Report:

Program Design

For

SD 01

Coursework - Part B 23/24

By:

Palihawadana A. A. N. Perera

Student ID (IIT):

20232667

Student ID (UoW):

20822596

Executive Summary

The transaction management system detailed in this design report offers a comprehensive solution for efficiently handling financial transactions. Designed with modularity and user-friendliness in mind, the system enables users to add, update, view, and delete transactions with ease. Notable features include support for manual input and file import, robust error handling, and intuitive user interfaces. With a focus on clarity, maintainability, and reliability, this system is poised to streamline transaction management tasks for individuals and businesses alike.

Methodology

The design approach for the transaction management system adopts a bottom-up methodology, starting with the identification of core functionalities and gradually building upon them to create a comprehensive system. The methodology encompasses the following steps:

1. Identifying Core Functionalities: The design process begins by identifying the fundamental tasks the system must perform, such as adding, updating, viewing, and deleting transactions.

2. Modular Function Design: Each core functionality is broken down into smaller, modular functions, each responsible for a specific task. This modular design approach promotes code reusability, maintainability, and scalability.

3. Detailed Function Specification: Each modular function is meticulously specified with clear input-output requirements, descriptions, and pseudo-code snippets. This ensures that the purpose and functionality of each function are well-understood and documented.

4. Testing and Iteration: Throughout the design process, each function is tested rigorously to verify its correctness and robustness. Feedback from testing informs iterative improvements and refinements to the system.

5. Integration and System Testing: Once individual functions are developed and tested, they are integrated to form the complete transaction management system. System-level testing is conducted to validate the system's overall functionality and performance.

6. Documentation and Reporting: Comprehensive documentation, including design reports, user manuals, and technical specifications, is prepared to facilitate understanding, implementation, and maintenance of the system.

By following a bottom-up approach, the design methodology ensures that the transaction management system is built upon a solid foundation of well-defined, modular components. This approach promotes clarity, flexibility, and scalability, resulting in a robust and user-friendly system tailored to meet the needs of diverse users.

Project Specification:

Language: Python - Dictionary-based with JSON serialization.

This program is a Personal Finance Tracker developed using Python, focusing on essential programming principles such as dictionaries, loops, functions, input/output, and input validation. It manages financial transactions using dictionaries, where keys represent the type of expenses and values are lists of transactions themselves. The program utilizes JSON serialization for data storage and retrieval, enhancing its usability for larger datasets.

List of Python programming principles used:
- Decision structures.
- Loops and nested loops.
- Dictionaries and nested dictionaries.
- Dictionary manipulation.
- Functions.
- File handling: JSON serialization and deserialization.
- Data structures.

Core functionality of this program: To perform error-free CRUD operations on financial transactions.

Program Requirements:
1. The program must perform CRUD operations error-free.
2. Utilize JSON files for data persistence and storage.
3. Utilize dictionaries as the primary data structure for managing financial transactions.
4. Manage data using dictionaries and dictionary manipulations.
5. Store transactions using dictionaries and nested dictionaries.

Example structure of transactions in the program:
```python
all_transactions = {
   "groceries": {"type": "expense", "transactions":[
         {'amount': 102.00, 'date': datetime.date(2024, 1,1)},
         {'amount': 102.00, 'date': datetime.date(2023, 2,1)},
         {'amount': 102.00, 'date': datetime.date(2022, 3,1)}
         ]},
   "salary": {'type': 'income', 'transactions': [
         {'amount': 1500.00, 'date': datetime.date(2024, 1, 1)},
         {'amount': 102.00, 'date': datetime.date(2023, 4,1)}
         ]},
   "bonus": {'type': 'income', 'transactions': [
         {'amount': 1500.70, 'date': datetime.date(2024, 1, 1)},
```

```
        {'amount': 100.79, 'date': datetime.date(2023, 4,1)}
        ]}
}
```

Program Features:

1. Add, View, Update, Delete transactions.
2. Show summaries of transactions based on time periods.
3. Perform file operations for bulk reading and writing of transactions.
4. Modular code structure with integration of JSON file handling.
5. Main menu with optional submenus for better user experience.
6. User guide with setup instructions and feature descriptions.
7. Robust testing with a comprehensive test plan and presentation of test findings.

Development Plan:

- ☐ **01 Design Data Structure:**
  - ☐ - Define the structure of financial transactions using dictionaries.
  - ☐ - Determine the key-value pairs for storing transaction details.

- ☐ **02 Modularize Program:**
  - ☐ - Define functions for all operations such as Add, View, Update, Delete transactions.
  - ☐ - Create a Python package to organize the program modules.

- ☐ **03 Design Program:**
  - ☐ - Design the file structure for organizing program files.
  - ☐ - Develop the main program logic for handling user interactions and menu navigation.
  - ☐ - Create a testing plan to ensure program functionality and reliability.

- ☐ **04 Begin Programming:**
  - ☐ - Implement JSON serialization/deserialization for data storage.
  - ☐ - Develop the defined data structure for managing transactions.
  - ☐ - Implement the defined functions for CRUD operations and error handling.
  - ☐ - Create the main program to integrate modules and manage user interactions.
  - ☐ - Integrate error handling and input validation to ensure data integrity.
  - ☐ - (Optional) Develop a graphical user interface (GUI) for enhanced user experience.

- ☐ **05 Testing:**
  - ☐ - Perform integration testing to ensure modules work together seamlessly.
  - ☐ - Review and refine the program code based on testing results.
  - ☐ - Re-test the program to ensure all issues are addressed.
  - ☐ - Review and refine the program again for further improvements.

- ☐ **06 Documentation:**
  - ☐ - Create test papers to document testing procedures and results.
  - ☐ - Develop a user guide to provide instructions on using the program.
  - ☐ - Ensure comprehensive documentation covers setup instructions, feature descriptions, and troubleshooting tips.

01 Data Structure for the Program:

This program will utilize a dictionary of dictionaries to represent all transactions. Each transaction category will be a key in the parent dictionary, and the corresponding value will be a dictionary containing the transaction type and a list of transaction dictionaries.

1.1 The Main Data Structure:

The main data structure for storing transactions is as follows:

- Parent Dictionary (all_transactions):
  - Keys: Transaction categories (strings)
  - Values: Dictionary representing transaction details
    - Keys: "type" (string), "transactions" (list of dictionaries)
    - Values: Transaction type (string), List of transaction dictionaries

Each transaction dictionary will have the following key-value pairs:
- "amount": Amount of the transaction (float)
- "date": Date of the transaction (string in ISO 8601 format) for serialization.
  : Date of the transaction (datetime.date object) for deserialization,

Example transaction structure for serialization:

```
{
   'amount': 102.00,
   'date': '2024-01-01'
}
```

Example transaction structure for deserialization:

```
{
   'amount': 102.00,
   'date': datetime.date(2024, 1, 1)
}
```

Finally, each transaction dictionary will be nested inside the parent dictionary all_transactions, where the key represents the transaction category.

Example Data Structure for all_transactions:

```
all_transactions = {
```

```
    "groceries": {
        "type": "expense",
        "transactions": [
            {'amount': 102.00, 'date': '2024-01-01'},
            {'amount': 102.00, 'date': '2023-02-01'},
            {'amount': 102.00, 'date': '2022-03-01'}
        ]
    },
    "salary": {
        "type": "income",
        "transactions": [
            {'amount': 1500.00, 'date': '2024-01-01'},
            {'amount': 102.00, 'date': '2023-04-01'}
        ]
    },
    "bonus": {
        "type": "income",
        "transactions": [
            {'amount': 1500.70, 'date': '2024-01-01'},
            {'amount': 100.79, 'date': '2023-04-01'}
        ]
    }
}
```

Note: this example shown is the data structure used for JSON serialization, the structure used for deserialization will be the same as the above example, but will use datetime.date objects for the dates.

Serialization: See end_program function in main program module.
Deserialization: See load_transactions function in main program module.

1.2 File Structure.

20232667_CW_Part_B (parent folder)

- Program Documentation (documentation folder)
- Program Functions (folder containing functions module)
    - all_functions.py (module containing all functions)

- bulk transactions.txt (bulk transactions file)
- personal_fin_tracker.py (main python module)
- transactions.json (main transactions save file)
- b_trasactions.json (backup transactions save file)

2.0 Modularising the Program.

2.1 List of all functions used.

Basic Functions:
1. `get_year() -> int`
2. `get_month() -> int`
3. `is_leap(year) -> bool`
4. `get_day(year: int, month: int) -> int`
5. `get_date() -> datetime.date`
6. `get_amount() -> float`
7. `get_type() -> str`
8. `get_transaction() -> tuple[str, dict]`
9. `get_option_choice(options_list: list[int, str]) -> int`
10. `get_sorted_transactions(transactions_to_sort: dict[str: dict]) -> dict[str: dict]`
11. `add_transaction_count(transactions_dict: dict[str: dict]) -> dict`
12. `get_by_date(transactions_dict: dict[str: dict]) -> dict`
13. `get_by_type(transactions_dict: dict[str: dict]) -> dict`

Printing Functions:
14. `print_transactions(transactions_dict: dict[str: dict])`
15. `print_with_count(transactions_list: dict[str: dict])`
16. `print_by_date(transactions_dict: dict[str: dict])`
17. `print_by_year(transactions_dict: dict[str: dict])`
18. `print_by_month(transactions_dict: dict[str: dict])`
19. `print_by_type(transactions_dict: dict[str: dict])`

Main Menu Functions:
20. `main_menu(transactions_dict: dict[str: dict]) -> bool`
21. `back_to_menu() -> bool`
22. `get_transaction_info(transactions_dict: dict[str, dict]) -> tuple[bool, str, int]`
23. `choose_transaction(transactions_dict: dict[str, dict]) -> tuple[bool, str, dict[str: dict]]`
24. `add_transactions(transactions_dict: dict[str, dict]) -> None`
25. `update_transactions(transactions_dict: dict[str, dict]) -> None`
26. `delete_transactions(transactions_dict: dict[str, dict]) -> None`
27. `view_transactions(transactions_dict: dict)`
28. `read_transactions_from_file(file_path: str, transactions_dict: dict[str, dict]) -> None`

These functions cover various aspects of managing and interacting with financial transactions in the program.

2.2 Pseudo Code for Functions.

For the development of the Personal Finance Tracker program, the pseudo code for certain key functions has been provided in the report. These functions were selected based on their significance in managing financial transactions and interacting with the user interface. While many functions in the program were adapted from Coursework Part A and therefore already have corresponding pseudo code, the focus of this section was to highlight the newly implemented or modified functions. Pseudo code was provided for functions such as

- `get_transaction()`,
- `get_sorted_transactions()`,
- `add_transaction_count()`,
- `get_by_date()`,
- `get_by_type()`,
- `print_transactions()`,
- `print_with_count()`,
- `get_transaction_info()`,
- `choose_transaction()`,
- `add_transactions()`,
- `update_transactions()`,
- `delete_transactions()`,
- `view_transactions()`,
- `read_transactions_from_file()`.

These functions were deemed essential for the core functionality of the program and required detailed explanation in the report to demonstrate their implementation and functionality within the overall system.

Function name: get_transaction

Args: None

Description: This function prompts the user to enter details of a transaction, including the category, amount, type, and date. It then constructs a dictionary representing the transaction and returns it as a tuple along with the category.

Pseudo code:

```
Function get_transaction():
    While True:
        Prompt the user to enter the category of the transaction in lowercase and store it in the variable category.
        If the category is empty:
```

```
        Print an error message indicating that the category must not be empty.
    Else:
        Break out of the loop.

Call the function get_amount() and store the returned value in the variable amount.
Call the function get_type() and store the returned value in the variable transaction_type.
Call the function get_date() and store the returned value in the variable date.

Create a dictionary representing the transaction with the category as the key and a nested dictionary containing the type and transactions as the value.
    The transactions key has a list of dictionaries containing the amount and date of the transaction.

Return a tuple containing the category and the transaction dictionary.
```

Function name: get_sorted_transactions

Args: transactions_to_sort (dict)

Description: This function takes a dictionary of transactions as input and sorts the transactions within each category by date. It returns a new dictionary containing the sorted transactions.

Pseudo code:

```
Function get_sorted_transactions(transactions_to_sort: dict) -> dict:
    Initialize an empty dictionary sorted_transactions to store the sorted transactions.

    For each category and details in transactions_to_sort.items():
        Access the category and details of each transaction.

        Sort the transactions within each category by date using the sorted function and a lambda function as the key.
        The lambda function extracts the date from each transaction dictionary.

        Assign the sorted transactions to the sorted_transactions dictionary with the same category key.
        Preserve the type information from the original transactions.

    Return the sorted_transactions dictionary containing the sorted transactions.
```

Function name: add_transaction_count

Args: transactions_dict (dict)

Description: This function adds transaction numbers to the transactions in each category. It takes a dictionary containing transaction details as input and returns a new dictionary with transaction numbers added to each transaction.

Pseudo code:
```
Function add_transaction_count(transactions_dict: dict) -> dict:
    Call the get_sorted_transactions function with transactions_dict as input and assign the result to sorted_transactions.

    Initialize an empty dictionary transactions_with_count to store transactions with counts.

    For each category in sorted_transactions:
        Initialize a count variable to 0 to keep track of transaction numbers.

        Initialize an empty list to store transactions with transaction numbers.

        For each transaction in the sorted_transactions[category]['transactions']:
            Increment the count by 1 for each transaction.

            Append a new dictionary to the transactions_with_count[category]['transactions'] list.
            This dictionary contains the transaction number, amount, and date of each transaction.

    Return the transactions_with_count dictionary containing transactions with transaction numbers.
```

Function name: get_by_date

Args: transactions_dict (dict)

Description: This function returns a dictionary containing transactions of a particular date. It takes a dictionary containing transaction details as input and returns a filtered dictionary with transactions that match the specified date.

Pseudo code:
```
Function get_by_date(transactions_dict: dict) -> dict:
    Get the date from the user using the get_date function and assign it to the variable date.
```

```
    For each category in transactions_dict:
        Initialize an empty list filtered_transactions to store filtered transactions.

        For each transaction in transactions_dict[category]['transactions']:
            If the date of the transaction matches the specified date:
                Append the transaction to the filtered_transactions list.

        Update the transactions_dict[category]['transactions'] with the filtered_transactions list.

    Return the updated transactions_dict containing transactions filtered by date.
```

Function name: get_by_type

Args: transactions_dict (dict)

Description: This function filters transactions by transaction type and returns a dictionary containing transactions of the specified type. It takes a dictionary containing transaction details as input and returns a filtered dictionary with transactions that match the specified transaction type.

Pseudo code:

```
Function get_by_type(transactions_dict: dict) -> dict:
    Get the transaction type from the user using the get_type function and assign it to the variable transaction_type.

    Initialize an empty dictionary transactions_by_type to store transactions filtered by type.

    For each category in transactions_dict:
        If the type of transactions_dict[category] matches the specified transaction_type:
            Add transactions_dict[category] to transactions_by_type with a new key.

    Return the transactions_by_type dictionary containing transactions filtered by type.
```

Function name: print_transactions

Args: transactions_dict (dict)

Description: This function prints transactions stored in a dictionary in a formatted table. It takes a dictionary containing transaction details as input and prints the transactions along with their total amount for each category.

Pseudo code:

```
Function print_transactions(transactions_dict: dict):
    Call the get_sorted_transactions function to sort the transactions by date.

    If the length of transactions_dict is 0:
        Print "No transactions to display."
    Else:
        For each category in transactions_dict:
            Initialize an empty list transactions_to_print to store transactions for printing.
            Initialize a running total variable total to 0.

            Print the category and type of transactions_dict[category].

            For each item in transactions_dict[category]['transactions']:
                Append [item['amount'], item['date']] to transactions_to_print.
                Add item['amount'] to total.

                 Print transactions_to_print using tabulate with headers=['Amount (LKR)', 'Date'] and formatted as a table.

            If transactions_dict[category]['type'] is 'expense':
                Print the total LKR spent on the category.
            Else:
                Print the total LKR received from the category.
```

Function name: print_with_count

Args: transactions_list (dict)

Description: This function prints transactions with transaction numbers stored in a dictionary in a formatted table. It takes a dictionary containing transaction details as input and prints the transactions along with their respective transaction numbers for each category.

Pseudo code:

```
Function print_with_count(transactions_list: dict):
    Call the add_transaction_count function to add transaction numbers to the transactions in the input dictionary.

    If the length of transactions_list is 0:
        Print "No transactions to display."
    Else:
```

For each element in transactions_list:
Initialize an empty list printable_transactions to store transactions for printing.

Print the element and type of transactions_list[element].

For each transaction in transactions_list[element]['transactions']:
Append [transaction['transaction num'], transaction['amount'], transaction['date']] to printable_transactions.

Print printable_transactions using tabulate with headers=['No.', 'Amount (LKR)', 'Date'] and formatted as a table.
```

**Function name: get_transaction_info**

**Args:** transactions_dict (dict)

**Description:** This function prompts the user to enter transaction information and validates the input. It returns a tuple containing a boolean indicating whether the transaction was found, the transaction category, and the transaction number.

**Pseudo code:**
```
Function get_transaction_info(transactions_dict: dict) -> tuple[bool, str, int]:
While True:
Prompt the user to enter a transaction category.
If the user enters 'none':
Return False, transaction_category, 0 # No transaction category found, return False for transaction_found.
Else if the entered category is not in transactions_dict:
Print an error message.
Else:
Break the loop.

While True:
Prompt the user to enter a transaction number.
If the user enters 'none':
Return False, transaction_category, 0 # No transaction found, return False for transaction_found.
Try to convert the input to an integer.
If successful:
If the entered number is in the valid range:
Return True, transaction_category, transaction_number # Transaction found, return True for transaction_found.
```
```

```
        Else:
            Print an error message indicating that the transaction number is out of range.
    Else:
        Print an error message indicating that the input must be an integer.
```

Function name: choose_transaction

Args: transactions_dict (dict)

Description: This function presents options to the user to find and select a transaction based on various criteria such as all transactions, date, or type.

Pseudo code:

```
Function choose_transaction(transactions_dict: dict) -> tuple[bool, str, dict[str:dict]]:
    Define an options_list with different search criteria.
    Get the option choice from the user using the get_option_choice function.

    If the option choice is to find a transaction using all transactions:
        Print instructions to choose a transaction.
        Add transaction numbers to each transaction in transactions_dict.
        Print transactions with counts.
        Get transaction information from the user using the get_transaction_info function.
        If the transaction is found:
            Create a dictionary containing the chosen transaction.
        Else:
            Set chosen_transaction to None and print an error message.

    Else if the option choice is to find a transaction using date:
        Get transactions filtered by date.
        Print instructions to choose a transaction.
        Add transaction numbers to each transaction in the filtered transactions.
        Print transactions with counts.
        Get transaction information from the user using the get_transaction_info function.
        If the transaction is found:
            Create a dictionary containing the chosen transaction.
        Else:
            Set chosen_transaction to None and print an error message.

    Else if the option choice is to find a transaction using type:
        Get transactions filtered by type.
        Print instructions to choose a transaction.
        Add transaction numbers to each transaction in the filtered transactions.
```

Print transactions with counts.
Get transaction information from the user using the get_transaction_info function.
If the transaction is found:
Create a dictionary containing the chosen transaction.
Else:
Set chosen_transaction to None and print an error message.

Else:
Set transaction_found to False, transaction_category to None, and chosen_transaction to None.

If chosen_transaction is not None:
Remove the transaction number from the chosen transaction dictionary.
Print the chosen transaction using the print_transactions function.

Return transaction_found, transaction_category, and chosen_transaction.
```

**Function name: add_transactions**

**Args:** transactions_dict (dict)

**Description:** This function allows the user to add a transaction to the existing transactions.

**Pseudo code:**
```
Function add_transactions(transactions_dict: dict) -> None:
Define a constant SUCCESS_MESSAGE for successful transaction addition.
Print instructions to enter transaction information.

Loop indefinitely:
Get transaction details from the user using the get_transaction function.

Check if the category already exists in transactions_dict:
If yes:
Check if the type of the new transaction matches the existing type in transactions_dict:
If not, print an error message and continue to the next iteration.

Check if the new transaction already exists in the category:
If yes, print an error message.
If no, add the new transaction to the existing category and print a success message.
Else:
```
```

Add a new category and transaction to transactions_dict and print a success message.

Check if the user wants to add another transaction:
If yes, continue the loop.
If no, break out of the loop.

End of function.
```

**Function name: update_transactions**

**Args:** transactions_dict (dict)

**Description**: This function allows the user to update an existing transaction.

**Pseudo code:**
```
Function update_transactions(transactions_dict: dict) -> None:
  If the length of transactions_dict is not 0: # Ensure there are transactions to update
    Loop until return_to_menu is True:
      Create an options_list with different update options.
      Print instructions to find a transaction to update.
      Call the choose_transaction function to find the transaction to update and get its details.

      If transaction_found is True: # Transaction found, proceed with update options
        Print instructions for choosing an update option.

        Get the option_choice from the user using the get_option_choice function.

        If option_choice is 1: # Update amount of the transaction
          Get the new_amount from the user.
          Iterate over transactions in the chosen category:
            If the transaction matches the chosen transaction, update its amount with the new_amount.
          Print a success message.

        If option_choice is 2: # Update date of the transaction
          Get the new_date from the user.
          Iterate over transactions in the chosen category:
            If the transaction matches the chosen transaction, update its date with the new_date.
          Print a success message.
```
```

If option_choice is 3: # Update type of the transaction
Get the new_type from the user.
Print a warning message about the impact on all transactions in the category.
Get user confirmation to proceed.
If user confirms:
Update the type of transactions in the category with the new_type.
Print a success message.
If user cancels:
Print a termination message and break out of the loop.

If option_choice is 4: # Update category of the transaction
Print a warning message about the impact on all transactions in the category.
Get user confirmation to proceed.
If user confirms:
Get the new_category from the user.
If new_category does not exist in transactions_dict:
Rename the category to new_category.
Else:
Add transactions to the existing category.
Print a success message.
If user cancels:
Print a termination message and break out of the loop.

Update return_to_menu based on the return value of back_to_menu function.

Else: # Transaction not found
Print a message indicating unable to find a transaction to update and set return_to_menu to True.

Else: # No transactions to update
Print a message indicating unable to find transactions to update.

Function name: delete_transactions

Args: transactions_dict (dict)

Description: This function allows the user to delete an existing transaction.

Pseudo code:
```
Function delete_transactions(transactions_dict: dict) -> None:
If the length of transactions_dict is not 0: # Ensure there are transactions to delete
Loop until return_to_menu is True:
```

```
        Print instructions to find a transaction to delete.
        Call the choose_transaction function to find the transaction to delete and get its details.

        If transaction_found is True: # Transaction found, proceed with deletion
            Remove the chosen transaction from transactions_dict.
            Print a success message.
            Update return_to_menu based on the return value of back_to_menu function.

        Else: # Transaction not found
            Print a message indicating no transactions found and set return_to_menu to True.

    Else: # No transactions to delete
        Print a message indicating no past transactions to delete and return to the main menu.
```

Function name: view_transactions

Args: transactions_dict (dict)

Description: This function allows the user to view different summaries of the saved transactions.

Pseudo code:

```
Function view_transactions(transactions_dict: dict) -> None:
    Initialize return_to_menu to False.

    If the length of transactions_dict is not 0: # Ensure there are transactions to view
        Initialize options_list containing different ways to view transactions.
        Print instructions to choose how to view transaction summaries.
        Get the option choice from the user using the get_option_choice function.

        While return_to_menu is False:
            Depending on the option choice:
                If option_choice is 1:
                    Call the print_transactions function to view all transactions.
                Else if option_choice is 2:
                    Call the print_by_type function to view transactions by type.
                Else if option_choice is 3:
                    Call the print_by_year function to view transactions by year.
                Else if option_choice is 4:
                    Call the print_by_month function to view transactions by month.
                Else:
```

Call the print_by_date function to view transactions by date.

Update return_to_menu based on the return value of the back_to_menu function.

Else: # No transactions to view
Print a message indicating no transactions to view and return to the main menu.

Function name: read_transactions_from_file

Args: file_path (str), transactions_dict (dict)

Description: This function reads transaction data from a text file and adds it to the provided dictionary containing transaction details.

Pseudo code:
```
Function read_transactions_from_file(file_path: str, transactions_dict: dict[str, dict]) -> None:
Display a warning message indicating that transactions will be read from a text file and added to the current database.
Print instructions for the required format of the text file.
Ask the user if they want to proceed.

If the user chooses to proceed:
Initialize an empty list called skipped_lines to store the line numbers of skipped transactions.
Initialize a variable line_count to keep track of the current line number.

Print a message indicating that transactions are being loaded from the file.

Open the specified file in read mode using a context manager.

Iterate over each line in the file:
Increment the line_count for each line read.
Split the line into parts using comma as the delimiter.

If the number of parts is not equal to 6:
Print a message indicating that the transaction does not meet the data structure standard.
Add the line number to skipped_lines.
Continue to the next line.

Extract and format the category, transaction type, amount, year, month, and day from the parts.
```

Attempt to convert amount, year, month, and day to their respective data types.
If any conversion fails, print a corresponding error message and add the line number to skipped_lines.
Continue to the next line.

Check if the transaction date is in the future.
If so, print a message indicating that the transaction date is in the future and add the line number to skipped_lines.
Continue to the next line.

Check if the category already exists in transactions_dict.
If it does, check if the transaction type matches the existing type for that category.
If the types do not match, print a message indicating the mismatch and add the line number to skipped_lines.
Otherwise, append the transaction to the existing category in transactions_dict.
If the category does not exist, create a new category in transactions_dict with the transaction.

Print a message indicating that loading is complete.
If any transactions were skipped, print the line numbers of skipped transactions.
Otherwise, print a success message.

If an error occurs during file reading or processing:
Print an error message indicating the specific error.

Conclusion

In this design report, a transaction management system has been meticulously outlined, crafted with the intention of meeting the requirements of Coursework Part B. The system boasts a well-structured architecture, comprising modular functions tailored to handle various transaction management tasks efficiently.

The design emphasizes clarity and maintainability, with functions named intuitively and documented comprehensively. Each function's purpose and functionality are elucidated through clear descriptions and pseudo-code snippets, facilitating ease of understanding and future modifications.

Key functionalities of the system encompass the ability to add, update, view, and delete transactions seamlessly. Noteworthy features include the capacity to input transactions manually or via text files, as well as robust error-handling mechanisms to bolster reliability.

Collectively, the design of this transaction management system reflects a systematic and pragmatic approach to software development. With its emphasis on modularity, clarity, and resilience, the system stands ready to serve as an effective tool for managing transactions across diverse scenarios.