Expense CLI: A Command-Line Expense Management Program

Kyle Dormer (S1802423)

January 13, 2019

Contents

1 Introduction **1**

2 Design of System **2**
2.1 Design Paradigm 2
2.2 Modular Programming 3
2.3 Recursion 4
2.4 Algorithms for the Objectives 5
2.4.1 Control Flow 5
2.4.2 Enter Monthly Income 6
2.4.3 Set Overall Monthly Budget 8
2.4.4 Enter Expenses Based on Categories 8
2.4.5 Add New Categories 9
2.4.6 View Expense Report 9
2.4.7 Generate Graph of Expenses in PDF Format and Display Average and Total Expense for Category 10
2.5 Database 13

3 Testing the System **13**

A User Guide **14**

B Code **15**

1 Introduction

This piece of software is a command-line expense management tool. Its intended purpose is to allow the user to enter their income and their expenses and to have them stored and tracked for them. More specifically, the objectives of the software are that the user should be able to:

- Enter their monthly income, be it from single or from multiple sources.
- Set an overall monthly budget as well as a budget for categories of expense.
- Enter their expenses based on expense categories.
- Add new expense categories.
- View an exense report in terms of day/week/month/year and also with respect to a category.
- Generate and export a graph of their expenses in *PDF* format.
- The expense report should should inform the user when they are over/under budget for a specific category in a specific month.
- The expense report should also display the average expense per category.
- These objectives should be achieved using a backend *SQLite* database and with the *Pandas* and *Matplotlib* modules.

In retrospect to the development process, all of these objectives have been successfully implemented and achieved relatively smoothly and without issue. The biggest problems faced throughout the development of the software involved averaging the expenses of the user while generating the expense report. Furthermore, displaying the user's expenses in a nicely formatted fashion in a command-line environment proved difficult, as opposed to implementing a *GUI* solution. The format was solved by designing an algorithm that makes use of Python's concept of *list comprehension* in synergy with the inbuilt *any()* function.

2 Design of System

2.1 Design Paradigm

First and foremost, the design of the software is based upon the *procedural* programming paradigm, derived from the *imperative* paradigm. This can very much be reflected in the structure of the codebase. The program functions by the *__main__* module making a call to the *main()* function or *procedure*, which in turn make calls to other functions which also make calls to other functions. An example of this can be observed within *expcli.py*.

```
option_choice = get_user_option([])

if option_choice[0] == 1:
    get_income()

elif option_choice[0] == 2:
    monthly_budget = get_monthly_budget([])
    sql.store_monthly_budget(monthly_budget[0])

elif option_choice[0] == 3:
    expense = get_expense()
    sql.store_expense(expense)

elif option_choice[0] == 4:
    categories = get_categories([])
    sql.store_categories(categories)

elif option_choice[0] == 5:
    date = get_expense_date([])
    expenses = get_display_expenses('day', date[0], None)
    display_expenses(expenses)
```

Listing 1: An example of procedural program design from *expcli.py*

As shown in Listing 1, the program obtains the user's choice for which option to take. Depending on the user's choice, the *control flow* of the program is determined by making a function call to the function that corresponds with the user's choice. In fact, when combined with *recursion*, this is how the entirety of the program functions; the user enters their option, the corresponding function calls are made, and then the user enters another option *ad infinitum*.

The procedural paradigm was chosen in the design of this program due to its focus on using *variables*, *data structures* and *functions* to design and carry out algorithmic tasks. Whereas, for example, the *object-oriented* design paradigm focuses on conceptualising real world concepts as objects that contain both data and methods to interact with said data. In the context of this program, user expenses and input are stored as data structures, more specifically as lists of dictionaries. Therefore, using procedurally designed functions to interact with this data makes more sense. Furthermore, object-oriented programing is often less efficient than procedural programming[1], and with the relatively simple nature of the data involved in this program (expenses have only 4 properties), object-oriented was not deemed necessary nor favourable.

2.2 Modular Programming

In combination with the procedural paradigm, the technique of *modular programing* has also been focused upon throughout the design of this program. There are multiple reasons for this design choice.

Firstly, splitting the program into individual modules that each handle one area of functionality effectively *encapsulates* complexity. Therefore, once a module has been implemented and thoroughly tested, it can be reused by I or another programmer without knowing how it works. All the knowledge required to use it would be its purpose, its arguments (if it has any) and its output. An example of this can be observed many times throughout the codebase of the program, although most notably in the usage of the *exptools* module.

```
def get_monthly_budget(budget_var):

    monthly_budget = input('Enter monthly budget: ')

    if ex.validate_input(monthly_budget, 1, 18, str):
        try:
            monthly_budget = float(monthly_budget)
            budget_var.append(monthly_budget)

        except (Exception):
            ex.print_important('Invalid number, please try again!')
            get_monthly_budget(budget_var)
    else:
        ex.print_important('Invalid number, please try again!')
```

```
    get_monthly_budget(budget_var)

    return budget_var
```

Listing 2: An example of encapsulation achieved via modular programming in *expcli.py*

This function's purpose is to recursively collect a monthly budget from the user and makes use of the *validate_input()* and *print_important()* functions from the *exptools* module, although in this context its module object is defined as *ex* for shorthand use.

Secondly, as alluded to in the first point, modular programming allows for very effective *code reuse*. Indeed, this was in fact the original reasoning behind the creation of modular programming, first implemented in *ALGOL 68-R* in 1970.[2]. As can be seen in Listing 2, the *validate_input()* function can be called (reused) whenever needed.

Thirdly and finally, modular programming achieves *separation of concerns*. This means that each concern (or aspect of functionality) of the program belongs to a self-contained module. This ties in with the first point in that it allows the code of modules to change completely and still work perfectly as long as their expected inputs and outputs remain the same to interface properly with the rest of the program.

2.3 Recursion

Most of the functions, especially functions that gather user input, in this program shall be *recursive*. This involves a function calling itself to achieve a purpose. This is very useful in obtaining user input which must be validated for length or type. One objective of the program is to allow the user to enter their expenses. This will involve the expense's category, expense's date and the expense's amount; a string, string (or *datetime* object) and a float respectively. Furthermore, the strings must be between a minimum and maximum length. However, it is unlikely that the user will perfectly enter the desired input in a proper fashion each time. Therefore if validation of the user's input fails, the program should alert the user to the correct format of input and prompt them again. An algorithm for this could be implemented as pseudocode as such.

```
WHILE TRUE:
  GET input
  IF VALIDATE input FALSE:
```

```
GET input
```

Listing 3: An example pseudocode algorithm to inefficiently obtain valid user input

However, this algorithm is flawed in that the user may not enter the input in the correct format even after being prompted, and to deal with this the *if* block would have to keep nesting under itself *ad infinitum*. Therefore, instead of having an infinite loop and a potentially infinite control structure, the function could simply call itself again if the user does not enter the input correctly. This would be resource efficient as it would only be called as much as needed and would simplify and reduce the lines of code needed to be written. The pseudocode for this newly imagined algorithm can be expressed in Listing 4.

```
SUB get_input:
  GET input
  IF VALIDATE input TRUE:
    RETURN input
  ELSE:
    get_input
```

Listing 4: A correct recursive algorithm to obtain valid user input

An implementation of the Listing 4 algorithm can be observed in Listing 2. This algorithm will be repurposed throughout the program for whenever the user needs to input data (expenses, expense categories, income sources, and monthly budget).

2.4 Algorithms for the Objectives

The purpose of this section is to specify algorithms that achieve each objective set out in the specification.

2.4.1 Control Flow

The overall algorithm has been alluded to previously. It will display the options to the user and procedurally carry out the desired option and then recursively allow the user to enter another option (unless the user opts to exit the program).

The flowchart for this algorithm can be expressed as such.

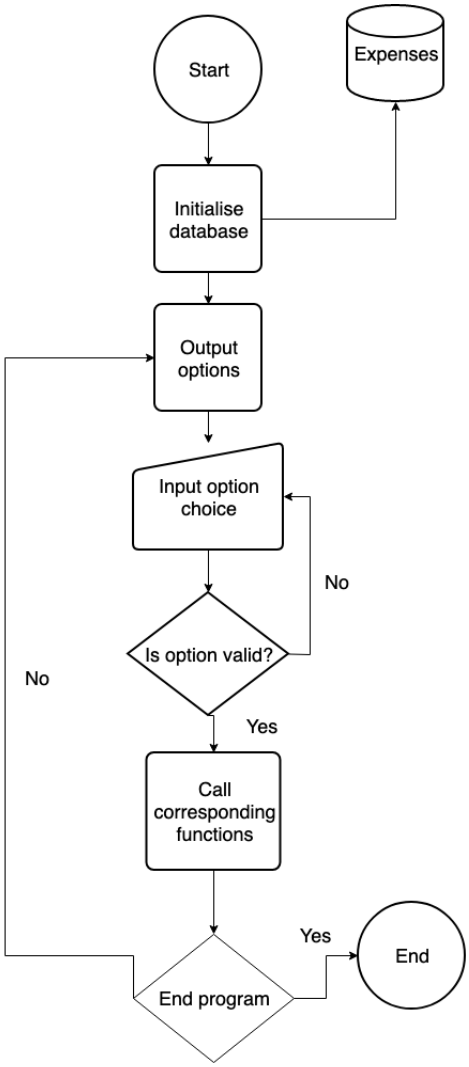

Figure 1: The overall control and execution flow of the program.

2.4.2 Enter Monthly Income

To obtain the user's monthly income, the algorithm will use recursion as discussed previously. Furthermore, each source of income will be stored in a data structure formed of a list of dictionaries. For instance, the user enters two income sources *'Salary'* and *'Side Job*, the former with an income of *£1000* and the latter with *£250*. Assuming the input is valid, the final data structure would be a list containing two dictionaries. The dictionary

would have a key for the name and a key for the income of the source. To store these in the database, a *foreach* loop can be used to loop through each dictionary in the list and gather the income. The algorithm can be expressed in flowchart format as such.

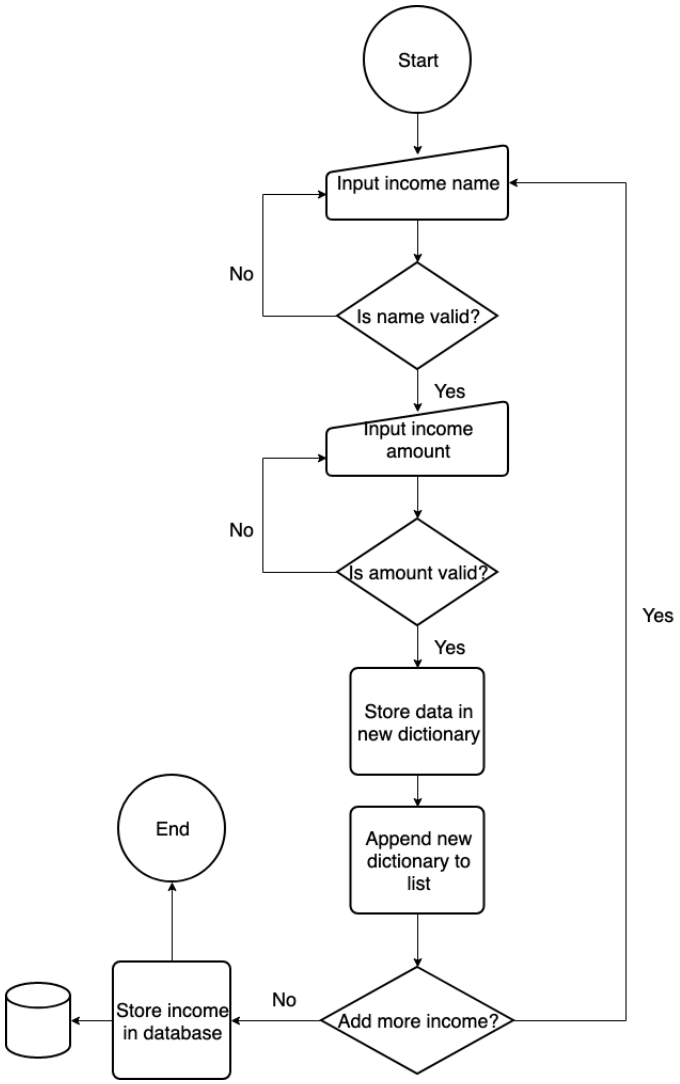

Figure 2: An algorithm to obtain and store the user's monthly income.

2.4.3 Set Overall Monthly Budget

Setting the overall monthly budget is a relatively simple process. Firstly, the program will take the user's desired budget as input using the same aforementioned recursive algorithm. Then, it will update the *Budget* field in the database for each month. This can be expressed in pseudocode.

```
SUB get_budget:
  INPUT budget
  IF VALIDATE budget TRUE:
    UPDATE Budget in Month WITH budget
  ELSE:
    get_budget
```

Listing 5: An algorithm to set the overall monthly budget.

2.4.4 Enter Expenses Based on Categories

To allow the user to enter expenses based on categories, the program will get all expense categories from the database and then allow them to choose which category the expense belongs to. It will also get the date of the expense and the amount. Then, the expense will be stored in a data structure consisting of a list of dictionaries, each dictionary containing the amount, date and category. The algorithm for this can be expressed in Python as such.

```
def get_expense():
  expense_category = get_category_choice([])
  expense_date = get_expense_date([])
  expense_amount = get_expense_amount([])

  if (expense_category and expense_date) and expense_amount:
      expense = {'category': expense_category[0],
                 'date': expense_date[0], 'amount': expense_amount[0]}

      return expense
```

Listing 6: An algorithm to collect expenses from the user.

The functions *get_category_choice()*, *get_expense_date()* and *get_expense_amount()* will all follow the recursive input algorithm detailed previously. For illustration, the code for *get_expense_amount()* can be shown as such.

```
def get_expense_amount(amount_arr):

    amount = input('Enter expense amount: ')

    if ex.validate_input(amount, 1, 12, str):
        try:
            amount = float(amount)
            amount_arr.append(amount)
        except (Exception):
            ex.print_important(
                'Invalid amount! Please enter a valid number!')
            get_expense_amount(amount_arr)
    else:
        ex.print_important('Invalid amount! Please enter a valid number!')
        get_expense_amount(amount_arr)

    return amount_arr
```

Listing 7: Source code for *get_expense_amount()*

2.4.5 Add New Categories

Adding new categories remains similar to the previous algorithms. The categories will be collected recursively from the user. Then, the user will have the option of inputting a monthly budget for the category. If yes, the budget is inputted and validated and if no, the budget is set to a Python signal object of *None*.

2.4.6 View Expense Report

To display an expense report to the user, the *Pandas* module is used to tabulate the data. The user has the option to see an expense report by day/week/month/year and category. For dates, there will be functions that will get the date specified and check which day/week/month/year it belongs to. Then, each expense that matches this day/week/month/year will be put into an expense array/list. Then, this expense list will be passed to a *Pandas* data frame. Then, using the *to_string()* method of a data frame object, the data will be sorted by category and displayed in table format to the user.

2.4.7 Generate Graph of Expenses in PDF Format and Display Average and Total Expense for Category

The program will use the *matplotlib* module to achieve this. The average and total expenses for each expense category will be fetched from the database. This will be done by pasing an array of all expenses to a function belonging to the *exptools* module. The source code for this function can be observed below in Listing 8. The algorithm takes an array of expense *tuples* as an argument. Then, it uses Python's *any()* function and an iterator to check if the expense category has already been added to the *categories* list. If it hasn't already been added, it is appended to the list in the form of a dictionary with keys for the category name, expense total, expense average and category budget. Once this operation has been completed, the categories list is looped through using a *foreach* loop. For each category in the categories list, the budget for the category is retrieved from the database and added to the corresponding dictionary key. Then, an expense counter variable is declared. All expenses in the expense array will then be iterated through and checked to see if they belong to the current category being looped through. If the expense indeed does belong to the category, the category's total expense becomes equal to its current value added to the new expense's amount and the expense count has one added to it. Then, the category's *mean* average expense is obtained by dividing the category's total expense by the number of expenses within that category, held within the expense count variable. Then, the category's total expense is checked to see if it is lower or higher than the category's set monthly budget. If the category has no budget, it is set to 'None' by handling the *exception* that occurs. Once this process has finished, the list of dictionaries is passed to a *Pandas* data frame object. This object can then be passed to a *matplotlib* bar chart and exported to PDF or it can be displayed in the expense report in tabular format to the user.

```
def get_average_expenses(expense_array):
categories = []

for expense in expense_array:
    if not any(category['Category'] == expense[2] for category in categories):
        categories.append(
            {'Category': expense[2], 'Total': 0, 'Average': 0, 'Budget': None})

for category in categories:
    expense_count = 0
    budget = sql.get_budget(category['Category'])

    for expense in expense_array:
        if expense[2] == category['Category']:
            category['Total'] += expense[3]
            expense_count += 1

    category['Average'] = category['Total'] / expense_count

    try:
        if category['Total'] > budget[0]:
            category['Budget'] = 'Over'
        else:
            category['Budget'] = 'Under'
    except (TypeError):
        category['Budget'] = 'None'
```

```
        expense_count = 0

    return categories
```

Listing 8: Source code for *get_average_expenses()*.

2.5 Database

The design of the database can be observed in the following entity-relationship diagram.

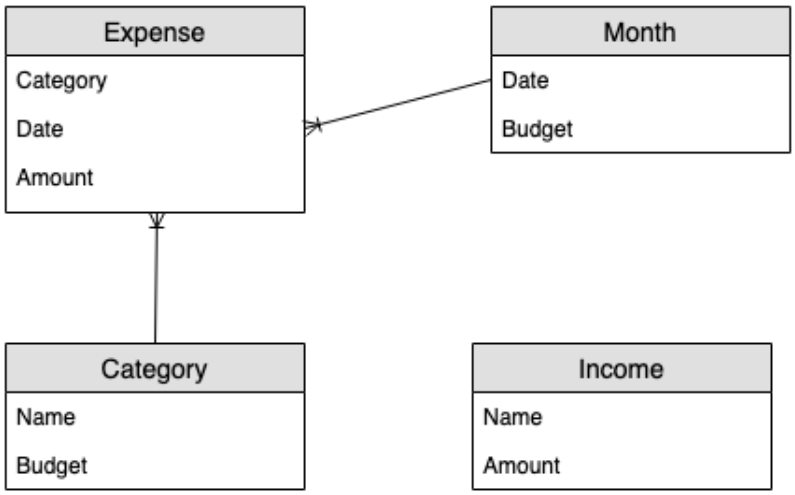

Figure 3: Entity relationsip diagram for the proposed database.

All operations involving the database will be contained in a module named *expsql.py*.

3 Testing the System

The testing process used throughout the development of the system has been *black-box testing*. This involves ignoring the inner complexities of the program and instead focusing on its outputs and features. For instance, instead of individually unit testing each and every function, they are tested as a procedural whole. To elaborate, when testing the functionality of inputting and storing an expense, there are 5 functions involved in this process, but instead the testing takes place by inputting the data (and incorrect data) and seeing what the program outputs. The expected output in this case is a message stating that the expense has been successfully stored within the database and then seeing the expense stored within the database. If inputted data isn't in the correct format, the validation capabiliities of the program to handle invalid data are tested. All tests took place by putting an expected output against the actual output.

A User Guide

NOTE: When entering dates in use of the program, the format must be YYYY-MM-DD. This is stated within the program. **NOTE:** Before performing operations such as exporting expenses and entering expenses, ensure you have entered income and expense categories first. The program will handle not doing this, but works properly if done so.

Starting the Program To start the program run the command *python3 __main__.py*.

Enter Monthly Income To enter monthly income, simply enter option 1 in the menu when prompted and follow the instructions.

Set Monthly Budget To set the overall monthly budget, simply enter option 2 in the menu when prompted and follow the instructions.

Enter Expense To enter an expense, simply enter option 3 in the menu when prompted and follow the instructions.

Add Expense Categories To add an expense categorym simply enter option 4 in the menu when prompted and follow the instructions.

View Expense Report for Day To view the expense report for a specific day, simply enter option 5 in the menu and follow the instructions. Ensure your inputted date conforms to the format detailed above.

View Expense Report for Week To view the expense report for a specific week, simply enter option 6 in the menu and follow the instructions. Ensure your inputted date conforms to the format detailed above.

View Expense Report for Month To view the expense report for a specific month, simply enter option 7 in the menu and follow the instructions. Ensure your inputted date conforms to the format detailed above.

View Expense Report for Year To view the expense report for a specific year, simply enter option 8 in the menu and follow the instructions. Ensure your inputted date conforms to the format detailed above.

View Expense Report for Category To view the expense report for a specific category, simply select option 9 in the menu when prompted and follow the instructions.

Export Expense Graph to PDF To export the expenses to a graph format in a PDF, simply select option 10 in the menu when prompted and ensure there are actually expenses already in the database.

Export Expense Graph to CSV To export the expenses to a CSV excel format, simply select option 11 in the menu when prompted and follow the onscreen instructions. As before, ensure that there are actually expenses already in the database.

Exit the Program To exit the program, simply enter option 12 in the menu when prompted.

B Code

The code for this program spans across four files.

Listing 9: Source code for *__main__.py*.

```
    """
Author: Kyle Dormer
Date: 27/10/2019
Student Number: s1802423
"""

import atexit

from expcli import exit_handler, main
atexit.register(exit_handler)

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

Listing 10: Source code for *expcli.py*.

```
    """
Author: Kyle Dormer
```

```
Date: 27/10/2019
Student Number: s1802423
"""

import sys
import numpy as np
import pandas as pd
from datetime import datetime
import exptools as ex
import expsql as sql


def main():
    """
    Main function to be called from __main__.
    """
    sql.db_init()
    display_options()


def exit_handler():
    """
    Closes database operation and outputs a farewell message to the user.
    Called automatically on program exit.
    """
    sql.db_close()
    ex.print_important('Have a great day!')


def get_income():
    """
    Recursively get sources of income from the user and store them in the
    """
    income_sources = get_income_sources([])
    monthly_income_total = 0

    for income_source in income_sources:
        monthly_income_total += income_source['source_income']

    try:
```

```
            sql.store_income_sources(income_sources)
            print('Income sources stored successfully!')
        except (Exception):
            print('There was an error while storing your income. Please try ag


def get_income_sources(income_sources_list):
    """
    Recursively collects sources of income from the user and returns them
    list of dictionaries.
    """
    source_name = input('What is the name of the income source? ')

    if ex.validate_input(source_name, 1, 12, str):
        source_income = input('What is the monthly income of the source? '

        try:
            source_income = float(source_income)
            income_sources_list.append({'source_name': source_name,
                                        'source_income': source_income})
            repeat = input('Enter another input source? [Y/n]\n')

            if repeat.lower() == 'y':
                get_income_sources(income_sources_list)

        except (Exception):
            print('Invalid input! That\'s not a valid number!')
            get_income_sources(income_sources_list)

    else:
        print('Invalid input! Your name is too big or too small!')
        get_income_sources(income_sources_list)

    return income_sources_list


def get_categories(categories_list):
    """
    Recursively collects categories of expense from the user and returns th
    in a list of dictionaries.
```

```
        """
        category_name = input('What is the name of the expense category? ')

        if ex.validate_input(category_name, 1, 16, str):
            hasBudget = input('Does this category have a set budget? [Y/n]\n')

            if hasBudget.lower() == 'y':
                category_budget = input('Budget for ' + category_name + ': ')

                try:
                    category_budget = float(category_budget)
                    categories_list.append({'category_name': category_name,
                                            'category_budget': category_budget

                    repeat = input('Enter another expense category? [Y/n]\n')

                    if repeat.lower() == 'y':
                        get_categories(categories_list)

                except (Exception):
                    print('Invalid input! That\'s not a valid number!')
                    get_categories(categories_list)

            else:
                categories_list.append({'category_name': category_name,
                                        'category_budget': None})

                repeat = input('Enter another expense category? [Y/n]\n')

                if repeat.lower() == 'y':
                    get_categories(categories_list)

        else:
            print('Invalid input! Your category name is too big or too small!'
            get_categories(categories_list)

        return categories_list


def get_monthly_budget(budget_var):
```

```
    """
    Recursively get the monthly budget from the user and return it in an a
    """
    monthly_budget = input('Enter monthly budget: ')

    if ex.validate_input(monthly_budget, 1, 18, str):
        try:
            monthly_budget = float(monthly_budget)
            budget_var.append(monthly_budget)

        except (Exception):
            ex.print_important('Invalid number, please try again!')
            get_monthly_budget(budget_var)
    else:
        ex.print_important('Invalid number, please try again!')
        get_monthly_budget(budget_var)

    return budget_var


def display_options():
    """
    Display all possible options to the user, allow them to choose their d
    """

    options = ['1. Enter monthly income', '2. Set monthly budget', '3. Ent
               '6. View expense report for week', '7. View expense report

    for option in options:
        print(option)

    option_choice = get_user_option([])

    if option_choice[0] == 1:
        get_income()

    elif option_choice[0] == 2:
        monthly_budget = get_monthly_budget([])
        sql.store_monthly_budget(monthly_budget[0])
```

```
elif option_choice[0] == 3:
    expense = get_expense()
    sql.store_expense(expense)

elif option_choice[0] == 4:
    categories = get_categories([])
    sql.store_categories(categories)

elif option_choice[0] == 5:
    date = get_expense_date([])
    expenses = get_display_expenses('day', date[0], None)
    display_expenses(expenses)

elif option_choice[0] == 6:
    date = get_expense_date([])
    expenses = get_display_expenses('week', date[0], None)
    display_expenses(expenses)

elif option_choice[0] == 7:
    date = get_expense_date([])
    expenses = get_display_expenses('month', date[0], None)
    display_expenses(expenses)

elif option_choice[0] == 8:
    date = get_expense_date([])
    expenses = get_display_expenses('year', date[0], None)
    display_expenses(expenses)

elif option_choice[0] == 9:
    date = get_expense_date([])
    category = get_category_choice([])
    expenses = get_display_expenses('category', date[0], category[0])
    display_expenses(expenses)

elif option_choice[0] == 10:
    ex.export_pdf()

elif option_choice[0] == 11:
    ex.export_csv()
```

```
    elif option_choice[0] == 12:
        sys.exit()

    display_options()


def get_user_option(option_var):
    """
    Get option choice from the user and validate input.
    """
    option = input('Choose an option: ')

    try:
        option = int(option)

        if option in range(1, 13):
            option_var.append(option)

        else:
            ex.print_important(
                'Invalid option! Please enter the number that corresponds
            get_user_option(option_var)

    except (Exception):
        ex.print_important(
            'Invalid option! Please enter the number that corresponds with
        get_user_option(option_var)

    return option_var


def get_category_choice(choice_var):
    """
    Get category of expense from user input.
    """
    categories = sql.get_categories()
    category_number = len(categories)
    index = 1

    if category_number > 0:
```

```
        for category in categories:
            print(str(index) + '. ' + category)
            index += 1

        category_choice = input('Choose an expense category: ')

        if ex.validate_input(category_choice, 1, 18, str):
            try:
                category_choice = int(category_choice)

                if category_choice in range(1, category_number + 1):
                    choice_var.append(categories[category_choice - 1])
                else:
                    ex.print_important(
                        'Invalid category choice! Please enter a valid cate
                    get_category_choice(choice_var)

            except (Exception):
                ex.print_important(
                    'Invalid category choice! Please enter a valid category
                get_category_choice(choice_var)
        else:
            ex.print_important(
                'Invalid category choice! Please enter a valid category!')
            get_category_choice(choice_var)
    else:
        choice_var = None
        ex.print_important(
            'There are currently no expense categories! Please add some be

    return choice_var


def get_expense_date(date_arr):
    """
    Get and validate date of expense from user input.
    """
    print('Date format should be YYYY-MM-DD. Leave blank for the current da
    print('If entering a month or year, enter a date from the same month o
    date = input('Enter expense date: ')
```

```
    if ex.validate_date(date):
        date_arr.append(date)
    elif len(date) == 0:
        date = datetime.now().strftime('%Y-%m-%d')
        date_arr.append(date)
    else:
        ex.print_important('Incorrect date! Please ensure format is correct
        get_expense_date(date_arr)

    return date_arr


def get_expense_amount(amount_arr):
    """
    Get and validate expense amount from user input.
    """
    amount = input('Enter expense amount: ')

    if ex.validate_input(amount, 1, 12, str):
        try:
            amount = float(amount)
            amount_arr.append(amount)
        except (Exception):
            ex.print_important(
                'Invalid amount! Please enter a valid number!')
            get_expense_amount(amount_arr)
    else:
        ex.print_important('Invalid amount! Please enter a valid number!')
        get_expense_amount(amount_arr)

    return amount_arr


def get_expense():
    """
    Get expense to be stored from user input.
    """
    expense_category = get_category_choice([])
    expense_date = get_expense_date([])
```

```
    expense_amount = get_expense_amount([])

    if (expense_category and expense_date) and expense_amount:
        expense = {'category': expense_category[0],
                   'date': expense_date[0], 'amount': expense_amount[0]}

        return expense


def get_display_expenses(timeframe, date, category):
    expenses = sql.get_expenses(None)
    display_expenses = []

    if timeframe == 'day':
        for expense in expenses:
            if expense[1] == date:
                display_expenses.append(expense)
    elif timeframe == 'week':
        week = ex.get_week(date)

        for expense in expenses:
            if ex.get_week(expense[1]) == week:
                display_expenses.append(expense)

    elif timeframe == 'year':
        year = ex.get_year(date)

        for expense in expenses:
            if ex.get_year(expense[1]) == year:
                display_expenses.append(expense)

    elif timeframe == 'month':
        month = ex.get_month(date)

        for expense in expenses:
            if ex.get_month(expense[1]) == month:
                display_expenses.append(expense)

    elif timeframe == 'category':
        for expense in expenses:
```

```
            if expense[2] == category:
                display_expenses.append(expense)
    else:
        ex.print_important('Invalid timeframe given! Please try again!')

    return display_expenses


def display_expenses(expense_array):
    pd.set_option('colheader_justify', 'center')
    data_frame = pd.DataFrame(expense_array,
                              columns=['ID', 'Date', 'Category', 'Amount'])

    sorted_frame = data_frame.sort_values(by='Category')

    if not data_frame.empty:
        ex.print_important(sorted_frame.to_string(index=False))
        display_average_expenses(expense_array)
    else:
        ex.print_important('No expenses found for that date!')


def display_average_expenses(expense_array):
    average_expenses = ex.get_average_expenses(expense_array)

    pd.set_option('colheader_justify', 'center')
    data_frame = pd.DataFrame(average_expenses)

    print('-----------------------------------------')
    print(data_frame.to_string(index=False))
```

Listing 11: Source code for *exptools.py*.

```
    """
Author: Kyle Dormer
Date: 27/10/2019
Student Number: s1802423
"""

from datetime import datetime
```

```
import pandas as pd
import expsql as sql
import matplotlib as plt


def validate_input(user_input, lower_length, upper_length, desired_type):
    """
    Used for validating user input. It takes the user's input, the lowest
    acceptable length, the highest acceptable length and the desired type
    """
    return ((len(user_input) >= lower_length) and
            (len(user_input) <= upper_length)) and \
        type(user_input) == desired_type


def validate_date(date_string):
    """
    Validate data string passed as argument to ensure user inputted data c
    """
    date_format = '%Y-%m-%d'

    try:
        datetime_object = datetime.strptime(date_string, date_format)

        if datetime_object:
            return True

    except (ValueError):
        return False


def get_week(date_string):
    if validate_date(date_string):
        try:
            datetime_object = datetime.strptime(date_string, '%Y-%m-%d')
            return datetime_object.isocalendar()[1]
        except (TypeError):
            return False
```

```
def get_month(date_string):
    if validate_date(date_string):
        try:
            datetime_object = datetime.strptime(date_string, '%Y-%m-%d')
            return datetime_object.month
        except (TypeError):
            return False


def get_year(date_string):
    if validate_date(date_string):
        try:
            datetime_object = datetime.strptime(date_string, '%Y-%m-%d')
            return datetime_object.isocalendar()[0]
        except (TypeError):
            return False


def get_average_expenses(expense_array):
    categories = []

    for expense in expense_array:
        if not any(category['Category'] == expense[2] for category in cate
            categories.append(
                {'Category': expense[2], 'Total': 0, 'Average': 0, 'Budget

    for category in categories:
        expense_count = 0
        budget = sql.get_budget(category['Category'])

        for expense in expense_array:
            if expense[2] == category['Category']:
                category['Total'] += expense[3]
                expense_count += 1

        category['Average'] = category['Total'] / expense_count

        try:
            if category['Total'] > budget[0]:
                category['Budget'] = 'Over'
```

```
            else:
                category['Budget'] = 'Under'
        except (TypeError):
            category['Budget'] = 'None'

        expense_count = 0

    return categories


def export_csv():
    expenses = sql.get_expenses(None)

    if expenses:
        try:
            data_frame = pd.DataFrame(expenses)
            data_frame.to_csv('expenses.csv')
            print_important('Expenses exported successfully!')
        except (Exception):
            print_important(
                'There was an error exporting your expenses to a CSV file!
    else:
        print_important(
            'There was an error exporting your expenses to a CSV file! Ple


def export_pdf():
    expenses = sql.get_expenses(None)

    if expenses:
        try:

            average_expenses = get_average_expenses(expenses)

            data_frame = pd.DataFrame(average_expenses)
            data_frame.set_index('Category', drop=True, inplace=True)

            plot = data_frame.plot(kind='bar')
            plt.pyplot.tight_layout()
            plt.pyplot.savefig('expenses.pdf')
```

```
            print_important('Your expenses were successfully exported to Pl
        except (Exception):
            print_important(
                'There was an error exporting your expenses! Please ensure
    else:
        print_important(
            'There was an error exporting your expenses! Please ensure you


def print_important(string):
    print('\n')
    print(string)
    print('\n')
```

Listing 12: Source code for *expsql.py*.

```
    """
Author: Kyle Dormer
Date: 05/11/2019
Student Number: s1802423
"""

import sqlite3
import exptools as ex
from datetime import date as d
from datetime import datetime

connection = sqlite3.connect('expenses.db')
cursor = connection.cursor()


def db_init():
    """
    Initialise the database for the application by creating the tables and
    """
    cursor.execute('CREATE TABLE IF NOT EXISTS Expense(ID INTEGER NOT NULL
                   KEY AUTOINCREMENT UNIQUE
                   Date TEXT, Category TEXT
                   Amount REAL)')
```

```
    cursor.execute('CREATE TABLE IF NOT EXISTS Month(Date TEXT NOT NULL \
            PRIMARY KEY UNIQUE, Budget REAL)')

    cursor.execute('CREATE TABLE IF NOT EXISTS Category(Name TEXT NOT NULL
    KEY UNIQUE, Budget REAL)')

    cursor.execute(
        'CREATE TABLE IF NOT EXISTS Income(Name TEXT NOT NULL PRIMARY KEY U

    populate_months()


def db_close():
    """
    Close database connection.
    """
    cursor.close()
    connection.close()


def store_expense(expense):
    """
    Store the expense given as an argument in the database.
    """
    try:
        cursor.execute('INSERT INTO Expense(Date, Category, Amount) VALUES
                       (expense['date'], expense['category'], expense['amo
        ex.print_important('Expense stored successfully!')
        connection.commit()
    except (Exception):
        ex.print_important(
            'There was an error storing your expense! Please try again!')


def get_expense(ID):
    """
    Get the expense with the corresponding ID from the database and return
    """
    cursor.execute('SELECT * FROM Expense WHERE ID=?', (ID,))
```

```
    expense = cursor.fetchone()
    return expense


def get_expenses(category):
    """
    Get all expenses from the database and return them.
    """
    if category:
        cursor.execute(
            'SELECT * FROM Expense WHERE Category = (?)', (category,))
    elif category is None:
        cursor.execute('SELECT * FROM Expense')

    return cursor.fetchall()


def store_income_source(income_source):
    """
    Store the source of income passed as an argument in the database.
    """
    cursor.execute('INSERT INTO Income(Name, Amount) VALUES (?, ?)',
                   (income_source['source_name'], income_source['source_inc

    connection.commit()


def store_income_sources(source_array):
    """
    Store each income source in the source array passed in the database.
    """
    for source in source_array:
        cursor.execute('INSERT INTO Income(Name, Amount) VALUES (?, ?)',
                       (source['source_name'], source['source_income']))

    connection.commit()


def populate_months():
    """
```

```
    Populate the Month table of the database with all months for 2019 and
    """
    cursor.execute('SELECT * FROM Month')

    if cursor.fetchone() == None:
        months = ['01', '02', '03', '04', '05',
                  '06', '07', '08', '09', '10', '11', '12']
        years = ['2019', '2020']

        for year in years:
            for month in months:
                date_stamp = year + '-' + month
                cursor.execute(
                    'INSERT INTO Month(Date) VALUES (?)', (date_stamp,))

        connection.commit()


def store_monthly_budget(budget):
    """
    Set the monthly budget for each month in the database. Expected type i
    """
    try:
        cursor.execute('UPDATE Month SET Budget = (?)', (budget,))
        connection.commit()
        ex.print_important(
            'Your new monthly budget was stored successfully!')
    except (Exception):
        ex.print_important(
            'There was an error storing your new monthly budget. Please try


def store_categories(categories_array):
    """
    Store each category in the categories_array argument in the database.
    """
    try:
        for category in categories_array:
            cursor.execute('INSERT INTO Category(Name, Budget) VALUES (?,
                           (category['category_name'], category['category_b
```

```
        connection.commit()
        ex.print_important(
            'Your expense categories have been stored successfully!')
    except (Exception):
        ex.print_important(
            'There was an error storing your expense categories. Please try


def get_categories():
    """
    Get all expense categories from the database and return them.
    """
    cursor.execute('SELECT * FROM Category')

    categories = []

    for category in cursor.fetchall():
        categories.append(category[0])

    return categories


def get_budget(category):
    try:
        cursor.execute(
            'SELECT Budget FROM Category WHERE Name = (?)', (category,))
        budget = cursor.fetchone()
        return budget
    except (Exception):
        ex.print_important(
            'There was an error fetching the monthly budget! Please try ag
```

References

[1] Cardelli, Luca (1996), *Bad Engineering Properties of Object-Oriented Languages*, Digital Equipment Corporation Systems Research Center Sourced from: `http://lucacardelli.name/Papers/BadPropertiesOfOO.html`, Accessed: [22nd January 2020].

[2] C. H. Lindsey (University of Manchester) & H. J. Boom (Mathematisch Centrum, Amsterdam), *A Modules and Separate Compilation Facility for ALGOL 68*, Sourced from: `http://archive.computerhistory.org/resources/text/algol/ACM_Algol_bulletin/1061719/p19-lindsey.pdf`, Accessed: [22nd January 2020].