11/25/2022

Final Assignment 1

Finite State Machines

Joris Heemskerk
V2A

Table of Contents

1. Introduction

In this document I will go over my process, findings, and results pertaining to the first final assignment of Simulation-Programming. As I found the information of the assignment on Canvas not to match with the requests the teacher made, I will therefore handle my deliverables separately.

In the first half of this document I will go through the library I set up to make finite state machines. Said library will be demonstrated too, with some simple examples.

In the latter half of the document I will go through the assignment as made by my teacher. This section will expand upon the more basic library I constructed. It will implement the ability to provide manual inputs to the final state machine. On top of that, it will also store data outside of the states. I will elaborate on this later on in this document.

At the end of the document I will reflect upon both assignments, as given, and go over what I think I learned from both within the context of goals of the Simulation class. I hope this expansive document will be helpful to the teacher with regard to future assignments. As I found the mismatch in information on the Canvas page with what the teacher requested incredibly unhelpful. I think it shows in my, in my own opinion, rather lacklustre deliverables.

2. Finite state machine library

This chapter will go over the design, code, documentation, and an example from my simple, yet complete, finite state machine library

2.1. Design

Having previously worked with problems that can be expressed as graphs, e.g. shortest path algorithms like Dijkstra, I felt as though the finite state machine could be represented in a solution just like those. For that reason I decided to go with an object oriented implementation that contains nested states, instead of nodes that contain connections.

Do demonstrate my findings, I have designed the following simple class diagram for my State class:

State
- __state_list: np.ndarray[State] - __FINAL: const bool
+__init__(alphabet_size: int, final: bool): State + set_state_list(state_list: np.ndarray[State]): None + run(input: np.ndarray[int]): State + get_final(): const bool

Table 1: Class diagram State class

As you can see, this class is rather simple. Yet it performs everything one might need. I want to draw attention to the following matters:

- Private member variables

I have decided to enforce private member variables. This is not usually supported in Python, but can be semi done in accordance with PEP 8 by putting two underscores before a member variable.

- Constants

For constant variables like __FINAL, a variable that only needs to be set once and it contains a Boolean that decides if the state is final, I have also followed PEP 8. This suggest the use of all capitalised letters in constants. This, combined with the fact that this variable is private, results in the variable __FINAL.

- __state_list

To make seamless transitions from state to state based on any given condition, efficiency is of vital importance. To make this possible, a list of references to destination states is saved inside of the __state_list variable. One can transition from state to state by using the alphabet as restricted by the size of the __state_list variable. E.g. given a state with a __state_list that contains two references, if one gives the input of 0, the state on the first index in __state_list will be one to be returned.

2.2. Code

As this document only needs to contain the code necessary for the teacher to understand my workings. I will only elaborate on my most complex function; run().

```
def run(self, input: np.ndarray[int])-> State:
        if len(input) == 0:
            return self

        destination = self.__state_list[input[0]]
        if not destination:
            return self.run(input[1:])
        return destination.run(input[1:])
```

Figure 1: State::run

The above function is deliberately displayed without code documentation, to keep it concise. For further documentation I would refer to chapter *2.3. Documentation* or to the code itself.

As seen above, the run function is a recursive implementation that runs through the list of inputs, passing reduced lists to the corresponding nodes that are next in line. If a transition is not defined, the state will repeat itself, but with a new input. If no input is left, the state itself is returned.
In all other scenarios, the correct state will be called. Except in the case of an out of bounds error. This occurs when one of the inputs is not possible within the diagram. This error will not be checked for, as Python catches this more than well enough and an extra check or try-catch statement would heavily impact the relative runtime.

2.3. Documentation

In accordance with PEP 257, I have documented my code using docstrings. Once called, with *help(State)*, it will print the following:

```
class State(builtins.object)
 |  State(alphabet_size: int, final: bool) -> state.State
 |
 |  This class implements a state as seen in finite state machines.
 |  For more information on finite state machines see: https://en.wikipedia.org/wiki/Finite-
state_machine
 |
 |  Methods defined here:
 |
 |  __init__(self, alphabet_size: int, final: bool) -> state.State
 |      This initializer will initialise a State
 |      Upon doing so it will set two private variables: __state_list and __FINAL.
 |
 |      __state_list is the private member variable that contains the destination states.
 |      This list will be constructed as an empty numpy array of type State.
 |      The __state_list can be set with actual values using the set_state_list() setter.
 |
 |      __FINAL is a constant private member variable that represents whether or not the State is final.
 |
 |      @params
```

```
 |      alphabet_size (int)  : The size of the alphabet used for the State.
 |      final        (bool) : Boolean representing if the State is a final state.
 |
 |      @return
 |      Constructed State
 |
 |  get_final(self) -> bool
 |      This getter will check and return if the State is a final state.
 |
 |      @return
 |      boolean representing if the State is a final state.
 |
 |  run(self, input: numpy.ndarray[int]) -> state.State
 |      This method will run the current State.
 |      It will find the corresponding destination State for first item in the given input.
 |      If the input is empty, this is the final state and will therefore be returned.
 |      If the corresponding destination is not defined, it will be assumed the path runs to itself.
 |      It will therefore call itself with the rest of the input.
 |
 |      @params
 |      input (np.ndarray[int]): A numpy ndarray of input integers, each one corresponding (by index)
to a destination for any
given State.
 |
 |      @return
 |      The state corresponding to the output path of the entire input string.
 |
 |  set_state_list(self, state_list: numpy.ndarray[state.State]) -> None
 |      This setter will set the state_list of the State
 |      Since the __state_list is a private variable, it will have to be set with this setter.
 |      This is done because it is of vital importance that the state_list is correct and corresponds to
destination by index.
 |      As such, a copy will be made to assure the user does not alter the __state_list outside of the
class.
 |      This does of course not fully prevent altering of the class, but it should help to prevent users
from wrongdoing.
 |
 |      @params
 |      state_list (np.ndarray[State]): A numpy ndarray of States.
 |       Each of these states corresponds (by index) to the destination for any given input.
 |       E.g. state_list [q1, q3] means that the state will switch to q1 for input 0 and q3 for input 1.
 |
 |  ----------------------------------------------------------------------
```

2.4. Example

To demonstrate the workings of this state, I have implemented a finite state machine from one of the homework assignments. The following finite state machine accepts any given string of inputs with an odd amount of a's and an even amount of b's.

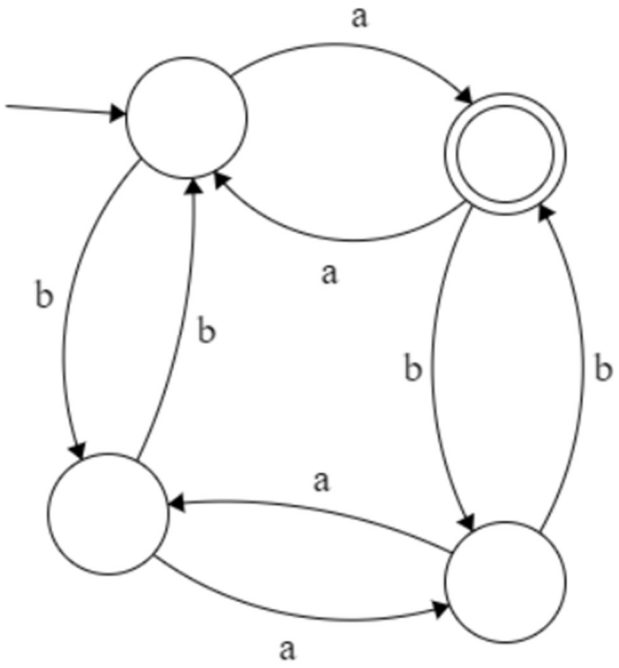

Figure 2: FMS that accepts odd a's and even b's

To implement this finite state machine, I used the following function:

```
def odd_a_even_b(input)-> bool:
    alphabet_size = 2
    q0 = State(alphabet_size, False)
    q1 = State(alphabet_size, True)
    q2 = State(alphabet_size, False)
    q3 = State(alphabet_size, False)

    q0.set_state_list(np.array([q1,q2]))
    q1.set_state_list(np.array([q0,q3]))
    q2.set_state_list(np.array([q3,q0]))
    q3.set_state_list(np.array([q2,q1]))

    destination = q0.run(input)
    return destination.get_final()
```

Figure 3: FMS odd a's, even b's in code

As seen in the example, an alphabet of 2 is accounted for for all the states. As well as a Boolean representing if the state is final. From there, the input is run from *q0*. The function returns if the destination state is final or not.

One can call and print this function using, for example:

```
print(odd_a_even_b(np.array([1,0,0,0,1])))
```

3. Manual input

As per my teachers requests, I expanded upon my previously mentioned library by supporting manual inputs. This means that instead of having the user input a string of characters, the user can put each character in one by one.

3.1. Design

To provide these options without adding unnecessary additions to my previous library, I decided to construct a child class for my State class. The new diagram looks as follows:

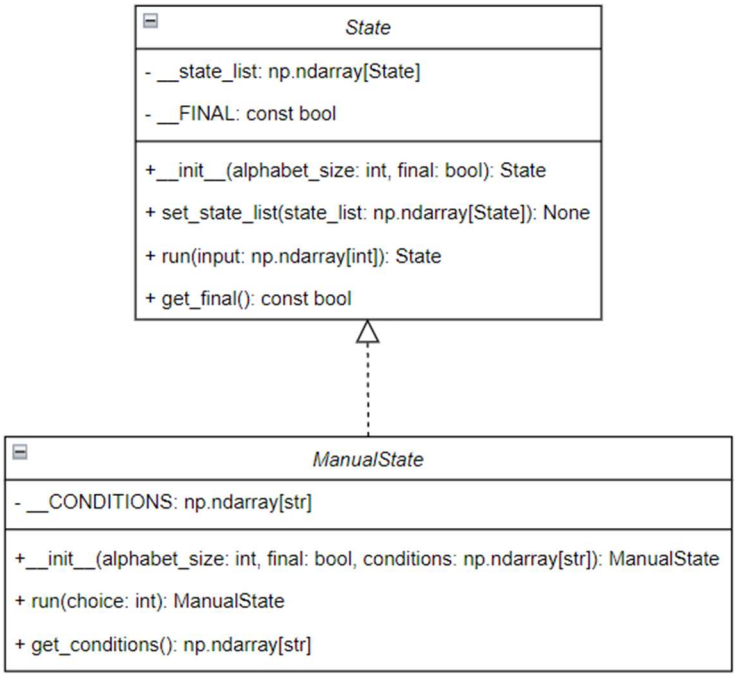

Table 2: Class diagram ManualState

As seen above, the new ManualState only overrides the State::run method to not provide a list of inputs, but only one. This simplifies the function and improves performance. It could have been done using the State::run() method, but this would add needles runtime complexity. A simple override would seem to be the best solution.

The ManualState also introduces a list of strings that form the conditions. As this class is there to provide manual inputs, the users that use this class inherently want the user to make transition decisions. To aid said users in their decisions, the conditions for each transition can be saved in the private constant __CONDITIONS variable. The class itself does not use these conditions, as the method of output and user input would preferably not be set in stone. These implementations will be left to the finite state machines created using the library.

3.2. Documentation

As before, this class is also documented using the PEP 257 standards of docstrings. To prevent the reader of this document from spitting through my code, I will again show the documentation below as provided by the help method integrated in Python.

```
class ManualState(state.State)
 |  ManualState(alphabet_size: int, final: bool, conditions: numpy.ndarray[str]) ->
manual_state.ManuelState
 |
 |  This class implements a manual version of the State class.
 |  It inherits from State and overrides the run function to only run for one character.
 |  On top of that, it also is able to store conditions. These will not be used, but can be easily
accessible once set.
 |
 |  Method resolution order:
 |      ManualState
 |      state.State
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __init__(self, alphabet_size: int, final: bool, conditions: numpy.ndarray[str]) ->
manual_state.ManuelState
 |      This initializer will set the appropriate State. On top of that it will save the conditions for this
ManualState.
 |      None of these variables are inherently accessible by calling the object. Getters are provided for
the conditions and final.
 |      To set the actual state_list use the set_state_list() method from State.
 |
 |      @params
 |      alphabet_size (int)                    : The size of the alphabet used for the ManualState.
 |      final        (bool)                    : Boolean representing if the ManualState is a final state.
 |      conditions   (np.ndarray[str]) : np.ndarray containing strings that pertain to the reasons for any
given transition
 |                                  (corresponding to __state_list by index).
 |
 |      @return
 |      Constructed ManualState
 |
 |  get_conditions(self) -> numpy.ndarray[str]
 |      This getter will return a copy of the provided conditions.
 |      This is done by copy to prevent the user from accidentally altering the data after they were set.
 |      As this is decidedly not intended behavior.
 |
 |      @return
 |      boolean representing if the State is a final state.
 |
 |  run(self, choice: int) -> manual_state.ManuelState
 |      This run method overrides the run from State.
```

```
 |      It will not check if the input is valid, therefore it can crash if the implementation does not catch
these errors.
 |      This Class is meant to be used when the implementation walks a user through a menu to decide
on their input.
 |      This Class is meant to be efficient, fast and lightweight. The implementation has to catch the
errors to ensure this.
 |
 |      @params
 |      choice (int): Index of the desired transition as in the __state_list.
 |
 |      @return
 |      - self if no destination was set for given index.
 |      - Error if invalid index was given.
 |      - Correct destination if all went well.
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from state.State:
 |
 |  get_final(self) -> bool
 |      This getter will check and return if the State is a final state.
 |
 |      @return
 |      boolean representing if the State is a final state.
 |
 |  set_state_list(self, state_list: numpy.ndarray[state.State]) -> None
 |      This setter will set the state_list of the State
 |      Since the __state_list is a private variable, it will have to be set with this setter.
 |      This is done because it is of vital importance that the state_list is correct and corresponds to
destination by index.
 |      As such, a copy will be made to assure the user does not alter the __state_list outside of the
class.
 |      This does of course not fully prevent altering of the class, but it should help to prevent users
from wrongdoing.
 |
 |      @params
 |      state_list (np.ndarray[State]): A numpy ndarray of States.
 |       Each of these states corresponds (by index) to the destination for any given input.
 |       E.g. state_list [q1, q3] means that the state will switch to q1 for input 0 and q3 for input 1.
 |
 |  ----------------------------------------------------------------------
```

Examples of this class in action will be put in chapter 4.

4. Finite state machine implementation – Coffee machine

Now for the second part of the actual assignment I will implement a more complicated finite state machine. As per the instructions I conferred with my teacher on my potential choices. I came to the coffee machine as a finite state machine with enough complexity for me to give a valid demonstration of my library at work.

In this chapter I will go into the design, workings, documentation, and finally a rundown of the user experience.

4.1. Design

To demonstrate my coffee machine design, I have constructed my finite state machine before implementing.

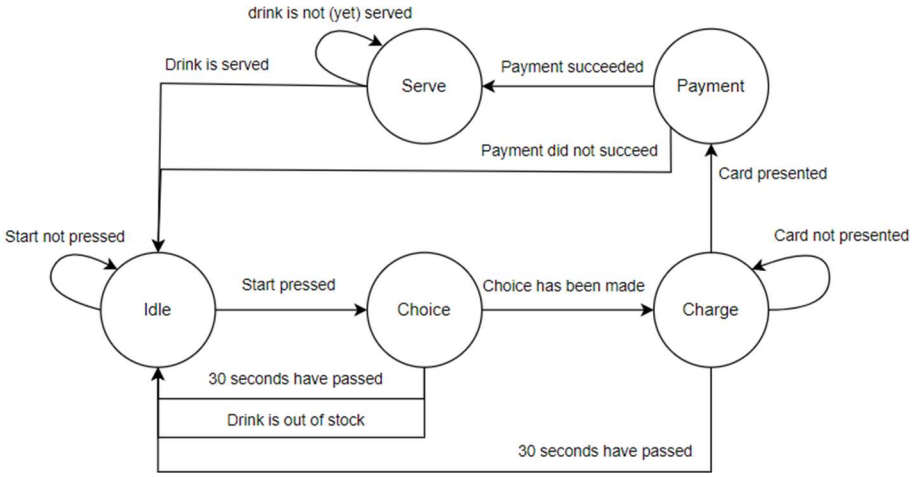

Figure 4: FSM for coffee machine

As seen in the diagram above, the finite state machine does not care for your choice of drink. My teacher stated this not to be necessary to model within the finite state machine. I tend to disagree, as still providing these options will bypass the finite state machine and deplete the goal of the assignment. I will elaborate on this in chapter 5.

To implement the design seen above, I constructed yet another class. This class is modelled below:

CoffeeMachine
- __price_stock_list: Iterable[Iterable[Typevar('Product_info', str, float, int)]] - state_idle: ManualState - state_choice: ManualState - state_charge: ManualState - state_payment: ManualState - state_serve: ManualState
+__init__(price_stock_list: Iterable[Iterable[Typevar('Product_info', str, float, int)]]): CoffeeMachine + run(): None

Table 3: class diagarm for CoffeeMachine

As seen in Table 3, the states are kept within the CoffeeMachine. These states are public as to give users outside of this finite state machine access to its contents. This will not likely be something done, but making these variables private will prevent this from being a possibility. As I made the libraries with preventing wrongdoing in mind, giving outside access will be tolerated.

The __price_stock_list variable is a list of mutable truples. Each containing the name of the product the coffee machine vends, the price in euros, and the amount of cups of said product that are left. The following are the default values:

```
[
    ["Coffee",          .30, 1 ],
    ["Tea",             .20, 10],
    ["Chocolate milk", .40, 7 ]
]
```

Figure 5: price_stock_list defaults

These values are just for demonstration sake. These will provide ample different testing possibilities. Testing is also the reason for the lack of coffee availability.

4.2. Code

So, how does my coffee machine implementation work? Well, to demonstrate I will highlight the most relevant code sections below.

4.2.1. Initialiser

First, lets quickly go over the __init__ method. This method is responsible for constructing the coffee machine. It will do so by setting the __price_stock_list private member variable. After which it will construct all the nodes. I doubt this code to be interesting/relevant for this document. I will refer to the actual code for those interested.

4.2.2. Run method

The only other function in this class is the run method. This method will prompt users with the relevant options to divert from the current state. It will do so with ‘normal‘ python print statements in the terminal. These functionalities are baked into the class. If a GUI would be preferred over the current CLI, one has to override this run method.
The code itself is not that interesting. It is just a statement to catch the users input, followed by a statement to check if the user has selected a drink and to adjust stock accordingly. After which a small code block checks if the user has selected a drink, and it will adjust the selection. Finally it will transition to the appropriate next state.

4.3. Documentation

As with all other classes, I documented this one too. See this documentation, provided by the help method, below.

class CoffeeMachine(builtins.object)
 | CoffeeMachine(price_stock_list: Iterable[Iterable[~PRODUCT_INFO]] = [['Coffee', 0.3, 1], ['Tea', 0.2, 10], ['Chocolate milk', 0.4, 7]]) -> coffee_machine.CoffeeMachine

```
 |
 |  This class contains an implementation of a coffee machine, represented mostly by a Finite State
Machine.
 |  Mostly, because stocks and prices are not contained in the final states as per the assignment as
given.
 |
 |  Methods defined here:
 |
 |  __init__(self, price_stock_list: Iterable[Iterable[~PRODUCT_INFO]] = [['Coffee', 0.3, 1], ['Tea', 0.2,
10], ['Chocolate milk', 0.4, 7]]) -> coffee_machine.CoffeeMachine
 |      This initializer lets one make a CoffeeMachine with a provided product, price, and stock list.
 |      It will also construct all necessary states and transitions.
 |
 |      @params
 |      price_stock_list (Iterable[Iterable[PRODUCT_INFO]]) : A list of mutable truples containing the
names, prices in Euros, and stock levels of all available drinks.
 |
 |      @return
 |      Constructed CoffeeMachine
 |
 |  run(self) -> None
 |      This complex function will roll out the Finite State Machine as defined.
 |      For each transition, the user will be prompted to pick from the following choices:
 |          -1 to terminate in current node.
 |          0 and onwards for each of the defined conditions.
 |      Upon choosing a condition in the terminal, the next ManualState will be entered.
 |
 |      If a drink is ordered, the stock will be lowered appropriately.
 |      If a drink is out of stock and is selected in the menu, the machine will kick you back to idle.
 |      If a drink is out of stock after you got your drink, it will notify you.
 |
 |      The function deliberately does not divide up its functions, as the amount of code is mostly
influenced by the overly bearing print statements.
 |      The actual logic is very minimal and not very subdivisible.
 |      To aid the understanding of the code, I will provide inline comments.
 |      I do not think this to be necessary, but it would aid me to get a better grade
 |
 |  ----------------------------------------------------------------------
```

4.4. Demonstration

To show how the user will interact with the CLI, I printed an example below. In this example, a complete run will be made from pressing start to receiving a coffee. The machine is currently set up with only one cup of coffee in stock. This means that the program should alert the user at the end that the coffee is now out of stock.

```
You are currently in a final state.
Type one of the following options to transition:
  - Type -1 if you want to stop in this state.
  - Type  0 To press start.
  - Type  1 To not press start.
Pleas provide your choice one of the above integers: 0

You are currently not in a final state.
Type one of the following options to transition:
  - Type -1 if you want to stop in this state.
  - Type  0 If 30 seconds have passed.
  - Type  1 To choose Coffee.
  - Type  2 To choose Tea.
  - Type  3 To choose Chocolate milk.
Pleas provide your choice one of the above integers: 1
You chose Coffee.
This item costs €0.30.

You are currently not in a final state.
Type one of the following options to transition:
  - Type -1 if you want to stop in this state.
  - Type  0 If 30 seconds have passed.
  - Type  1 To present your card.
  - Type  2 To not present your card.
Pleas provide your choice one of the above integers: 1

You are currently not in a final state.
Type one of the following options to transition:
  - Type -1 if you want to stop in this state.
  - Type  0 If payment has succeeded.
  - Type  1 If payment has not succeeded.
Pleas provide your choice one of the above integers: 0

You are currently not in a final state.
Type one of the following options to transition:
  - Type -1 if you want to stop in this state.
  - Type  0 If the drink is served.
  - Type  1 If the drink is not (yet) served.
Pleas provide your choice one of the above integers: 0
The machine is now out of Coffee.

You are currently in a final state.
Type one of the following options to transition:
  - Type -1 if you want to stop in this state.
  - Type  0 To press start.
  - Type  1 To not press start.
```

Figure 6: CLI demonstration one coffee purchase

Now, when a user wants to buy another coffee, this is what happens:

```
The machine is now out of Coffee.

You are currently in a final state.
Type one of the following options to transition:
  - Type -1 if you want to stop in this state.
  - Type  0 To press start.
  - Type  1 To not press start.
Pleas provide your choice one of the above integers: 0

You are currently not in a final state.
Type one of the following options to transition:
  - Type -1 if you want to stop in this state.
  - Type  0 If 30 seconds have passed.
  - Type  1 To choose Coffee.
  - Type  2 To choose Tea.
  - Type  3 To choose Chocolate milk.
Pleas provide your choice one of the above integers: 1
Sorry, but this drink is out of stock.

You are currently in a final state.
Type one of the following options to transition:
  - Type -1 if you want to stop in this state.
  - Type  0 To press start.
  - Type  1 To not press start.
Pleas provide your choice one of the above integers:
```

Figure 7: buying out of stock drinks

In the case of a user giving a wrong input. For example, 3 should be in the alphabet because it is available in at least one of the menus. Yet not every menu should respond to said input. Because of that, for each input the validity will be checked. See below for an example.

```
You are currently in a final state.
Type one of the following options to transition:
  - Type -1 if you want to stop in this state.
  - Type  0 To press start.
  - Type  1 To not press start.
Pleas provide your choice one of the above integers: a
Your input was invalid, please try again.
Pleas provide your choice one of the above integers: 3
Your input was invalid, please try again.
Pleas provide your choice one of the above integers:
```

Figure 8: catch wrong input

Now if a user wants to stop in the current state, this is what is shown:

```
You are currently in a final state.
Type one of the following options to transition:
  - Type -1 if you want to stop in this state.
  - Type  0 To press start.
  - Type  1 To not press start.
Pleas provide your choice one of the above integers: -1
You chose to stop the program in an accepting state.
```

Figure 9: terminating in an accepting state

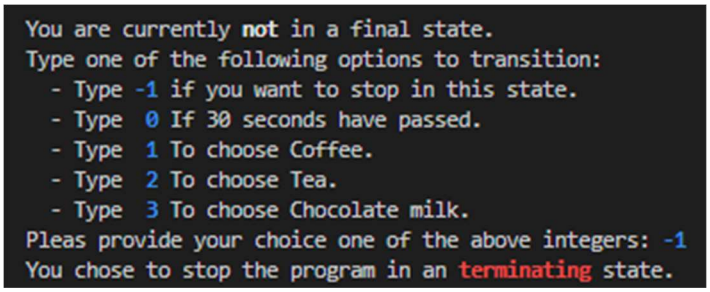

Figure 10: terminating in a terminating state

5. Reflection

I have already discussed my grievances regarding this exercise with my teacher. I was planning to elaborate on this in this section of the document. But I feel as though my grievances were heard and I will therefore not go in to this much more. However, if I wasn't clear enough or whatever it may be. Please don't hesitate to contact me, either in person or via email.