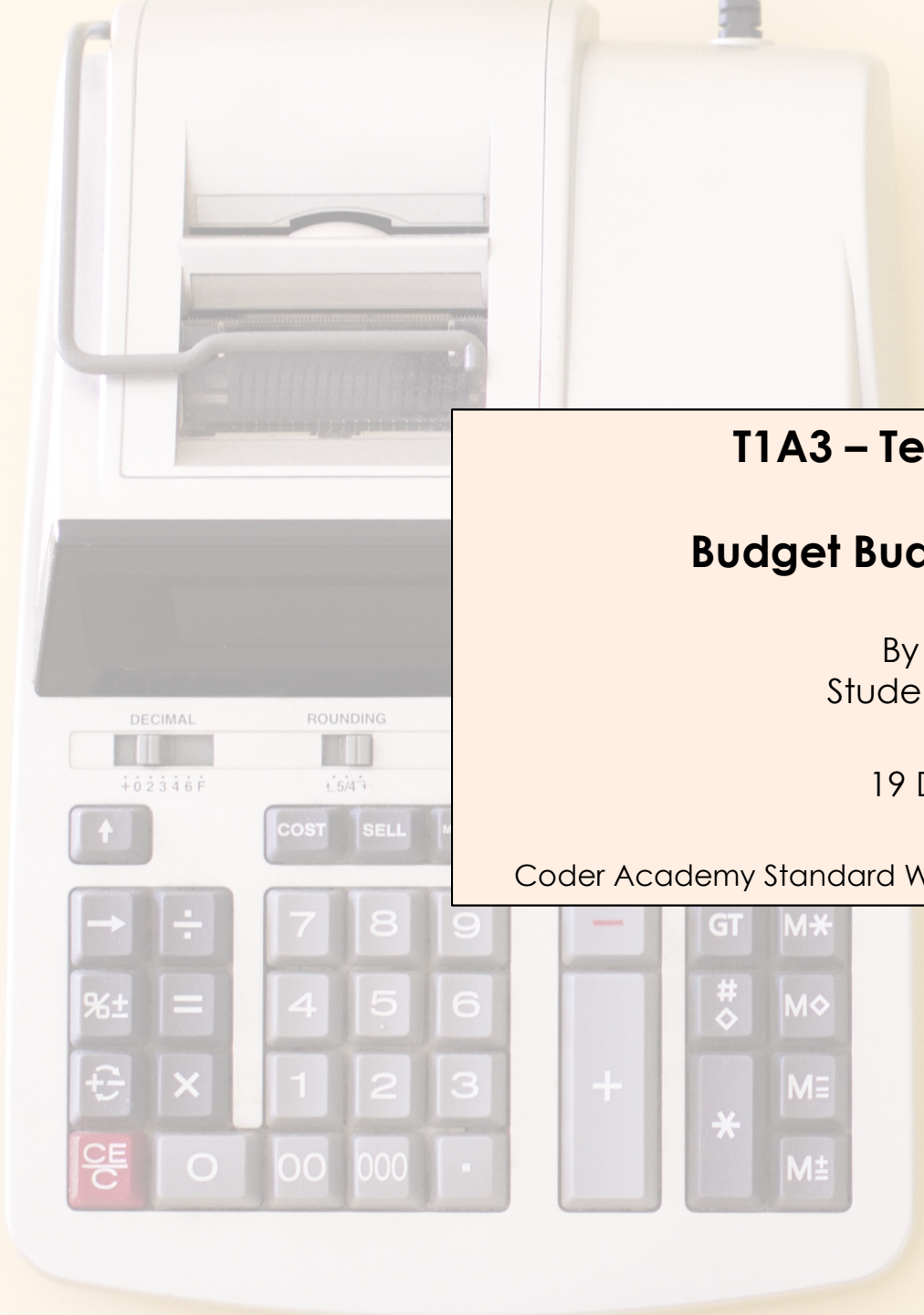


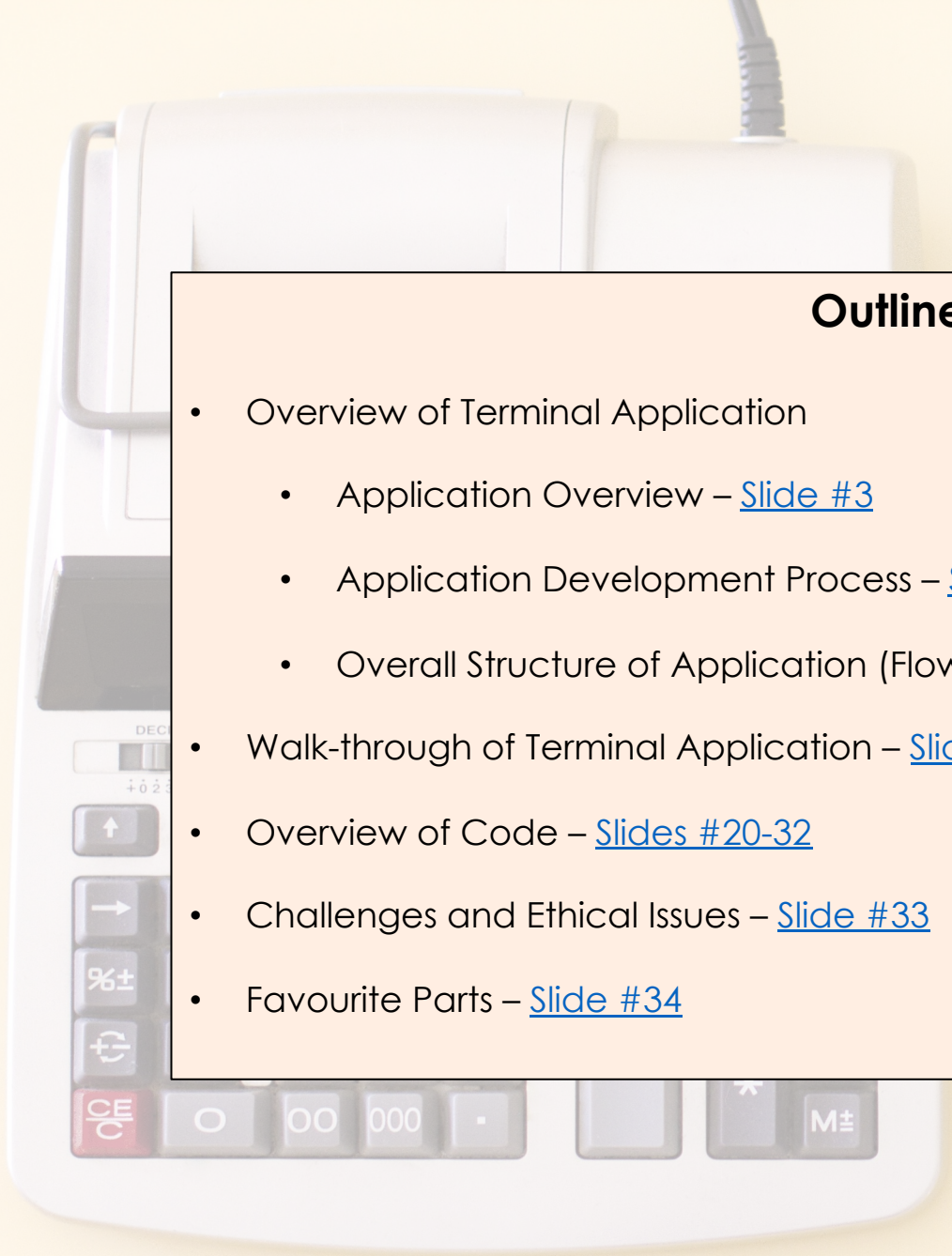
T1A3 – Terminal Application

Budget Buddy: Expense Tracker

By Ishan Acharya
Student Number - 14875

19 December 2023

Coder Academy Standard Web Development Bootcamp October 2023



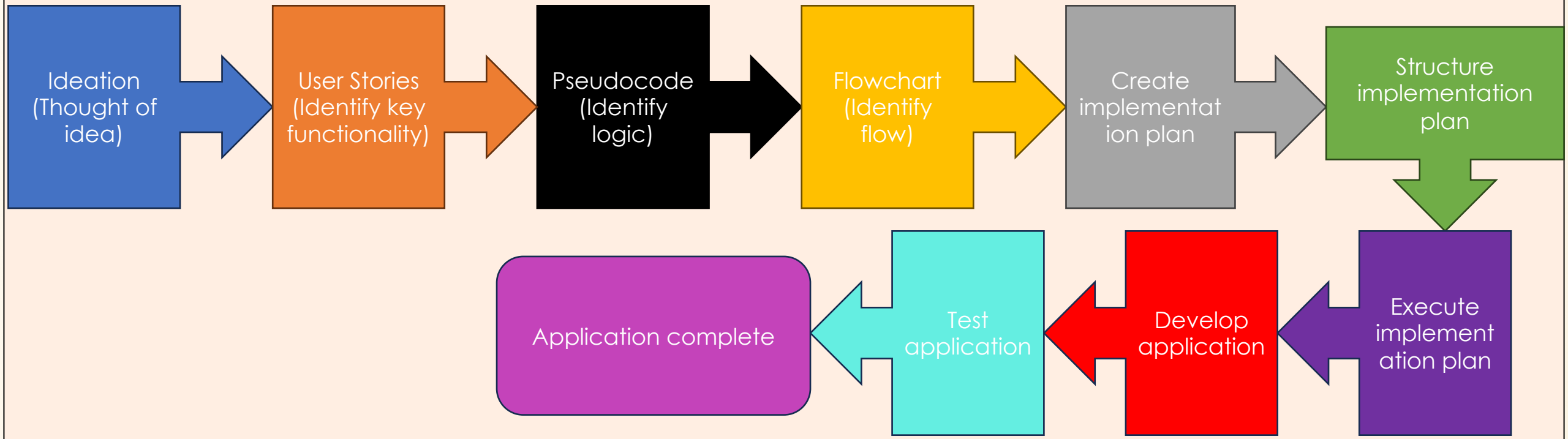
Outline of Presentation

- Overview of Terminal Application
  - Application Overview – Slide #3
  - Application Development Process – Slide #4
  - Overall Structure of Application (Flowchart) – Slides #5-6
- Walk-through of Terminal Application – Slides #7-19
- Overview of Code – Slides #20-32
- Challenges and Ethical Issues – Slide #33
- Favourite Parts – Slide #34

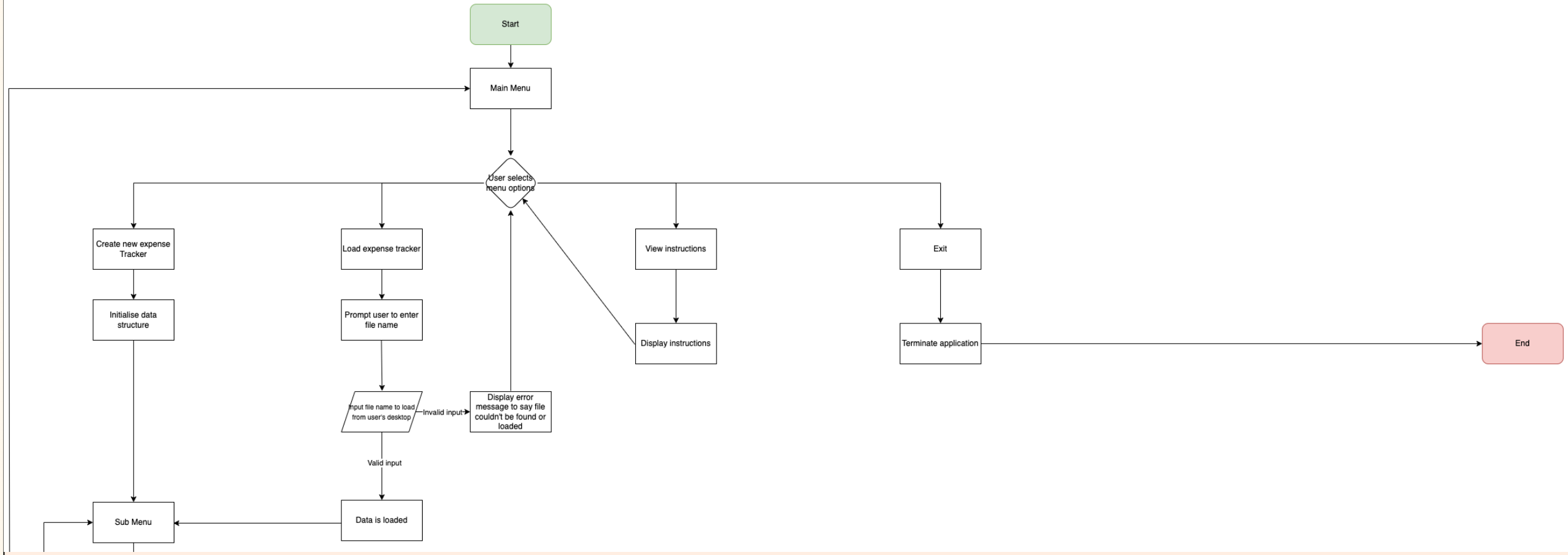
Application Overview

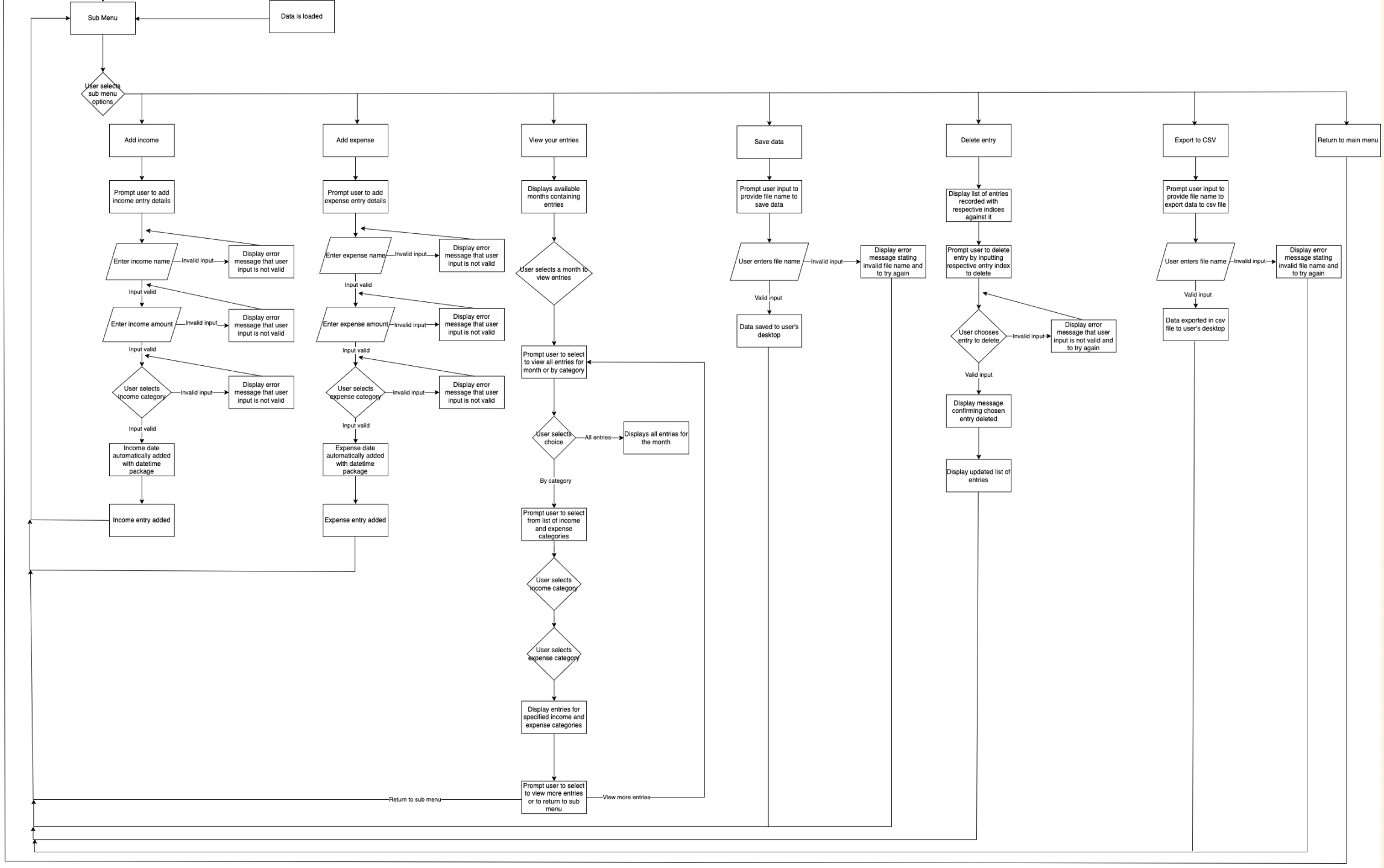
- **What is it?** Budget Buddy is a simple terminal expense tracking tool that exists to help users manage their finances
- **Why did I create it?**
    - Cost of living crisis in Australia, users need to do simple budgeting
    - Free simple tool that has no subscription service unlike majority of apps on other devices
- **Main Features**
    - Allows users to record their income and expenses
    - Allows users to view their income and expenses for the month, and by category
    - Allows users to save and load their application data, and export their data as csv file
- **Benefits of using the application**
    - Users can track their spending habits
    - Use exported data for external data analysis

Application Development Process

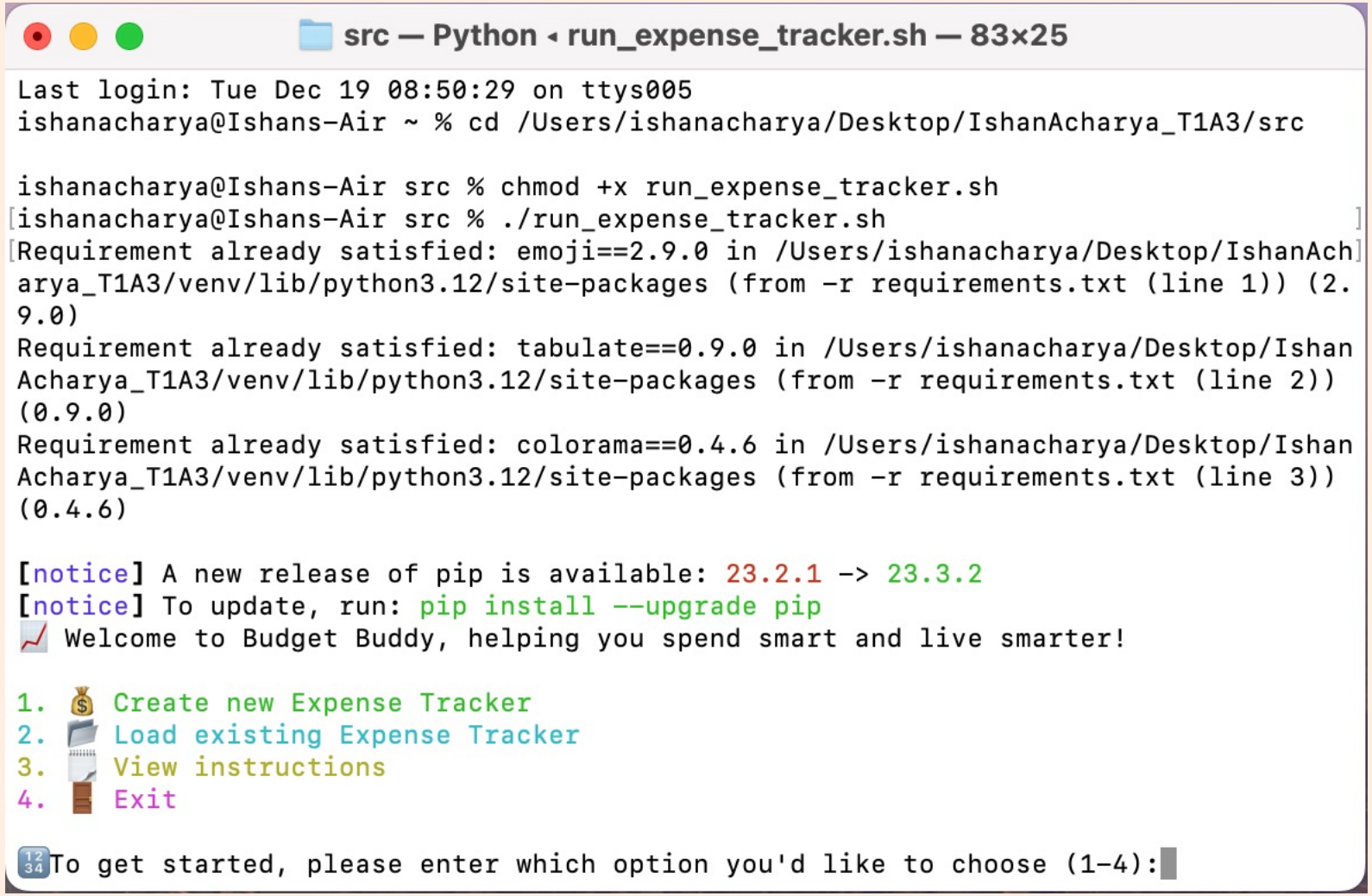


Overall Structure of Application Flowchart




Sub Menu
Data is loaded
User selects sub menu options
Add income
Prompt user to add income entry details
Enter income name
Invalid input
Display error message that user input is not valid
Input valid
Enter income amount
Invalid input
Display error message that user input is not valid
Input valid
User selects income category
Invalid input
Display error message that user input is not valid
Input valid
Income date automatically added with datetime package
Income entry added
Add expense
Prompt user to add expense entry details
Enter expense name
Invalid input
Display error message that user input is not valid
Input valid
Enter expense amount
Invalid input
Display error message that user input is not valid
Input valid
User selects expense category
Invalid input
Display error message that user input is not valid
Input valid
Expense date automatically added with datetime package
Expense entry added
View your entries
Displays available months containing entries
User selects a month to view entries
Prompt user to select to view all entries for month or by category
User selects choice
All entries
Displays all entries for the month
By category
Prompt user to select from list of income and expense categories
User selects income category
User selects expense category
Display entries for specified income and expense categories
Prompt user to select to view more entries or to return to sub menu
Return to sub menu
View more entries
Save data
Prompt user input to provide file name to save data
User enters file name
Invalid input
Display error message stating invalid file name and to try again
Valid input
Data saved to user's desktop
Delete entry
Display list of entries recorded with respective indices against it
Prompt user to delete entry by inputting respective entry index to delete
User chooses entry to delete
Invalid input
Display error message that user input is not valid and to try again
Valid input
Display message confirming chosen entry deleted
Display updated list of entries
Export to CSV
Prompt user input to provide file name to export data to csv file
User enters file name
Invalid input
Display error message stating invalid file name and to try again
Valid input
Data exported in csv file to user's desktop
Return to main menu

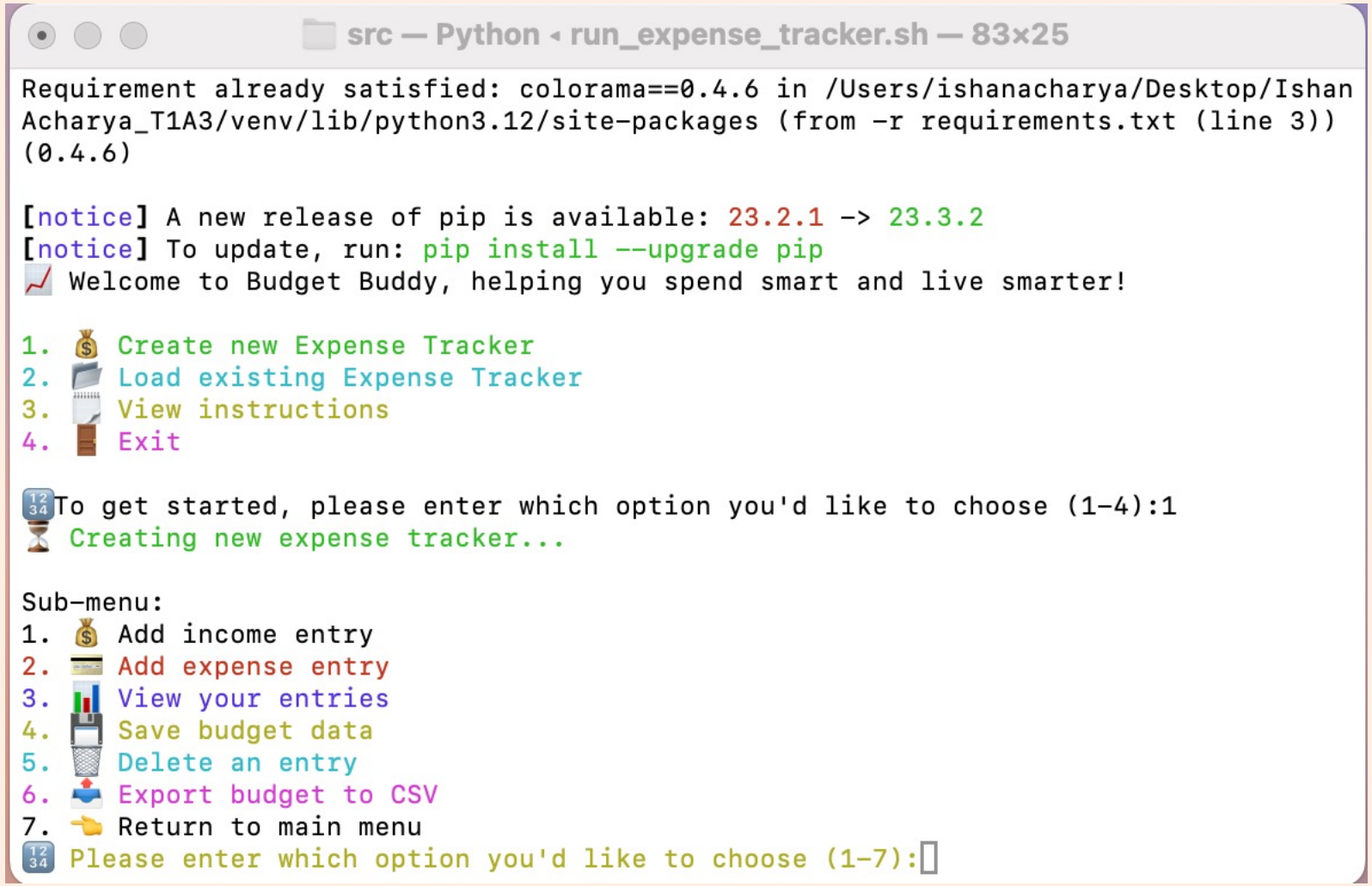
Application Walk-through – Starting Application/Main Menu



Features

- Utilises script to launch application
- Welcome message
- Packages used to give personality to application
- User input to select options for users to navigate around application

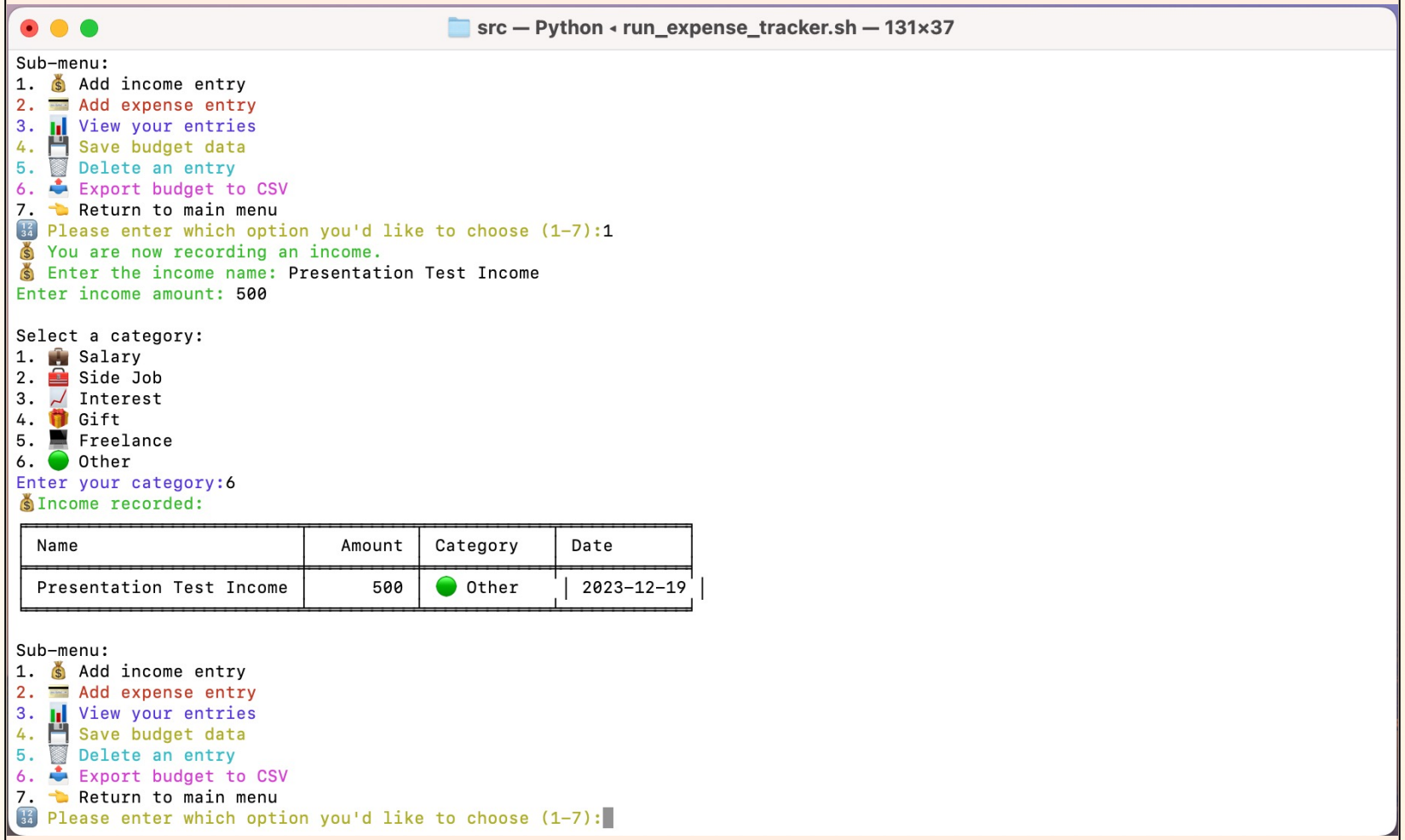
Application Walk-through – Create New Expense Tracker



Features

- User input to select sub menu options for users to take them to the main features of the application
  - Add income/expense
  - View entries
  - Save and export data (load data is in main menu)
- Loop in place for users to enter valid input between 1-7 to choose sub menu options

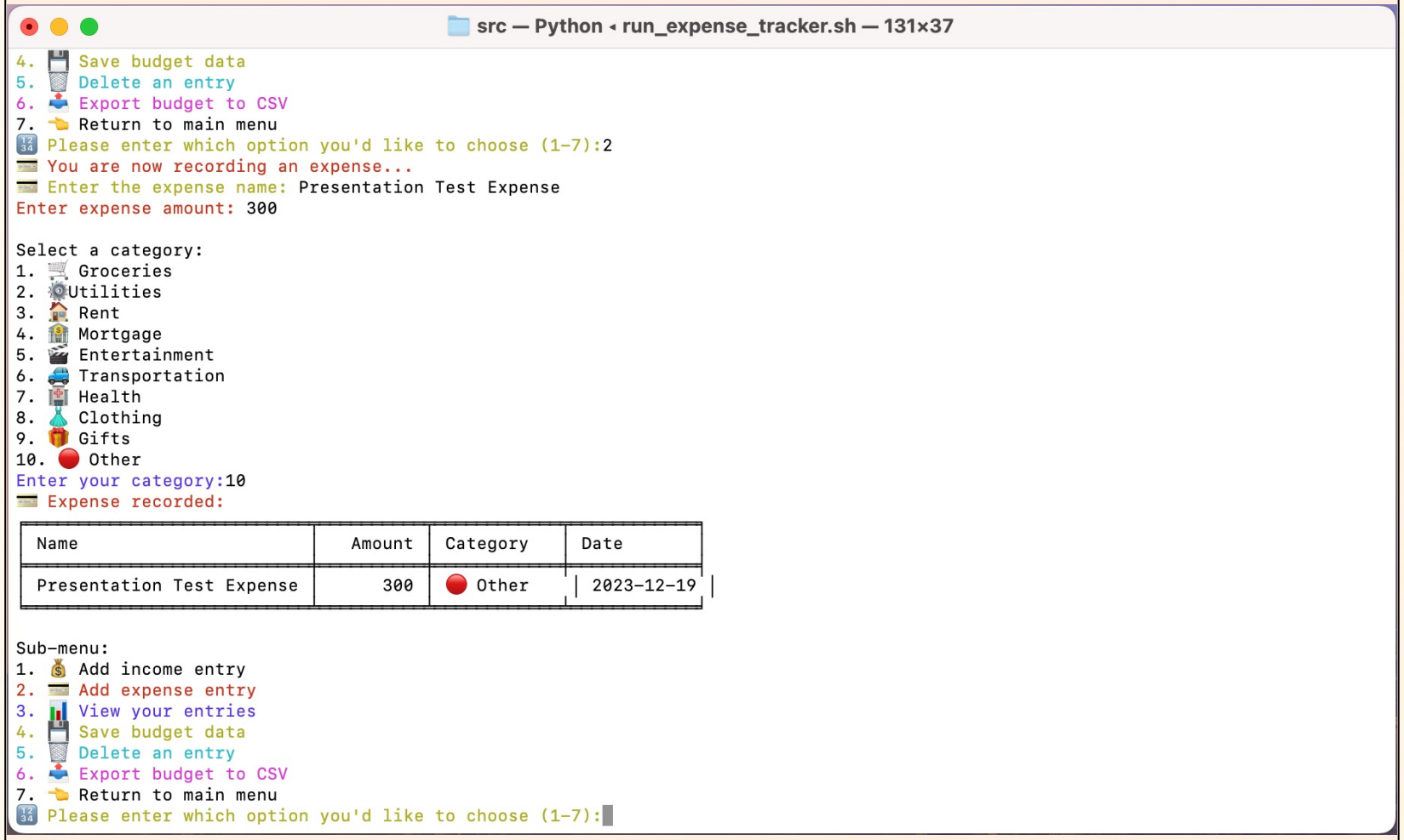
Application Walk-through – Add Income



Features

- User input to add income entry
  - Add income name
  - Add income amount
  - Select income category
  - Date automatically added from datetime package
  - Data printed nicely with tabulate package

Application Walk-through – Add Expense



Features

- User input to add expense entry
  - Add expense name
  - Add expense amount
  - Select expense category
  - Date automatically added from datetime package
  - Data printed nicely with tabulate package

Application Walk-through – View Entries (by Month)

src — Python ◂ run_expense_tracker.sh — 131×37

```
Sub-menu:
1. 💰 Add income entry
2. 💳 Add expense entry
3. 📊 View your entries
4. 💾 Save budget data
5. 🗑️ Delete an entry
6. 📤 Export budget to CSV
7. 👈 Return to main menu
🔢 Please enter which option you'd like to choose (1-7):3
viewing entries...
List of available months:
1. December 2023
Please enter which month you would like to view:1

View Options:
1. View your entries for a specific month
2. View your entries sorted by category for a specific month
Please enter your choice (1 or 2), or type 'r' to return to the sub menu:1

🗒️ Entries for December 2023:
💰 Incomes
```

Name	Amount	Date
Presentation Test Income	500	2023-12-19

💳 Expenses:

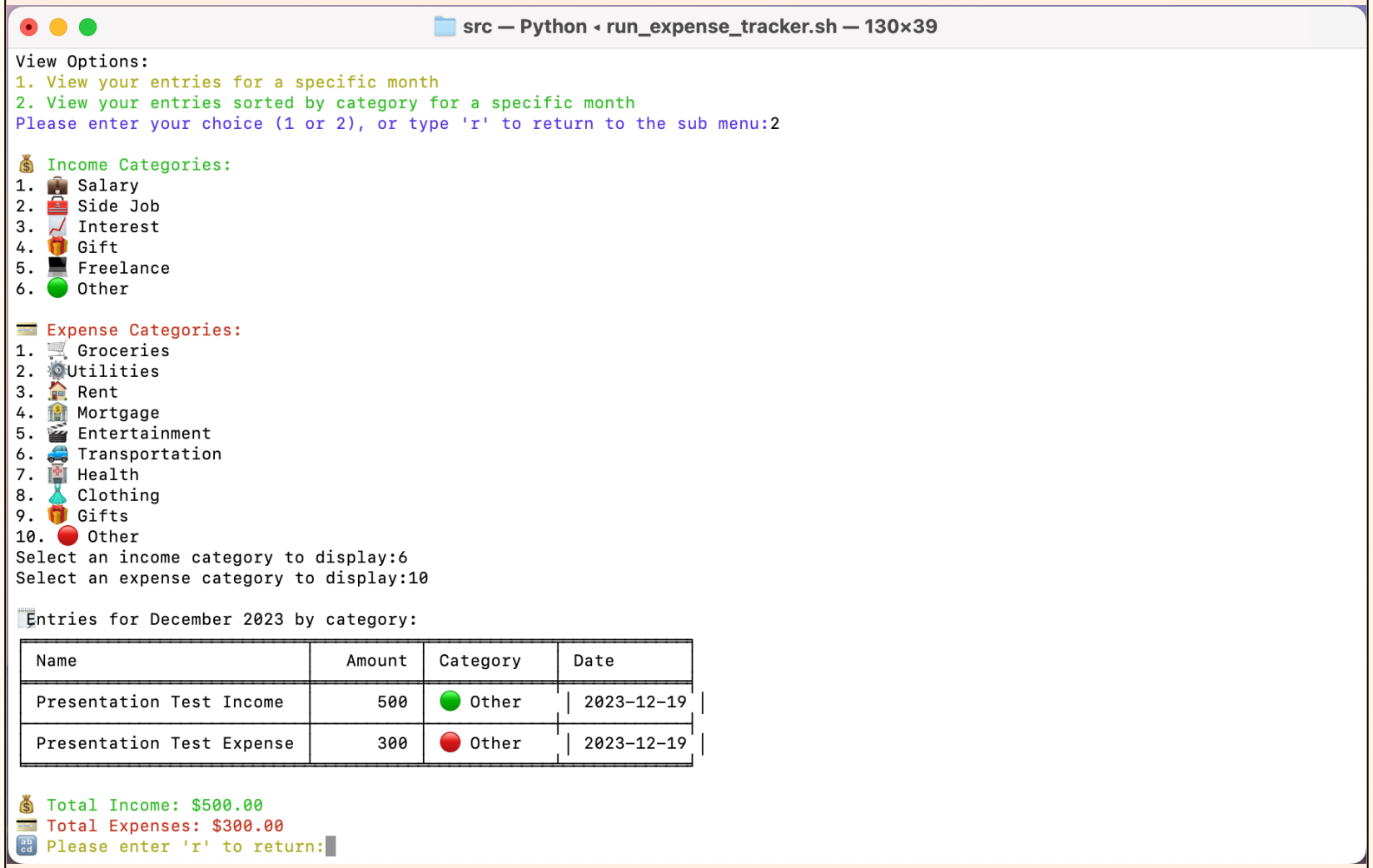
Name	Amount	Date
Presentation Test Expense	300	2023-12-19

```
💰 Total Income: $500.00
💳 Total Expenses: $300.00
🔡 Please enter 'r' to return:
```

Features

- User input to select between viewing income and expense entries by month, or by category
- Allows users to choose entries for months that contain entries
- Displays all entries for selected month nicely with tabulate package
- Displays total income and expense amount for the month

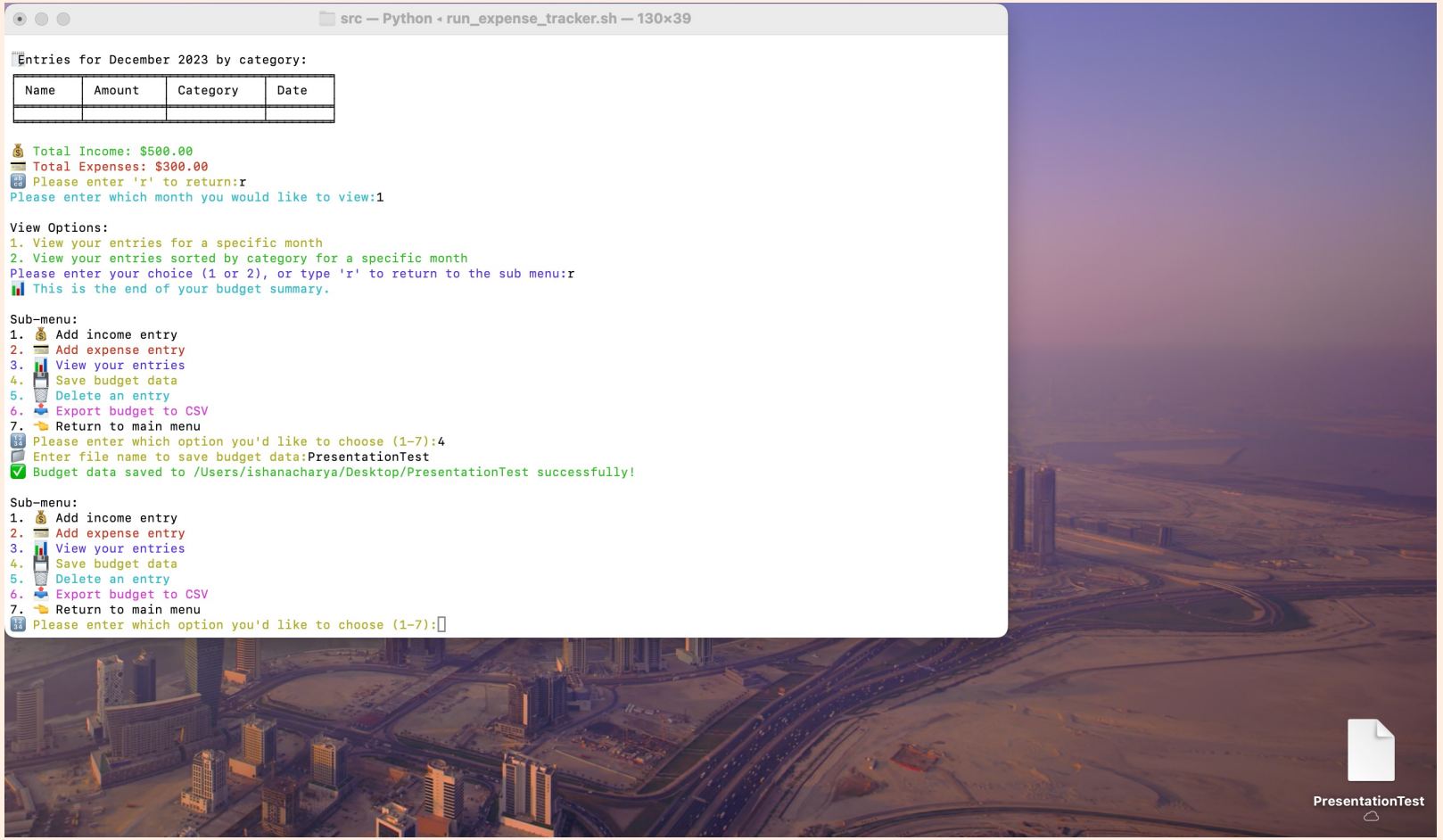
Application Walk-through – View Entries (by Category)



Features

- User input to select between viewing income and expense entries by month, or by category
- Allows users to choose to display entries based on income categories and expense categories
- Displays all entries for selected income and expense categories with tabulate package
- Displays total income and expense amount for the month

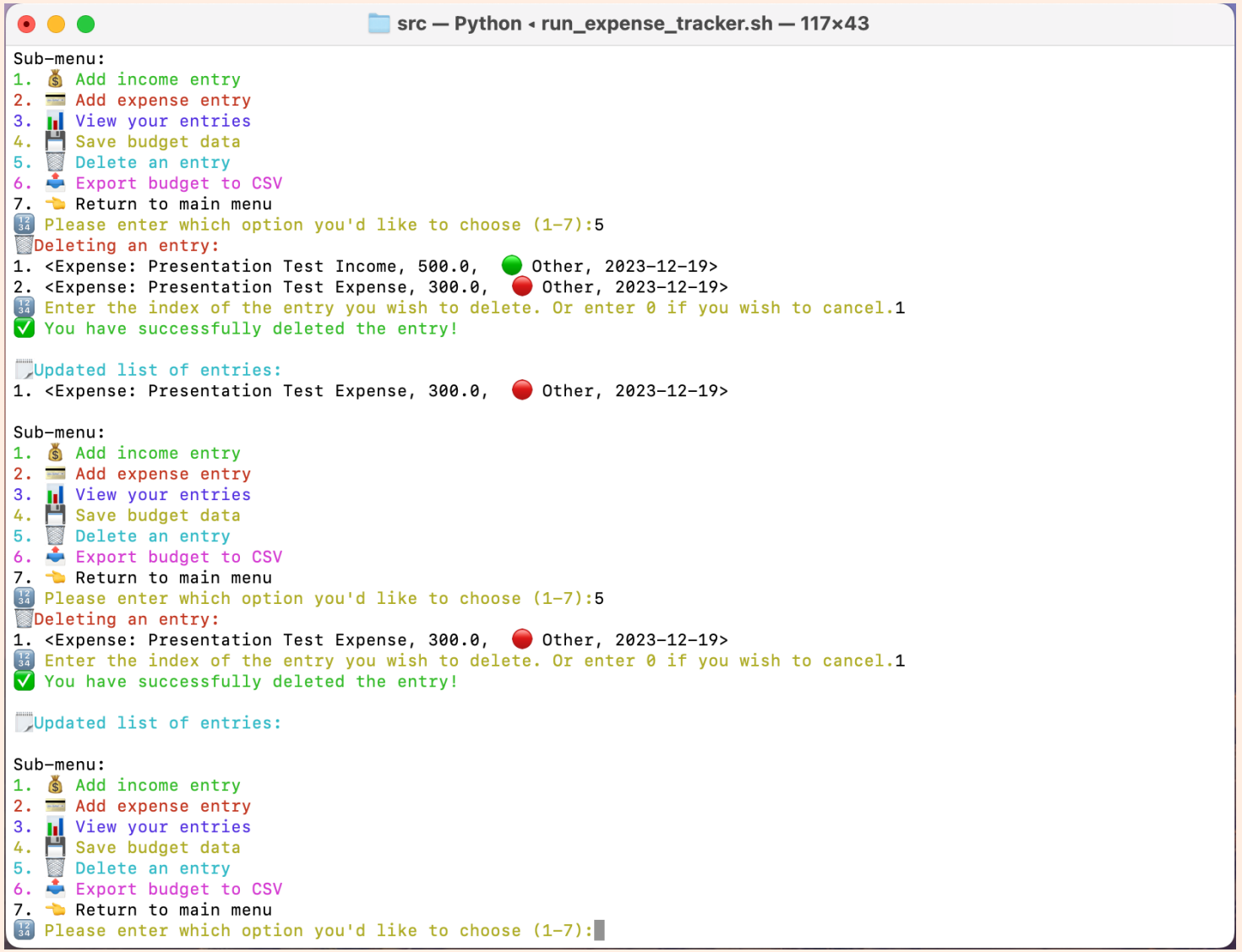
Application Walk-through – Save Data



Features

- User input to name file to save budget data to .txt file
- File saved in user's desktop location (os package to define filepath)
- Contains list of income and expense entries that can be restored back once file is loaded

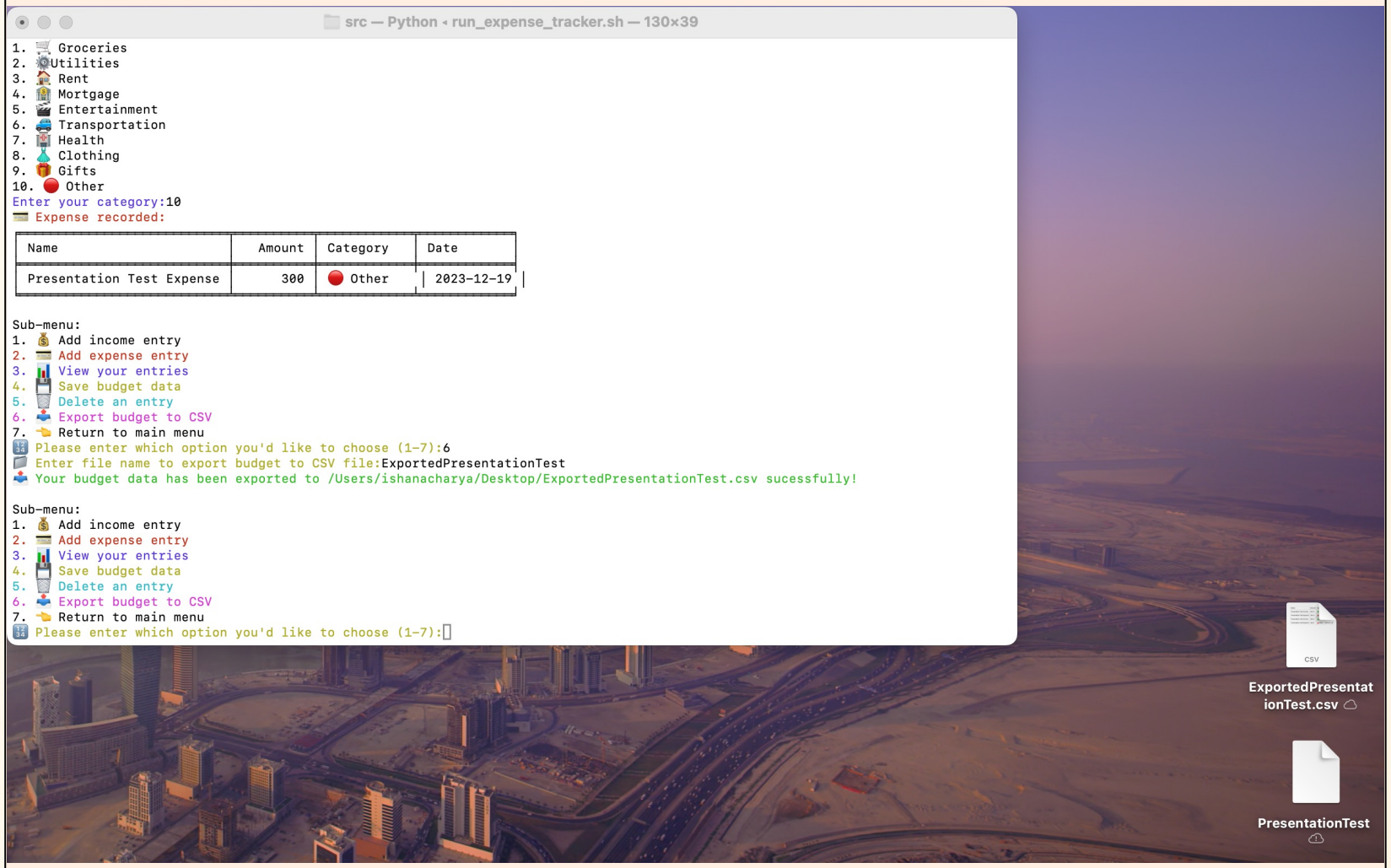
Application Walk-through – Delete Entry



Features

- Displays all current income and expense entries that are assigned against index number
- Users can provide input to delete entry of their choice by typing respective index number
- Once entry has been deleted, an updated list of entries is shown

Application Walk-through – Export Data



Features

- User input to name file to export budget data to .csv file
- File saved on user's desktop location
- Contains tabulated data of income and expense entries that can be used in external software for further analysis (eg. Excel, MYOB, etc.)

Application Walk-through – Return to Main Menu

src — Python ◂ run_expense_tracker.sh — 130×39

```
9. 🎁 Gifts
10. 🔴 Other
Enter your category:10
💳 Expense recorded:
```

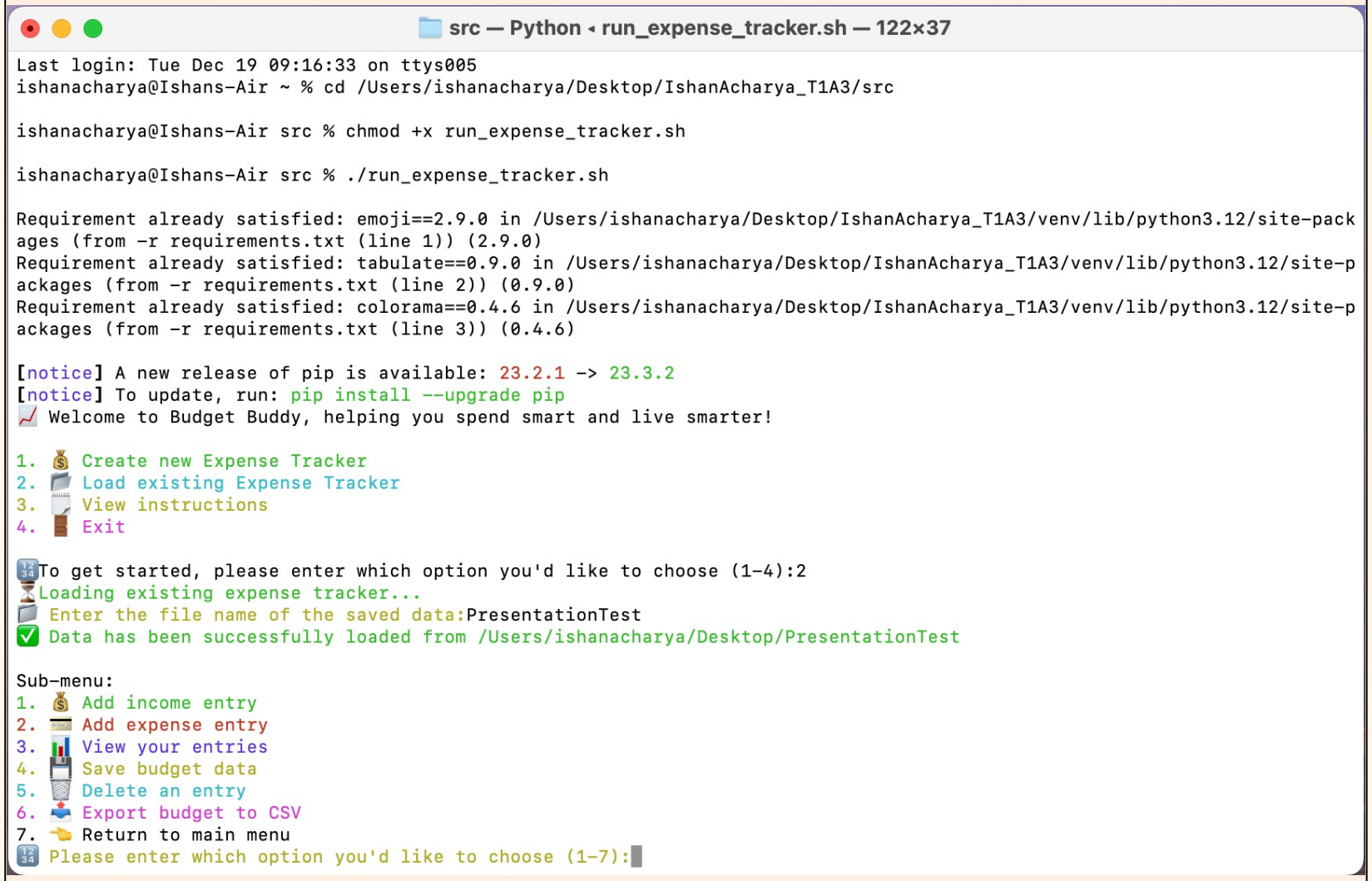
Name	Amount	Category	Date
Presentation Test Expense	300	🔴 Other	2023-12-19

```
Sub-menu:
1. 💰 Add income entry
2. 💳 Add expense entry
3. 📊 View your entries
4. 💾 Save budget data
5. 🗑️ Delete an entry
6. 📤 Export budget to CSV
7. 👈 Return to main menu
🔢 Please enter which option you'd like to choose (1-7):6
📁 Enter file name to export budget to CSV file:ExportedPresentationTest
📤 Your budget data has been exported to /Users/ishanacharya/Desktop/ExportedPresentationTest.csv sucessfully!

Sub-menu:
1. 💰 Add income entry
2. 💳 Add expense entry
3. 📊 View your entries
4. 💾 Save budget data
5. 🗑️ Delete an entry
6. 📤 Export budget to CSV
7. 👈 Return to main menu
🔢 Please enter which option you'd like to choose (1-7):7
👈 Returning back to the main menu.

1. 💰 Create new Expense Tracker
2. 📁 Load existing Expense Tracker
3. 📜 View instructions
4. 🚪 Exit

🔢To get started, please enter which option you'd like to choose (1-4):
```

Features

- User input to return back to the main menu

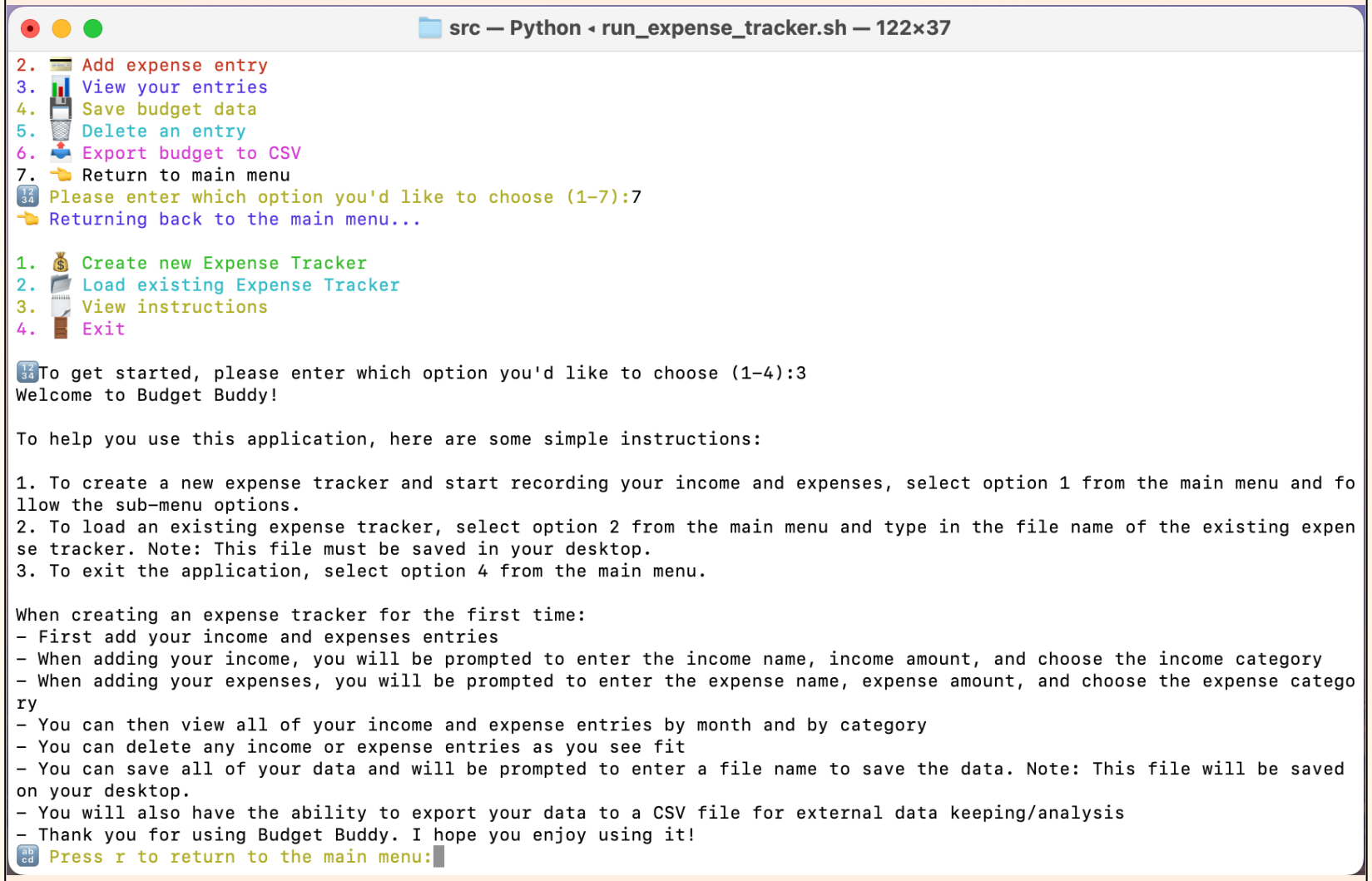
Application Walk-through – Load Data



Features

- User input to load saved file
- Looks for file on user's desktop location, where file was originally saved
- Restores data (income and expense entries) once file has been loaded

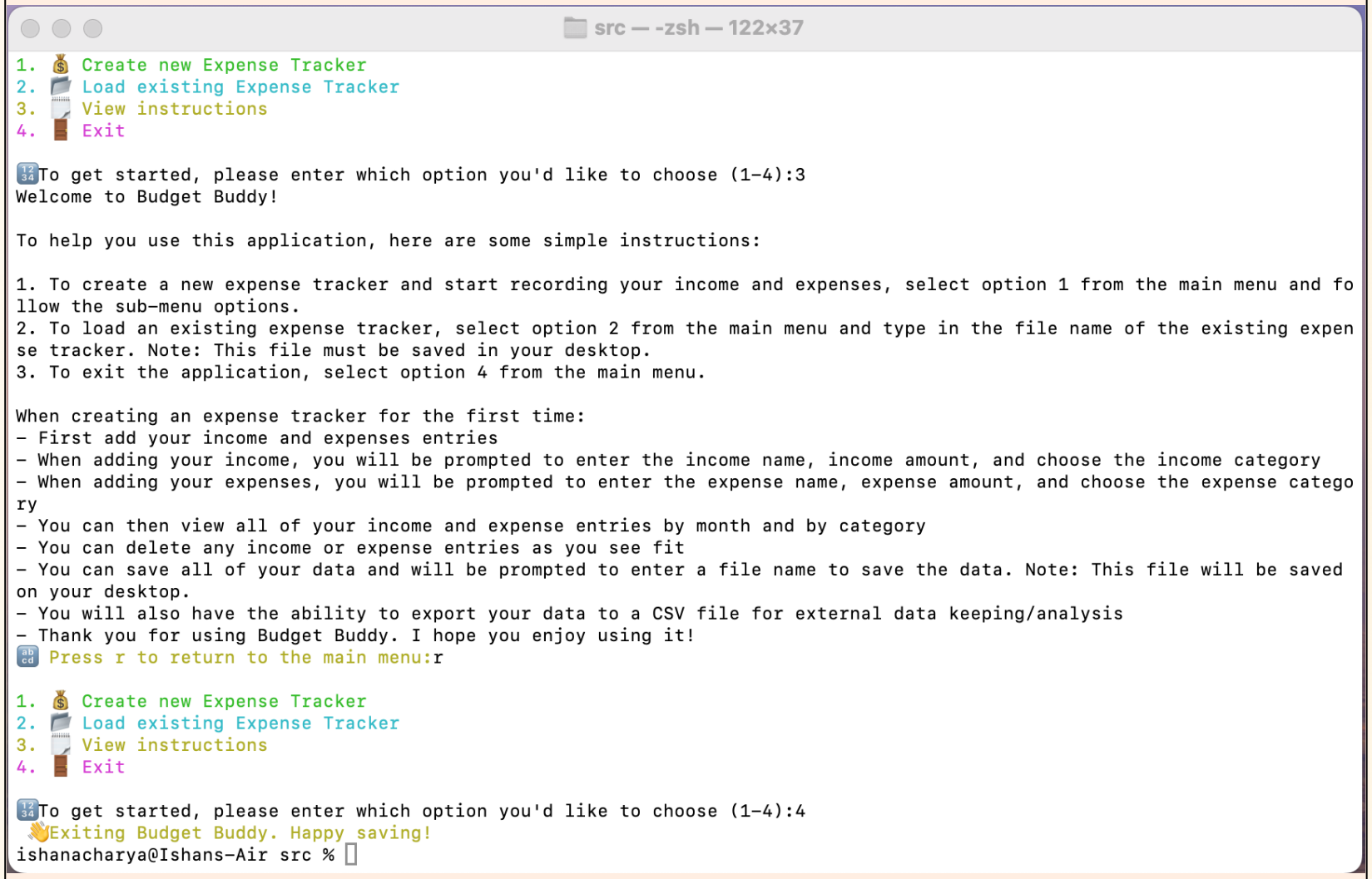
Application Walk-through – View Instructions



Features

- User input to view instructions
- Displays set of instructions on how to use the application
- User input to return back to the main menu

Application Walk-through – Exit Program



Features

- User input to return back to the main menu

Code Overview

- **Two files used for application code**
  - main.py – contains main logic for running the application by initialising the app and interacts with expense_tracker.py and ExpenseTracker class to deliver its features
  - expense_tracker.py – contains ExpenseTracker class and 15 functions to deliver the core functionality of the application
- **Imports and utilises three internal and external Python packages**
  - Tabulate
  - Colorama
  - Emoji
  - Datetime
  - csv
  - os
- **Usage of various key logic across the three key main application features**
  - Variables, loops, conditional control structures, error handling, functions, input and output, importing python packages, and using functions from Python Packages

Code Overview – Main Function (main.py)

```python
#Import ExpenseTracker class from expense_tracker module
from expense_tracker import ExpenseTracker
#Import colorama and emoji packages
from colorama import Fore
import emoji

# Main function for application
def main():
    #Welcome message
    print(f"{Fore.BLACK}{emoji.emojize(':chart_increasing:')} Welcome to Budget Buddy, helping you spend smart and live smarter!{Fore.RESET}")

    #Create instance of class
    expense_tracker = ExpenseTracker()
    #Application loop
    is_running = True

    while is_running:
        #Main menu options
        print(f"\n{Fore.GREEN}1. {emoji.emojize(':money_bag:')} Create new Expense Tracker{Fore.RESET}")
        print(f"{Fore.CYAN}2. {emoji.emojize(':open_file_folder:')} Load existing Expense Tracker{Fore.RESET}")
        print(f"{Fore.YELLOW}3. {emoji.emojize(':spiral_notepad:')}  View instructions{Fore.RESET}")
        print(f"{Fore.MAGENTA}4. {emoji.emojize(':door:')} Exit\n{Fore.RESET}")

       #User input to select menu options
        choice = input(f"{Fore.BLACK}{emoji.emojize(':input_numbers:')}To get started, please enter which option you'd like to choose (1-4):{Fore.RESET}")

        if choice == "1":
            expense_tracker.create_new_expense_tracker()
        elif choice == "2":
            expense_tracker.load_expense_tracker()
        elif choice == "3":
            expense_tracker.view_instructions()
        elif choice == "4":
            #Exiting application
            print(f"{Fore.YELLOW} {emoji.emojize(':waving_hand:')}Exiting Budget Buddy. Happy saving!{Fore.RESET}")
            is_running = False
        else:
            #Error handling for invalid input
            print(f"{Fore.RED}{emoji.emojize(':warning:')} Your input is invalid. Please choose a menu option between 1-4.{Fore.RESET}")

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

Code Logic

- **Use of variables**
  - 'expense_tracker' stores ExpenseTracker class
  - 'is_running' stores 'true' to run application
  - 'choice' stores user input
- **Use of loops and conditional control structures**
  - 'While' loop to run if is_running = True and display main menu
  - If/elif/else statements used to control user's menu option choices
- **Error handling**
  - Message displayed for invalid input if not between 1-4
- **Write and use simple functions**
  - 'main' function contains main logic for application
- **Input and output**
  - 'input' utilised for user's choice
  - 'print' utilised for output display
- **Importing python package**
  - 'from colorama import Fore'
  - 'import emoji'
- **Using functions from a python package**
  - Colorama and emoji utilised with print functions to enhance display with coloured text and emojis

Code Overview – ExpenseTracker Class (Expense_Tracker.py)

```python
from tabulate import tabulate # Import tabulate to create tables when displaying data
from colorama import Fore # Import colorama to add coloured text within expense tracker
import emoji # Import emoji module to add emojis to expense tracker
import csv #Import csv module to handle csv files
import datetime #Import datetime to implement time and date within expense tracker
import os # Import os to define file path to save/load/export data to user's desktop

# Class for Expense Tracker application
class ExpenseTracker:

    # List to define income categories
    income_categories = [
        emoji.emojize(":briefcase: Salary"),
        emoji.emojize(":toolbox: Side Job"),
        emoji.emojize(":chart_increasing: Interest"),
        emoji.emojize(":wrapped_gift: Gift"),
        emoji.emojize(":laptop: Freelance"),
        emoji.emojize(":green_circle: Other")]

    # List to define expense categories
    expense_categories = [
        emoji.emojize(":shopping_cart: Groceries"),
        emoji.emojize(":gear: Utilities"),
        emoji.emojize(":house: Rent"),
        emoji.emojize(":bank: Mortgage"),
        emoji.emojize(":clapper_board: Entertainment"),
        emoji.emojize(":sport_utility_vehicle: Transportation"),
        emoji.emojize(":hospital: Health"),
        emoji.emojize(":dress: Clothing"),
        emoji.emojize(":wrapped_gift: Gifts"),
        emoji.emojize(":red_circle: Other")]

    def __init__(self):
        # Empty list to store monthly income and expense data
        self.monthly_data = []
```

Code Logic

- **Class**
  - ExpenseTracker class contains functionalities of entire application
  - 'income_categories' and 'expense_categories' attributes contain lists to define income and expense categories'
- **Use of variables**
  - 'monthly_data' stores monthly income and expense entry data within every instance of the ExpenseTracker class
- **Importing python package**
  - 'import emoji'
  - 'import csv'
  - 'import datetime'
  - 'import os'
  - 'from tabulate import tabulate'
  - 'from colorama import Fore'
- **Using functions from a python package**
  - Utilises aforementioned packages within the ExpenseTracker class
    - Create tables to display data
    - Add coloured text and emojis to display
    - Handle csv files
    - Manage date and time for entries
    - Work with users' operating system to define download/upload file paths

Code Overview – Create New Expense Tracker

```python
# Create new Expense Tracker
    def create_new_expense_tracker(self):
        print(f"{Fore.GREEN}{emoji.emojize(':hourglass_not_done:')} Creating new expense tracker...{Fore.RESET}")

        # Data structure to store monthly income and expenses
        self.monthly_data = []

        while True:
            # Sub-menu within options to navigate around application
            print("\nSub-menu:")
            print(f"{Fore.BLACK}1. {emoji.emojize(':money_bag:')} Add income entry{Fore.RESET}")
            print(f"{Fore.RED}2. {emoji.emojize(':credit_card:')} Add expense entry{Fore.RESET}")
            print(f"{Fore.BLUE}3. {emoji.emojize(':bar_chart:')} View your entries{Fore.RESET}")
            print(f"{Fore.YELLOW}4. {emoji.emojize(':floppy_disk:')} Save budget data{Fore.RESET}")
            print(f"{Fore.CYAN}5. {emoji.emojize(':wastebasket:')}  Delete an entry{Fore.RESET}")
            print(f"{Fore.MAGENTA}6. {emoji.emojize(':outbox_tray:')} Export budget to CSV{Fore.RESET}")
            print(f"{Fore.BLACK}7. {emoji.emojize(':backhand_index_pointing_left:')} Return to main menu{Fore.RESET}")

            # User input to select sub menu options
            sub_choice = input(f"{Fore.YELLOW}{emoji.emojize(':input_numbers:')} Please enter which option you'd like to choose (1-7):{Fore.RESET}")

            if sub_choice == "1":
                self.record_income()
            elif sub_choice == "2":
                self.record_expense()
            elif sub_choice == "3":
                self.view_budget()
            elif sub_choice == "4":
                expense_file_path = input(f"{Fore.YELLOW}{emoji.emojize(':file_folder:')} Enter file name to save budget data:{Fore.RESET}")
                self.save_data_to_file(expense_file_path)
            elif sub_choice == "5":
                self.delete_entry()
            elif sub_choice == "6":
                csv_file_name = input(f"{Fore.YELLOW}{emoji.emojize(':file_folder:')} Enter file name to export budget to CSV file:{Fore.RESET}")
                self.export_to_csv(csv_file_name)
            elif sub_choice == "7":
                # Display message to return to main menu
                print(f"{Fore.BLUE}{emoji.emojize(':backhand_index_pointing_left:')} Returning back to the main menu.{Fore.RESET}")
                break

            else:
                # Error handling message for invalid input
                print(f"{Fore.RED}{emoji.emojize(':warning:')} Your input is invalid. Please choose a menu option between 1-7.{Fore.RESET}")
```

Code Logic

- **Use of variables**
  - 'self.monthly_data', 'sub_choice', 'expense_file_path', and 'csv_file_name' capture user input and data for various functions
- **Loops and conditional control structures**
  - 'while True' loop for menu choice/options until it is broken
  - if/elif/else statements based on user's menu option choices to direct them around application
- **Error Handling**
  - Message displayed for invalid input if not between 1-7
- **Input and output**
  - 'sub_choice', 'expense_file_path' and 'csv_file_name' utilised for user's input
  - 'print' utilised to display sub menu options and messages
- **Using functions from a python package**
  - Colorama and emoji packages utilised with print functions to enhance display with coloured text and emojis
- **Write and use simple functions**
  - Menu links to various functions
    - 'record_income()'
    - 'record_expense()'
    - 'view_budget()'
    - 'save_data_to_file()'
    - 'delete_entry()'
    - 'export_to_csv()'

Code Overview – Record Income

```python
# Record new income entry
    def record_income(self):
        # Display message indicating user is recording an income entry
        print(f"{Fore.GREEN}{emoji.emojize(':money_bag:')} You are now recording an income.{Fore.RESET}")

        # Data validation for valid income name
        while True:
            income_name = input(f"{Fore.GREEN}{emoji.emojize(':money_bag:')} Enter the income name: {Fore.RESET}")
            if income_name.replace(" ", "").isalpha():
                break
            else:
                # Error handling message for invalid input
                print(f"{Fore.RED}{emoji.emojize(':warning:')} Sorry your input is invalid. Please make sure the income name only contains text letters and no symbols or numbers.{Fore.RESET}")
        # User input for income amount and category
        income_amount = float(input(f"{Fore.GREEN}Enter income amount: {Fore.RESET}").replace(",", "").replace("$", ""))
        income_category = self.choose_category("income")
        # Current date for income entry from datetime package
        income_date = datetime.date.today()

        # Income object with income name, amount, category and date
        new_income = Income(name=income_name, amount=income_amount, category=income_category, date=income_date)

        # Add income entry to monthy_data list
        self.monthly_data.append(new_income)

        # Display message that income entry has been recorded
        print(f"{Fore.GREEN}{emoji.emojize(':money_bag:')}Income recorded:{Fore.RESET}")
        # Display recorded income entry within tabulate table
        print(tabulate([(new_income.name, f"{new_income.amount:.2f}", new_income.category, new_income.date)], headers=["Name", "Amount", "Category", "Date"], tablefmt="fancy_grid"))
```

Code Logic

- **Use of variables**
  - 'income_name', 'income_amount', 'income_category', 'income_date' and 'new_income' store user input data
- **Loops and conditional control structures**
  - 'While True' loop utilised for data validation for income name
  - if/else statements utilised for invalid input
- **Error Handling**
  - Message displayed for invalid 'income_name' input if it contains numbers or symbols
- **Input and output**
  - 'input' utilised for user's choice
  - 'print' utilised for output display and 'tabulate' utilised to display data in tabulated format
- **Using functions from a python package**
  - Tabulate package utilised to display income entry data in tabulated format
  - Colorama and emoji packages utilised with print functions to enhance display with coloured text and emojis

Code Overview – Record Expense

```python
# Record new expense entry
    def record_expense(self):
        # Display message indicating user is recording an expense entry
        print(f"{Fore.RED}{emoji.emojize(':credit_card:')} You are now recording an expense...{Fore.RESET}")

        # Data validation for valid expense name
        while True:
            expense_name = input(f"{Fore.YELLOW}{emoji.emojize(':credit_card:')} Enter the expense name: {Fore.RESET}")
            if expense_name.replace(" ", "").isalpha():
                break
            else:
                # Error handling message for invalid input
                print(f"{Fore.RED}{emoji.emojize(':warning:')}Sorry your input is invalid. Please make sure the expense name only contains text letters and no symbols or numbers.{Fore.RESET}")

        # User input for expense amount and category
        expense_amount = float(input(f"{Fore.RED}Enter expense amount: {Fore.RESET}").replace(",", "").replace("$", ""))
        expense_category = self.choose_category("expense")
        # Current date for income entry from datetime package
        expense_date = datetime.date.today()

        # Expense object with expense name, amount, category and date
        new_expense = Expense(name=expense_name, amount=expense_amount, category=expense_category, date=expense_date)

        # Add expense entry to monthy_data list
        self.monthly_data.append(new_expense)

        # Display message that expense entry has been recorded
        print(f"{Fore.RED}{emoji.emojize(':credit_card:')} Expense recorded:{Fore.RESET}")
        # Display recorded expense entry within tabulate table
        print(tabulate([(new_expense.name, f"{new_expense.amount:.2f}", new_expense.category, new_expense.date)], headers=["Name", "Amount", "Category", "Date"], tablefmt="fancy_grid"))
```

Code Logic

- **Use of variables**
  - 'expense_name', expense_amount', 'expense_category', 'expense_date' and 'new_expense' store user input data
- **Loops and conditional control structures**
  - 'While True' loop utilised for data validation for expense name
  - if/else statements utilised for invalid input
- **Error Handling**
  - Message displayed for invalid 'expense_name' input if it contains numbers or symbols
- **Input and output**
  - 'input' utilised for user's choice
  - 'print' utilised for output display and 'tabulate' utilised to display data in tabulated format
- **Using functions from a python package**
  - Tabulate package utilised to display expense entry data in tabulated format
  - Colorama and emoji packages utilised with print functions to enhance display with coloured text and emojis

Code Overview – View Entry (by Month)

```python
# View list of income and expense entries for specific month
    def view_budget(self):
        # Display message to indicate that user is viewing entries
        print(f"{Fore.CYAN}viewing entries...{Fore.RESET}")

        # Check to see if there are no entries within expense tracker
        if not self.monthly_data:
            # Error handling message for no data available
            print(f"{Fore.RED}{emoji.emojize(':warning:')} Sorry, no data available. Please add your income or expense entries first.{Fore.RESET}")
            return

        # Pull individual months that contain entries in them
        unique_month = set(entry.date.strftime("%B %Y") for entry in self.monthly_data)

        # Check to see whether there are no entries recorded for any months
        if not unique_month:
            # Error handling message for no available entries data
            print(f"{Fore.RED}{emoji.emojize(':warning:')} You have no available entries available for any month! Please add your entries first.{Fore.RESET}")
            return

        # Display months available that have entries in them
        print(f"{Fore.CYAN}List of available months:{Fore.RESET}")
        for index, month in enumerate(unique_month, start=1):
            print(f"{index}. {month}")

        while True:
            try:
                # User input for selected month
                selected_month_index = int(input(f"{Fore.CYAN}Please enter which month you would like to view:{Fore.RESET}"))
                selected_month = list(unique_month)[selected_month_index - 1]

            except  (ValueError, IndexError):
                # Error handling for invalid input
                print(f"{Fore.RED}{emoji.emojize(':warning:')} Invalid input. Please enter a valid number.{Fore.RESET}")
                continue

            # Filter entries for selected month
            month_entries = [entry for entry in self.monthly_data if entry.date.strftime("%B %Y") == selected_month]

            # Income and expense entries separated for viewing
            incomes = [entry for entry in month_entries if isinstance(entry, Income)]
            expenses = [entry for entry in month_entries if isinstance(entry, Expense)]
```

Code Logic

- **Use of variables**
  - 'self.monthy_data', 'unique_month', 'selected_month', 'month_entries', 'incomes', 'expenses, are utilised to store data and user input
- **Loops and conditional control structures**
  - 'for' loop utilised to check through unique_month for entries
  - 'while True' loop utilised for valid user input for month selection
  - if/else statements used to check if data is available within 'self.monthly_data' and error handling for invalid input
- **Error Handling**
  - Message displayed for invalid user input or selection, and if data is not available
  - 'try except' utilised for invalid user input when selecting month within index range
- **Input and output**
  - 'input' utilised for user's choice
  - 'print' utilised for output display
- **Using functions from a python package**
  - Colorama and emoji packages utilised with print functions to enhance display with coloured text and emojis
  - Datetime package is utilised to assign current date

Code Overview – View Entry (by Month) (cont.)

```
    # Display view options
    print("\nView Options:")
    print(f"{Fore.YELLOW}1. View your entries for a specific month{Fore.RESET}")
    print(f"{Fore.GREEN}2. View your entries sorted by category for a specific month{Fore.RESET}")

    while True:
        option = input(f"{Fore.BLUE}Please enter your choice (1 or 2), or type 'r' to return to the sub menu:{Fore.RESET}")

        if option == 'r':
            break
        elif option in {"1", "2"}:
            break
        else:
            # Error handling for invalid input
            print(f"{Fore.RED}{emoji.emojize(':warning:')} Invalid input. Please choose 1 or 2, or press 'r' to return to the sub menu.{Fore.RESET}")

    if option == "1":
        # Display entries for user selected month
        print(f"\n{emoji.emojize(':spiral_notepad:')}Entries for {selected_month}:")

        # Display income entries
        print(f"{Fore.GREEN}{emoji.emojize(':money_bag:')} Incomes{Fore.RESET}")
        # Display income entry data in tabulated form in table
        print(tabulate([(income.name, f"{income.amount:.2f}", income.date) for income in incomes], headers=["Name", "Amount", "Date"], tablefmt="fancy_grid"))

        # Display expense entries
        print(f"\n{Fore.RED}{emoji.emojize(':credit_card:')} Expenses:{Fore.RESET}")
        # Display expense entry data in tabulated form in table
        print(tabulate([(expense.name, f"{expense.amount:.2f}", expense.date) for expense in expenses], headers=["Name", "Amount", "Date"], tablefmt="fancy_grid"))

        # Display total income and expenses
        self.display_entry_total(incomes, expenses)

        while True:
            user_input = input(f"{Fore.YELLOW}{emoji.emojize(':input_latin_lowercase:')} Please enter 'r' to return:{Fore.RESET}").lower()

            if user_input == 'r':
                break
            else:
                # Error handling message for invalid input
                print(f"{Fore.RED}{emoji.emojize(':warning:')} Invalid input. Please enter r to return.{Fore.RESET}")

    elif option == "2":
        # Entries sorted by category for selected month
        self.view_by_category(selected_month, month_entries)
        # Display total income and expenses
        self.display_entry_total(incomes, expenses)

        while True:
            user_input = input(f"{Fore.YELLOW}{emoji.emojize(':input_latin_lowercase:')} Please enter 'r' to return:{Fore.RESET}").lower()

            if user_input == 'r':
                break
            else:
                # Error handling message for invalid input
                print(f"{Fore.RED}{emoji.emojize(':warning:')} Invalid input. Please enter r to return.{Fore.RESET}")
    else:
        break
# Display message stating the end of the budget summary
print(f"{Fore.CYAN}{emoji.emojize(':bar_chart:')} This is the end of your budget summary.{Fore.RESET}")
```

Code Logic

- **Use of variables**
  - 'option' and 'user_input' are utilised to store the user's choice to either view month or category per month, and for their input to return to the main menu when prompted
- **Loops and conditional control structures**
  - 'while True' loops used for user input to select between viewing entries by month or categories by month, or to return to main menu
  - if/elif/else statements used to check value of 'option' and 'user_input' variables in order to display relevant data
- **Error Handling**
  - Message displayed for invalid user input or selection when trying to choose to view entries by month or categories by month, and for invalid user input when returning to the main menu
- **Input and output**
  - 'input' utilised for user's choice
  - 'print' utilised to display messages, options and data in tables
- **Using functions from a python package**
  - Tabulate package utilised to display income and expense entry data in tabulated format
  - Colorama and emoji packages utilised with print functions to enhance display with coloured text and emojis

Code Overview – Display Entry Total

```python
# Display total income and expense amounts
    def display_entry_total(self, incomes, expenses):
        # Calculate total of income entries
        total_income = sum(income.amount for income in incomes)
        # Calculate total of expense entries
        total_expense = sum(expense.amount for expense in expenses)
        # Display total income
        print(f"\n{Fore.GREEN}{emoji.emojize(':money_bag:')} Total Income: ${total_income:.2f}{Fore.RESET}")
        # Display total expense
        print(f"{Fore.RED}{emoji.emojize(':credit_card:')} Total Expenses: ${total_expense:.2f}{Fore.RESET}")
```

Code Logic

- **Use of variables**
  - 'total_income' and 'total_expense' utilised to store calculated total income and total expense amounts of all income and expense entries
- **Input and output**
  - Entries within 'incomes' and 'expenses' displayed as input
  - Total amount of income and expense displayed as output
- **Using functions from a python package**
  - Colorama and emoji packages utilised with print functions to enhance display with coloured text and emojis

Code Overview – View Entry (by Category)

```python
# View list of income and expense entries sorted by category for a specific month
    def view_by_category(self, selected_month, month_entries):
        # Pull pre-defined list of income and expense categories from ExpeenseTracker class
        income_categories = ExpenseTracker.income_categories
        expense_categories = ExpenseTracker.expense_categories

        # Display income categories
        print(f"\n{Fore.GREEN}{emoji.emojize(':money_bag:')} Income Categories:{Fore.RESET}")
        for i, category in enumerate(income_categories, start=1):
            print(f"{i}. {category}")

        # Display expense categories
        print(f"\n{Fore.RED}{emoji.emojize(':credit_card:')} Expense Categories:{Fore.RESET}")
        for i, category in enumerate(expense_categories, start=1):
            print(f"{i}. {category}")

        # Income and expense categories to display
        selected_income_category = int(input("Select an income category to display:"))
        selected_expense_category = int(input("Select an expense category to display:"))

        # Filter entries by category
        filtered_entries = []

        # Include income and expense categories for month if conditions met
        for entry in month_entries:
            if selected_income_category == 0 and isinstance(entry, Income):
                filtered_entries.append(entry)
            elif selected_expense_category == 0 and isinstance(entry, Expense):
                filtered_entries.append(entry)
            elif isinstance(entry, Income) and entry.category == income_categories[selected_income_category -1]:
                filtered_entries.append(entry)
            elif isinstance(entry, Expense) and entry.category == expense_categories[selected_expense_category -1]:
                filtered_entries.append(entry)

        # Displayed entries by category
        print(f"\n{emoji.emojize(':spiral_notepad:')}Entries for {selected_month} by category:")
        # Empty list to store entries in tabulate format
        entries_table = []
        # Add entries to the entries table list
        for entry in filtered_entries:
            entries_table.append([entry.name, f"{entry.amount:.2f}", entry.category, entry.date])
        # Display entries with categories within tabulate table
        print(tabulate(entries_table, headers=["Name", "Amount", "Category", "Date"], tablefmt="fancy_grid"))
```

Code Logic

- **Use of variables**
  - 'income_categories', 'expense_categories', 'selected_income_category', 'selected_expense_category' and 'filtered_entries' are utilised to store pre-defined lists of income and expense categories, user input on selecting categories, entries that are filtered for income and expense categories, and displaying tabulated data in a table
- **Loops and conditional control structures**
  - 'for' loops utilised to check through income and expense categories and display the respective entries from the selected categories
  - if/else statements utilised to filter entries within income and expense categories
- **Input and output**
  - 'input' utilised for user input in selecting income and expense categories
  - 'print' utilised to display messages, options and tabulated data in tables
- **Using functions from a python package**
  - Tabulate package utilised to display income and expense entry data in tabulated format
  - Colorama and emoji packages utilised with print functions to enhance display with coloured text and emojis

Code Overview – Save Expense Tracker Data

```python
# Save data to file
    def save_data_to_file(self, file_name):
        # Path to user's desktop to save file
        desktop_path = os.path.join(os.path.expanduser("~"), "Desktop")
        # Defining full path for CSV file on the user's desktop
        expense_file_path = os.path.join(desktop_path, file_name)

        try:
            with open(expense_file_path, 'w') as file:
                for expense in self.monthly_data:
                    file.write(f"{expense.name}, {expense.amount}, {expense.category}, {expense.date}\n")

            print(f"{Fore.GREEN}{emoji.emojize(':check_mark_button:')} Budget data saved to {expense_file_path} successfully!{Fore.RESET}")

        except FileNotFoundError:
            # Error handling message for invalid input
            print(f"{Fore.RED}{emoji.emojize(':warning:')} Error: File not found. Please check the file path and try again{Fore.RESET}")

        except Exception as e:
            # Error handling message for invalid input
            print(f"{Fore.RED}{emoji.emojize(':warning:')} Error saving file: {e}{Fore.RESET}")
```

Code Logic

- **Use of variables**
  - 'desktop_path' and 'expense_file_path' are both utilised to store user input when providing file name and the user's desktop path by the os package
- **Loops and conditional control structures**
  - 'try except' blocks utilised to write data on save file and to save file with valid input, and to identify exceptions for possible errors
- **Error Handling**
  - 'try except' blocks are utilised to catch 'FileNotFoundError' if user provides invalid file name input
  - Message displayed for invalid user input and if file could not be found
- **Input and output**
  - 'input' utilised for user input for entering 'file _name' to save file
  - 'print' utilised to display messages to show successful save of file, or if there was an error saving the file
- **Using functions from a python package**
  - Colorama and emoji packages utilised with print functions to enhance display with coloured text and emojis
  - os package utilised to handle file paths to save file to user's desktop location

Code Overview – Load Expense Tracker Data

```python
# Load existing Expense Tracker from a file
    def load_expense_tracker(self):
        # Display message to indicate expense tracker is loading
        print(f"{Fore.GREEN}{emoji.emojize(':hourglass_not_done:')}Loading existing expense tracker...{Fore.RESET}")
        # Path to user's desktop to export file
        desktop_path = os.path.join(os.path.expanduser("~"), "Desktop")
        # User input for file name of saved data
        file_name = input(f"{Fore.YELLOW}{emoji.emojize(':file_folder:')} Enter the file name of the saved data:{Fore.RESET}")
        # Defining full path for file on the user's desktop
        full_file_path = os.path.join(desktop_path, file_name)

        try:
            # Open file
            with open(full_file_path, 'r') as file:
                reader = csv.reader(file)

                self.monthly_data = []

                for row in reader:
                    name, amount, category, date_str = row
                    date_str = date_str.strip()
                    date_format_in_file = '%Y-%m-%d'
                    date = datetime.datetime.strptime(date_str, date_format_in_file).date()

                    if category in self.income_categories:
                        new_entry = Income(name, float(amount), category, date)
                    else:
                        new_entry = Expense(name, float(amount), category, date)
                    self.monthly_data.append(new_entry)

            print(f"{Fore.GREEN}{emoji.emojize(':check_mark_button:')} Data has been successfully loaded from {full_file_path}{Fore.RESET}")
            # Data loaded successfully set to True
            loaded_successfully = True

        except FileNotFoundError:
            # Error handling message for invalid input
            print(f"{Fore.RED}{emoji.emojize(':warning:')} Error: File {full_file_path} could not be found. Please check the file name and try again.{Fore.RESET}")
            # Data loaded successfully set to False
            loaded_successfully = False

        except Exception as e:
            # Error handling message for invalid input
            print(f"{Fore.RED}{emoji.emojize(':warning:')} Error loading expense data: {e}{Fore.RESET}")

            # Data loaded successfully set to False
            loaded_successfully = False
```

Code Logic

- **Use of variables**
  - 'desktop_path', 'file_name', 'full_file_path', 'name', 'amount', 'category', 'date' are utilised to store data and print/load data to/from the file
- **Loops and conditional control structures**
  - 'for' loop and 'try except' block are utilised to read and load data from the saved file if they have valid input
  - if/else statements utilised to check if loaded data contains income or expense data
- **Error Handling**
  - 'try except' blocks are utilised to catch 'FileNotFoundError' if user provides invalid file name input
  - Message displayed for invalid user input and if file could not be found
- **Input and output**
  - 'input' utilised for user input for entering 'file _name' to load file
  - 'print' utilised to display messages to show successful load of file, or if there was an error loading the file
- **Using functions from a python package**
  - Colorama and emoji packages utilised with print functions to enhance display with coloured text and emojis
  - os package utilised to handle file paths to save file to user's desktop location
  - Datetime package utilised to obtain date from saved/loaded data file

Code Overview – Export Data (CSV)

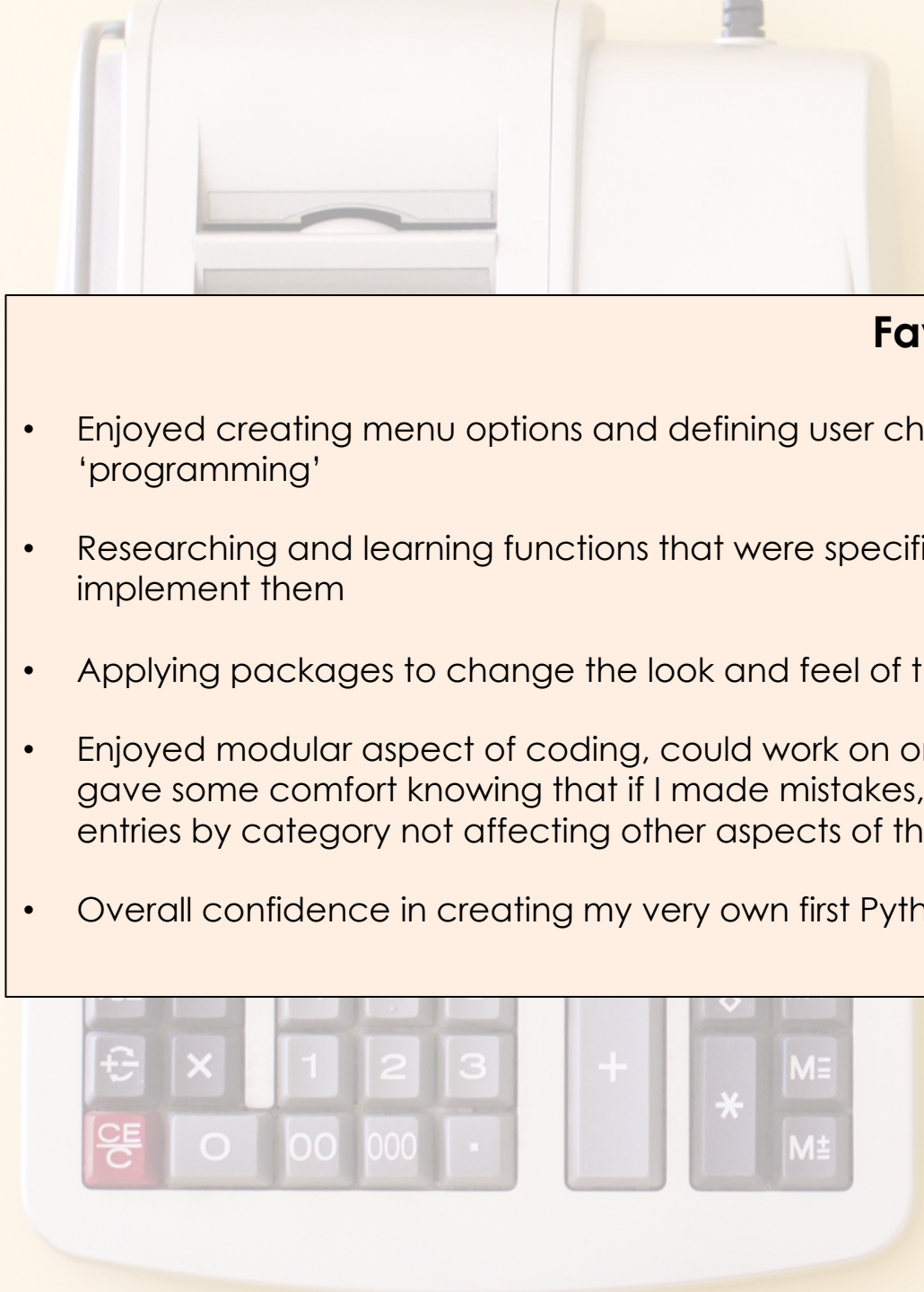
```python
# Export budget data to CSV file
    def export_to_csv(self, csv_file_name):
        # Path to user's desktop to export file
        desktop_path = os.path.join(os.path.expanduser("~"), "Desktop")
        # Defining full path for CSV file on the user's desktop
        csv_file_path = os.path.join(desktop_path, csv_file_name + ".csv")

        try:
            # Open csv file with new line
            with open(csv_file_path, 'w', newline='') as csvfile:
                writer = csv.writer(csvfile)
                # Header for csv file
                writer.writerow(['Name', 'Amount', 'Category', 'Date'])
                # Write income entries to csv file
                for income in self.monthly_data:
                    writer.writerow([income.name, income.amount, income.category, income.date])
                else:
                    # Write expense entries to csv file
                    for expense in self.monthly_data:
                        writer.writerow([expense.name, expense.amount, expense.category, expense.date])
            # Display message if file has been successfully exported
            print(f"{Fore.GREEN}{emoji.emojize(':outbox_tray:')} Your budget data has been exported to {csv_file_path} sucessfully!{Fore.RESET}")

        except Exception as e:
            # Error handling message for invalid file name
            print(f"{Fore.RED}{emoji.emojize(':warning:')} There was an error exporting your budget data. Please check the file name and try again.{Fore.RESET}")
```

Code Logic

- **Use of variables**
  - 'csv_file_path', 'csv_file_name' and 'desktop_path' all utilised to store user input on file name and data pertaining to where the file will be exported to (user's desktop location)
- **Loops and conditional control structures**
  - 'with' statement utilised to open csv file and print income and expense data to csv file, 'try' block utilised to handle file operations
- **Error Handling**
  - 'try except' block utilised to handle errors during file operation or writing data onto csv file that is to be exported
  - Message displayed if error encountered during exporting file
- **Input and output**
  - 'input' is utilised through user input when providing 'csv_file_name'
  - 'print' utilised to display messages to show successful export of file, or if there was an error exporting the file
- **Using functions from a python package**
  - Colorama and emoji packages utilised with print functions to enhance display with coloured text and emojis
  - os package utilised to handle file paths to export file to user's desktop location
  - csv package utilised to write csv data on file that is to be exported

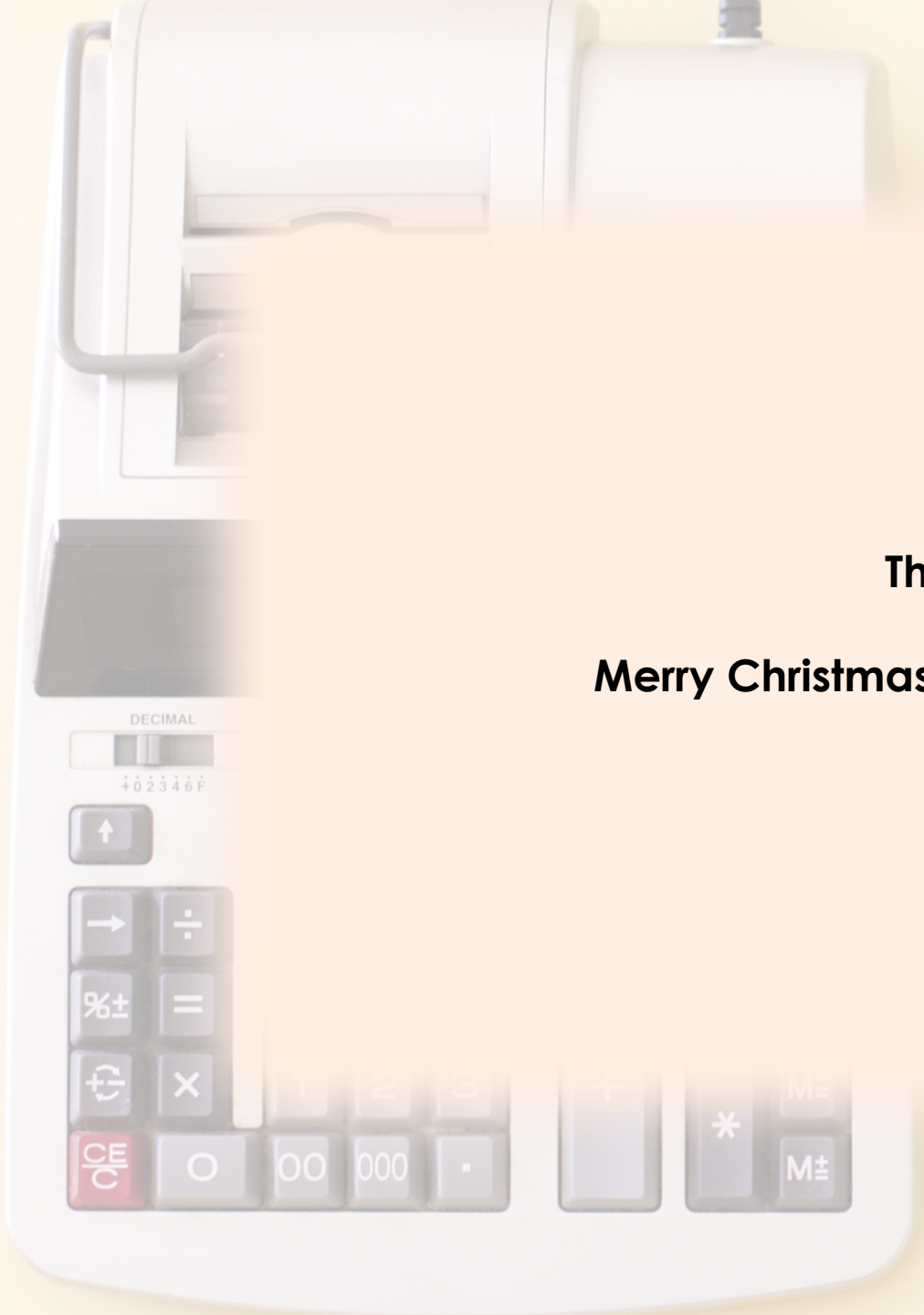
Challenges and Ethical Issues

- **Challenges**
    - Initial challenges implementing packages into functions (but was easy after watching guides/reading package notes)
    - Keeping DRY principle in check, particularly in my view_budget function (reflection post-development)
    - Implementing saving data and exporting file to csv function(s), was made a lot easier with os package (defining file path)
    - Filtering entries for income/expense categories was time-consuming, kept running into needing more functions set up in order to enable the function to work. Good learning experience in seeing linkage between functions
- **Ethical Issues**
- Little to no user or data privacy due to limited scope of project and my lack of expertise to implement security measures (addressed with disclaimer in readme file)
- Lack of accuracy for real-world financial reporting usage (addressed with disclaimer in readme file)
- Variance of users with different skill levels (provided user-friendly prompts for input and error handling, and added instructions to main menu to assist users on how to use the application)



Favourite Parts

- Enjoyed creating menu options and defining user choices with loop/conditions, first feeling/sense that I was ‘programming’
- Researching and learning functions that were specific to an expense tracker application and learning how to implement them
- Applying packages to change the look and feel of the application (e.g. emojis, colorama, tabulate)
- Enjoyed modular aspect of coding, could work on one aspect of the application at a time, or multiple at a time, which gave some comfort knowing that if I made mistakes, it would not ruin the whole code (e.g. Isolated issue with viewing entries by category not affecting other aspects of the application)
- Overall confidence in creating my very own first Python application, and the confidence to build more



Thank you! ☺

Merry Christmas and a Happy New Year!