Expense CLI: A Command-Line Expense Management Program

Kyle Dormer (S1802423)

January 13, 2019

Contents

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 corre-

sponding 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.

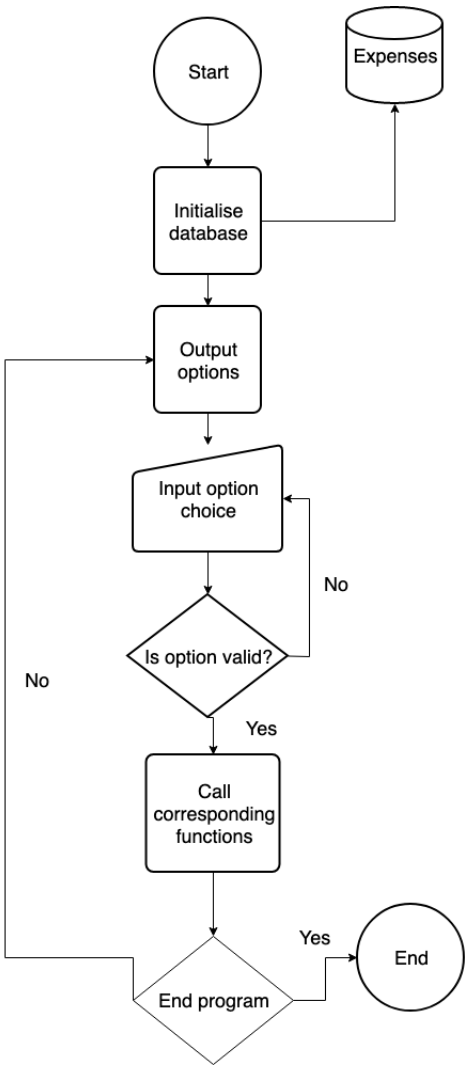

Figure 1: The overall control and execution flow of the program.

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.

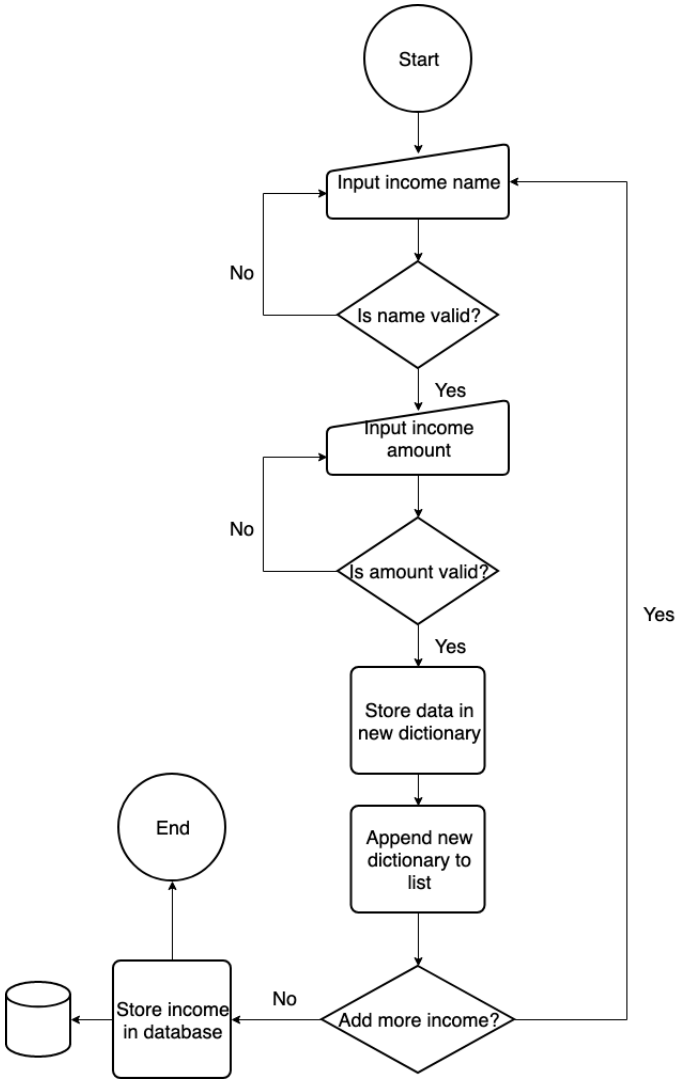

Figure 2: An algorithm to obtain and store the user's monthly income.

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()*.

3 Testing the System

4 Demonstrating the System

A User Guide

B Code

C Test Suites

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].