:37,143 - Contacts sorted by first_name in ascending order.
2024-09-18 18:44:41,591 - Listing all contacts.
2024-09-18 18:45:41,087 - Listing all contacts.
2024-09-18 18:46:06,409 - Contacts sorted by first_name in ascending order.
```

3.6 Input Validation and Error Handling

The system includes input validation for attributes of contacts.

Error handling is implemented to ensure that invalid data is appropriately handled, particularly during batch imports from CSV files.

3.6.1 Validate formats of phone numbers and email addresses

The system uses regular expression (regex) pattern to ensure inputs are valid.

Both validations rely on the `re.match` function, which attempts to match the regex pattern to the start of the string. If the pattern does not match, the function returns `None`, triggering the `ValueError`. This approach helps maintain data integrity by enforcing strict format requirements for email addresses and phone numbers.

Code:

```python
#contact.py
def __init__(self, first_name, last_name, phone_number, email=None,
address=None):

        # Validation for email format
        if email and not re.match(r"[^@]+@[^@]+\.[^@]+", email):
            raise ValueError("Invalid email address")

        # Validation for phone number format
        if not re.match(r"\(\d{3}\) \d{3}-\d{4}", phone_number):
            raise ValueError("Phone number must be in the format (###) ###-
####")

        # Rest of the code...
```

3.6.1 Validation for required attributes

First name, last name and phone number are required when adding new contacts.

Code:

```
#contact.py
def __init__(self, first_name, last_name, phone_number, email=None,
address=None):
        #Validation for required fields
        if not first_name:
            raise ValueError("First name is required.")
        if not last_name:
            raise ValueError("Last name is required.")
        if not phone_number:
            raise ValueError("Phone number is required.")

        # Rest of the code...
```

3.7 Exporting contacts to a CVS file

Code:

```
#phone_book.py
def export_contacts(self, csv_file):
        '''
            Export contacts to a CSV file.
            The CSV file will contain the following columns: first_name,
last_name, phone_number, email, address.
            Raise a ValueError if the contacts cannot be exported due to an IO
error.
        '''
        try:
            with open(csv_file, 'w', newline='') as file:
                writer = csv.writer(file)
                writer.writerow(['first_name', 'last_name', 'phone_number',
'email', 'address'])
                for contact in self.contacts:
                    writer.writerow([contact.first_name, contact.last_name,
contact.phone_number, contact.email, contact.address])
                logging.info(f"Contacts exported to {csv_file}")
        except IOError as ioe:
            logging.error(f"Error writing to file {csv_file}: {ioe}")
            print(f"Error writing to file {csv_file}: {ioe}")
```

Example Output:

```
#export.csv
first_name,last_name,phone_number,email,address
John,Smith,(123) 456-7890,john@example.com,456 Edward St
Johnson,Doe,(999) 999-9999,johnson@example.com,80 Prince Rd
Lily,Smith,(343) 343-7890,lilys@example.com,76 Laurier St
Emily,Ying,(555) 455-9999,,
```

3.8 Testing and Debugging

During and after the development of the phone book management application, unit tests were employed to verify the correctness of the functionalities, detect bugs early, and ensure the application behaved as expected. The unit tests focused on the core classes and functionalities: `PhoneBook`, `Contact`, and `PhoneBookCLI`, including:

- **Test_PhoneBook Class**
- **Test_Contact Class**
- **Test_PhoneBookCLI Class**

Relevant codes can be found in the repo.