Individual assignment – Trains

Background

1. Two CSV-files are given which are to be construed into a simulation of trains moving through a fictive train network with a given probability.
2. See if it´s possible to go from given point “X” to “Y” withing “*n”* time units in the given CSV-file.

Method

Structuring data

At first, I had to reform the data in the CSV-files into a way that I could easily manipulate to get the result. Most of the time was spend on how I should structure the program, rather than solving the different problems. I try to give a short visual overview in figure 1 of the thought process.

I used the following modules:

- Itertools (permutations and combinations)
- Random

Walkthrough Task 1

1a. First, the program asks the user for the two CSV-files (train network and probability) and the number of trains to simulate.

1b. The program sees if the files exist and if the number of trains given by the user is only numbers.

2a. To more easily use the data provided by the user, I restructured it. All the restructuring of data is put under one class. *Note: It´s unclear in the instructions if the delay occurs traveling to or from the station, or if the probability is additive. I assume it´s whilst traveling from the station.*

- Probability:
    - Returns a dictionary of probability:
        - {"A":0.01, "B":0.2}
- Train stations:
    - Returns a dictionary of train stations:
        - {"green": ["A", "B"]}

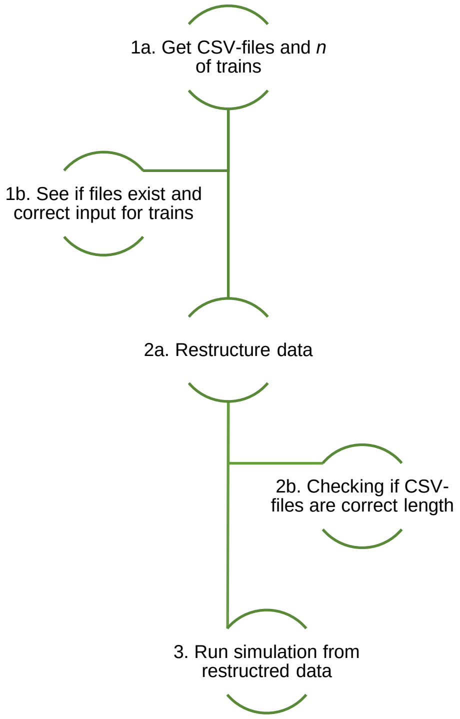

1a. Get CSV-files and *n* of trains
1b. See if files exist and correct input for trains
2a. Restructure data
2b. Checking if CSV-files are correct length
3. Run simulation from restructred data

Figure 1

- The code discriminates between South and North in the CSV-file, e.g: train_stations = {"green": ["A", "B"]}, then train_stations["green"][0] is the most Northern station whilst train_stations["green"][-1] is the most Southern station. This is true for no matter how many stations.

- Trains:
    - Returns a dictionary of trains, e.g: {1: ['blue', 'C', 'S']}
      Key == train number, Value [0] == Line, Value [1] == Current station, Value [2] == Direction of travel.

2b. class Initializing_train_simulator. The program checks if the CSV-files are correct lengthwise.

3. class simulation.

A while loop with options that controls for input.

Walkthrough Task 2

1a. Gather input from user

- Start station, end station and number of jumps

1b. Control user input is correct

- Checks if stations exist
- Checks if a number is given as "number of jumps

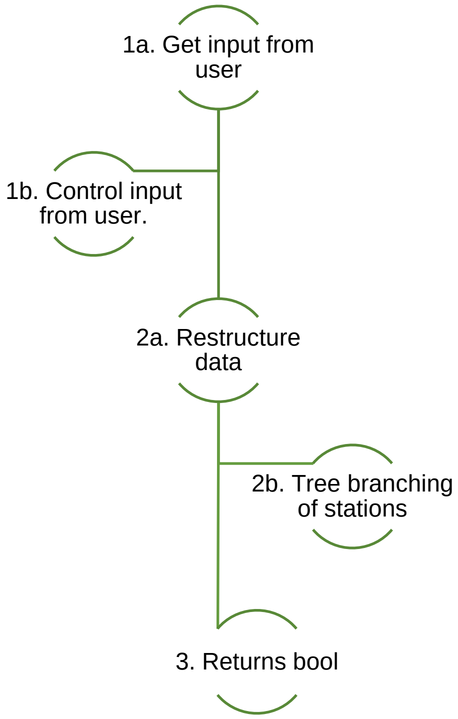
2a. Convert train stations as to a set, e.g:
{"A", "B", "C"}

2b. Method that look for connecting stations and continuously checks with another method if the start and end station is connected.

If the stations are on the same line, no branching occurs, else:

[start_station] + path + [end_station]

path = continuously trying all the ways the train could travel by using branching.



3. Returns Boolean answer, if it´s possible to travel between given stations within the given number of jumps.

Testing

Testing was done by:

- CSV-file with different travel direction (N/S)
- Wrong length of CSV files
- Different station systems and lengths
- Wrong input of stations, number of trains, number of jumps
- Wrong name of CSV-file

It´s hard to calculate for EVERY wrong input available. But I feel like I got the most common ones. I did NOT check if the CSV-file contains correct information.

Discussion

Optimization versus readability

The focus of the code has been readability and not optimization of run-time of different blocks. I´ve tried to:

1. Keep each method/function maximum of one page
2. Maximum 100 lines of code

I´ve tried using large CSV-files for testing with 1000´s of trains. But it runs flawlessly on my computer.

What to improve

The use of classes is subpar at best. I´m not comfortable enough to fully utilize classes with constructors, inheritance, polymorphism, instance attributes and so on. If I would have to improve anything, it would be the structuring of the different methods in the classes and the classes themselves.

Test files:

Feel free to manipulate to your will whilst testing. If you´re uncertain how to manipulate the CSV-files, please read up on the INDU-assign for "Trains". Remember to keep the files in the same directory as the program:

Connections:

A,B,blue,S

B,C,blue,S

C,D,blue,S

X,Y,green,S

Y,C,green,S

C,Z,green,S

Probability:

A,0.001

B,0.001

C,0.2

D,0.001

X,0.1

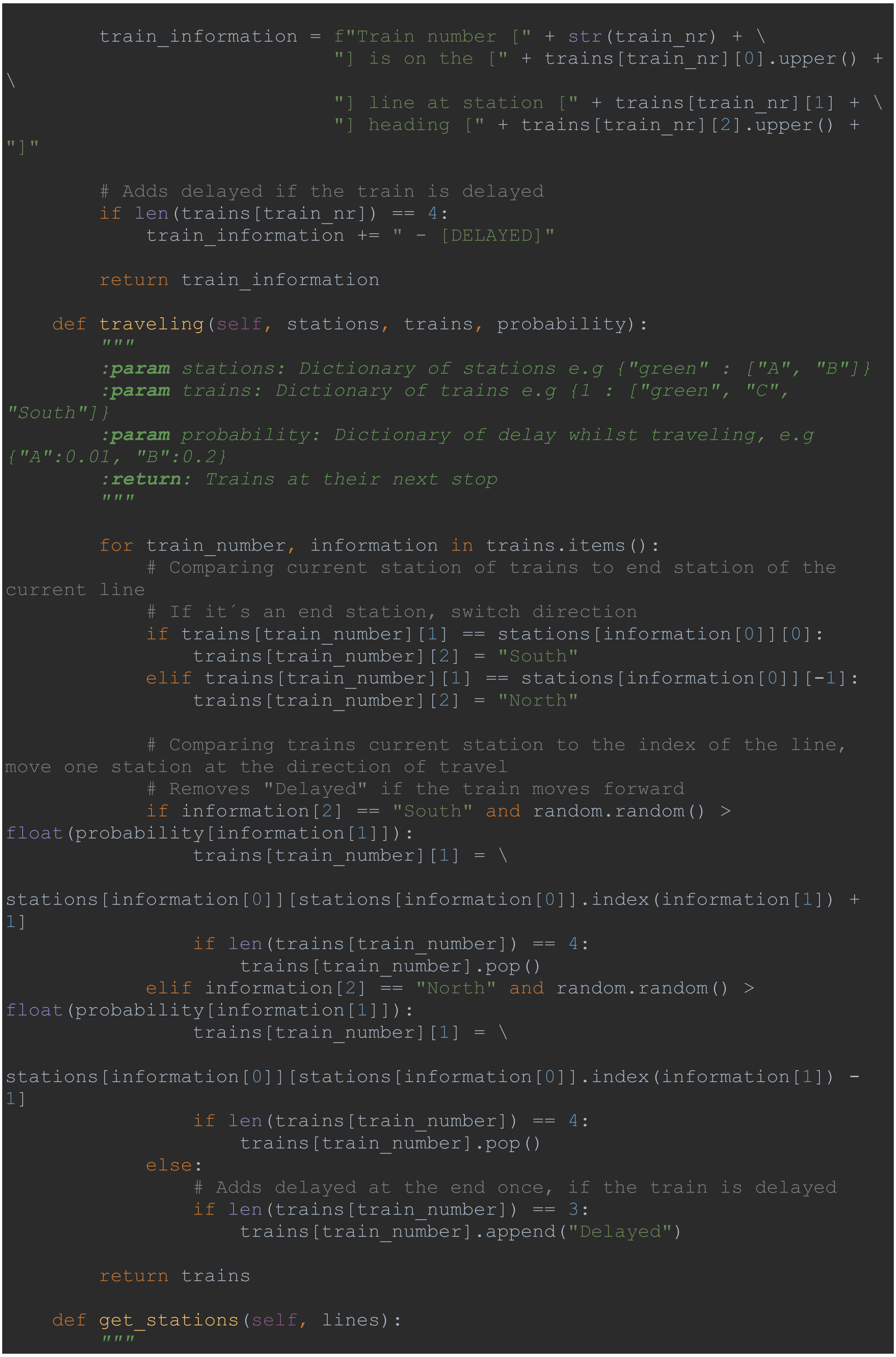
Y,0.1

Z,0.1

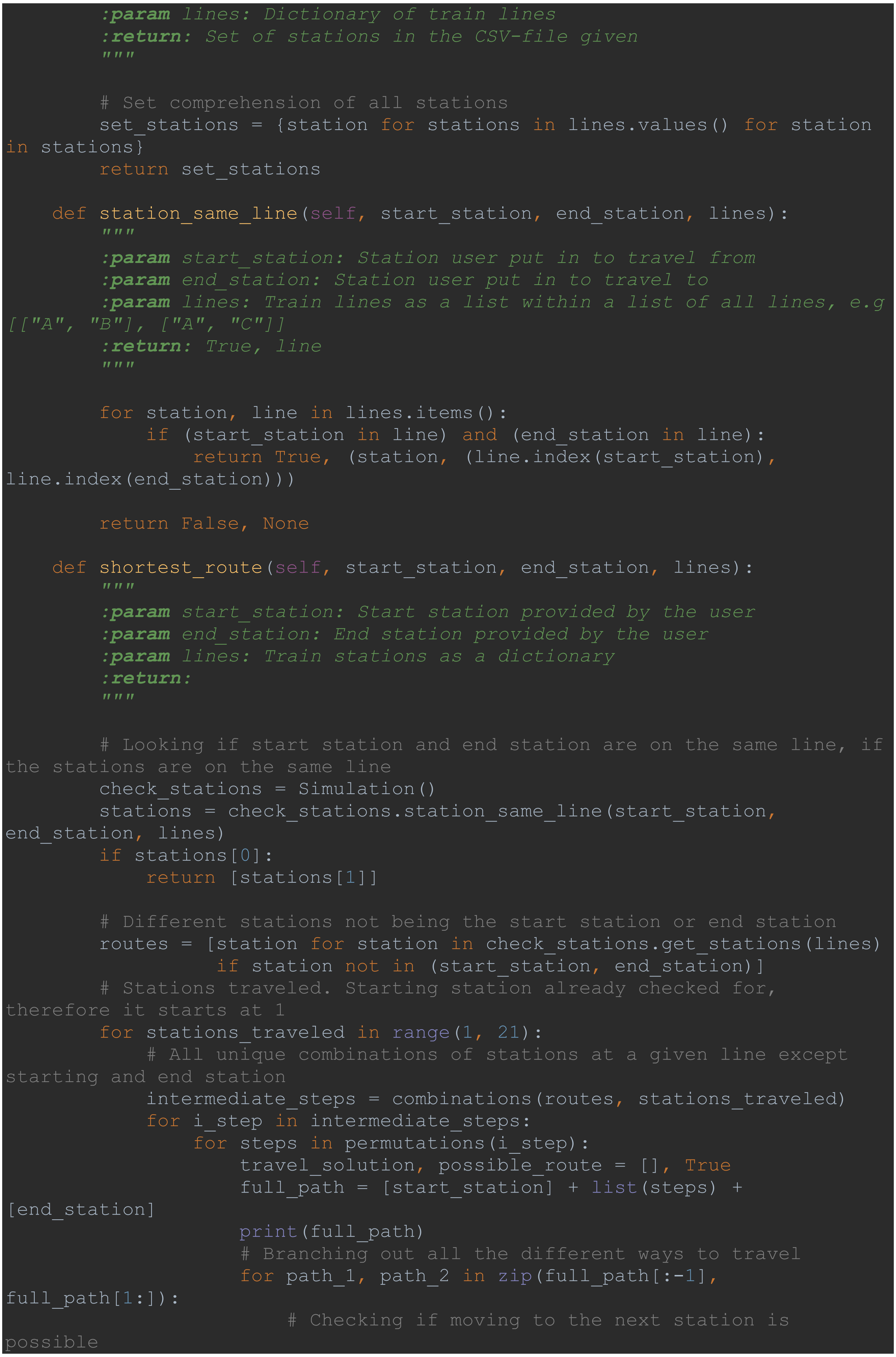
Code:

```
import random
from itertools import combinations, permutations


class Initializing_train_simulator:

    def __init__(self, csv_stations, csv_probability, train_count):
        self.csv_stations = csv_stations
        self.csv_probability = csv_probability
        self.train_count = train_count

    def structure_stations(self):
        """
        :return: Dictionary of stations from CSV-file
        """

        # Importing stations and direction from CSV-file
        with open(self.csv_stations, "r") as h:
            stations_structured = {}
            for line in h:
                stations = line.split(",")
                assert len(stations) == 4, "Please look over the CSV-file
containing stations."

                # If the key (station color) is in the dictionary, add the
rest of the values as a list
                if stations[2] in stations_structured.keys():
                    # Adding stations depending on direction (South/North)
                    if stations[3][0].upper() == "S":

stations_structured[stations[2]].append(stations[1])
                    elif stations[3][0].upper() == "N":
                        stations_structured[stations[2]].insert(0,
stations[1])

                # If the key (station color) is not in the dictionary, add
it
                if stations[2] not in stations_structured.keys():
                    # Adding stations depending on direction (South/North)
                    if stations[3][0].upper() == "S":
                        stations_structured[stations[2]] = [stations[0],
stations[1]]
                    elif stations[3][0].upper() == "N":
                        stations_structured[stations[2]] = [stations[1],
stations[0]]
```

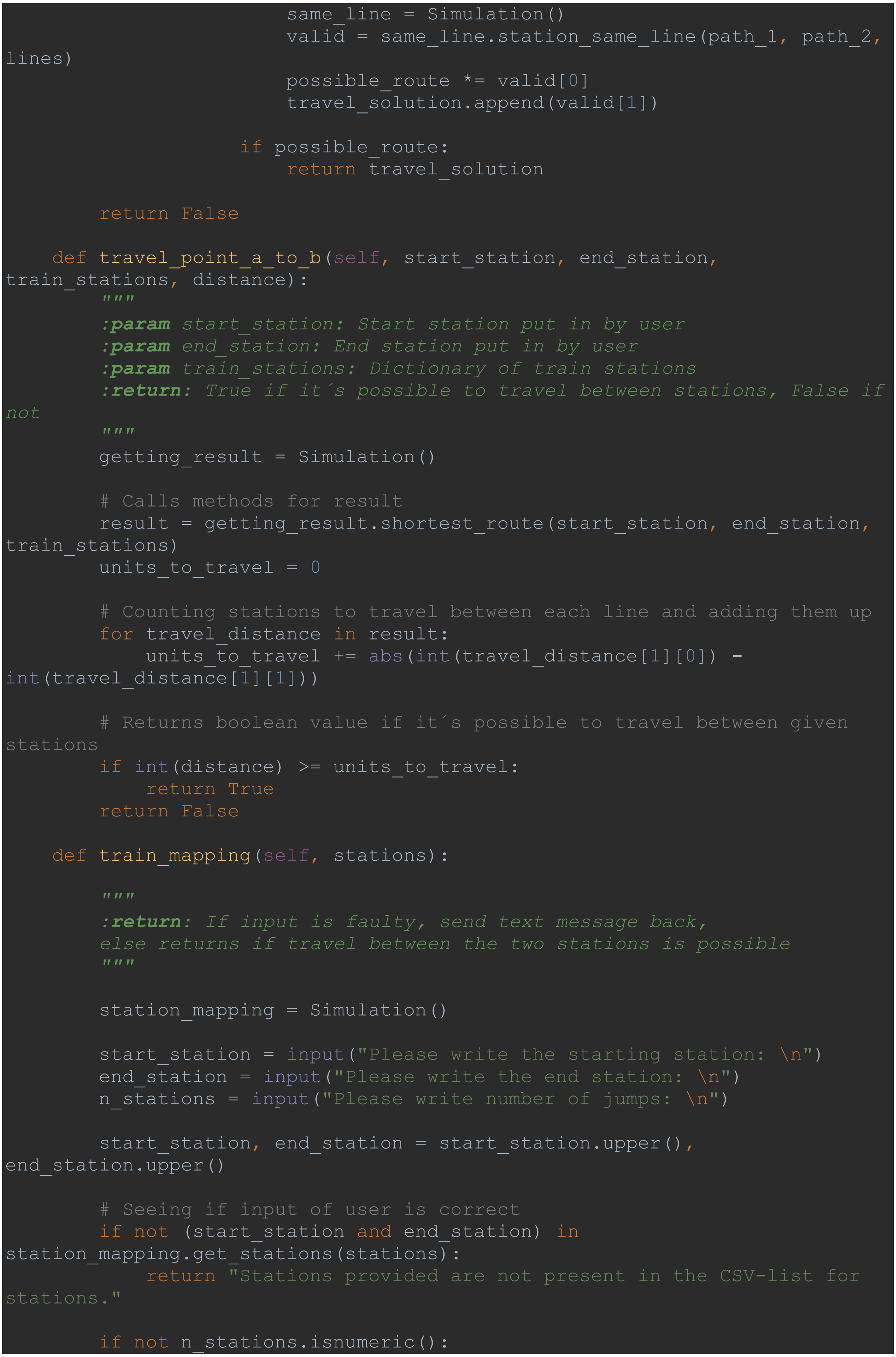
```
        return stations_structured

    def get_trains(self, stations):
        """
        Method for initializing trains
        :param stations: Uses method "structure_stations" as a base
        :return: Dictionary of trains, e.g {1: ['blue', 'C', 'S']}
        Key == train number
        Value [0] == Line
        Value [1] == Current station
        Value [2] == Direction of travel
        """

        train_color = [i for i in stations.keys()]
        trains = {}
        direction = ["South", "North"]

        # Random.choice is pseudo-random, so the key is associated with the
value
        for i in range(1, self.train_count + 1):
            train = random.choice(train_color)
            trains[i] = [train, random.choice(stations[train]),
random.choice(direction)]

        return trains

    def get_probability(self):
        """
        :return: Dictionary of probability of delay when traveling to a
station
        """

        # Open CSV-file of probability of delay
        with open(self.csv_probability, "r") as h:
            probability_structured = {}
            # For-loop for creating dictionary
            for line in h:
                probability = line.split(",")
                assert len(probability) == 2, "Please look over the CSV-
file containing probabilities."
                probability_structured[probability[0]] = probability[1][:-
1]

        return probability_structured


class Simulation:


    def train_information(self, trains):
        """
        :param trains: Dictionary of trains e.g {1 : ["green", "C",
"South"]}
        :return: Returns information about requested train number
        """

        # Presenting choices of trains
        trains_nr_list = [i for i in trains.keys()]
        presenting_choices = f"{trains_nr_list[0]} - {trains_nr_list[-1]}"
        train_nr = int(input("Which train [" + presenting_choices + "]: "))
```



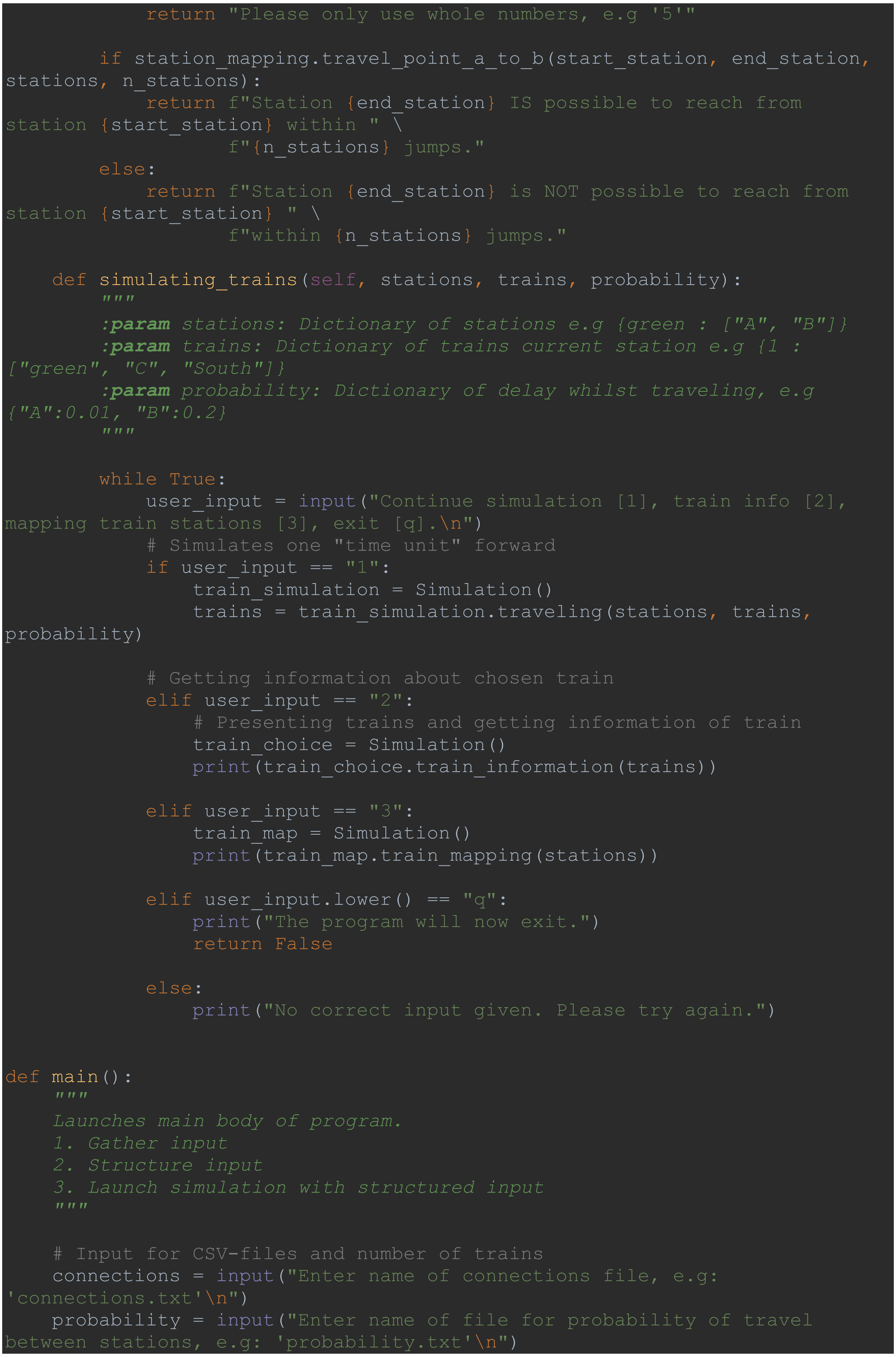
```
        train_information = f"Train number [" + str(train_nr) + \
                            "] is on the [" + trains[train_nr][0].upper() +
\
                            "] line at station [" + trains[train_nr][1] + \
                            "] heading [" + trains[train_nr][2].upper() +
"]"

        # Adds delayed if the train is delayed
        if len(trains[train_nr]) == 4:
            train_information += " - [DELAYED]"

        return train_information

    def traveling(self, stations, trains, probability):
        """
        :param stations: Dictionary of stations e.g {"green" : ["A", "B"]}
        :param trains: Dictionary of trains e.g {1 : ["green", "C",
"South"]}
        :param probability: Dictionary of delay whilst traveling, e.g
{"A":0.01, "B":0.2}
        :return: Trains at their next stop
        """

        for train_number, information in trains.items():
            # Comparing current station of trains to end station of the
current line
            # If it´s an end station, switch direction
            if trains[train_number][1] == stations[information[0]][0]:
                trains[train_number][2] = "South"
            elif trains[train_number][1] == stations[information[0]][-1]:
                trains[train_number][2] = "North"

            # Comparing trains current station to the index of the line,
move one station at the direction of travel
            # Removes "Delayed" if the train moves forward
            if information[2] == "South" and random.random() >
float(probability[information[1]]):
                trains[train_number][1] = \

stations[information[0]][stations[information[0]].index(information[1]) +
1]
                if len(trains[train_number]) == 4:
                    trains[train_number].pop()
            elif information[2] == "North" and random.random() >
float(probability[information[1]]):
                trains[train_number][1] = \

stations[information[0]][stations[information[0]].index(information[1]) -
1]
                if len(trains[train_number]) == 4:
                    trains[train_number].pop()
            else:
                # Adds delayed at the end once, if the train is delayed
                if len(trains[train_number]) == 3:
                    trains[train_number].append("Delayed")

        return trains

    def get_stations(self, lines):
        """
```



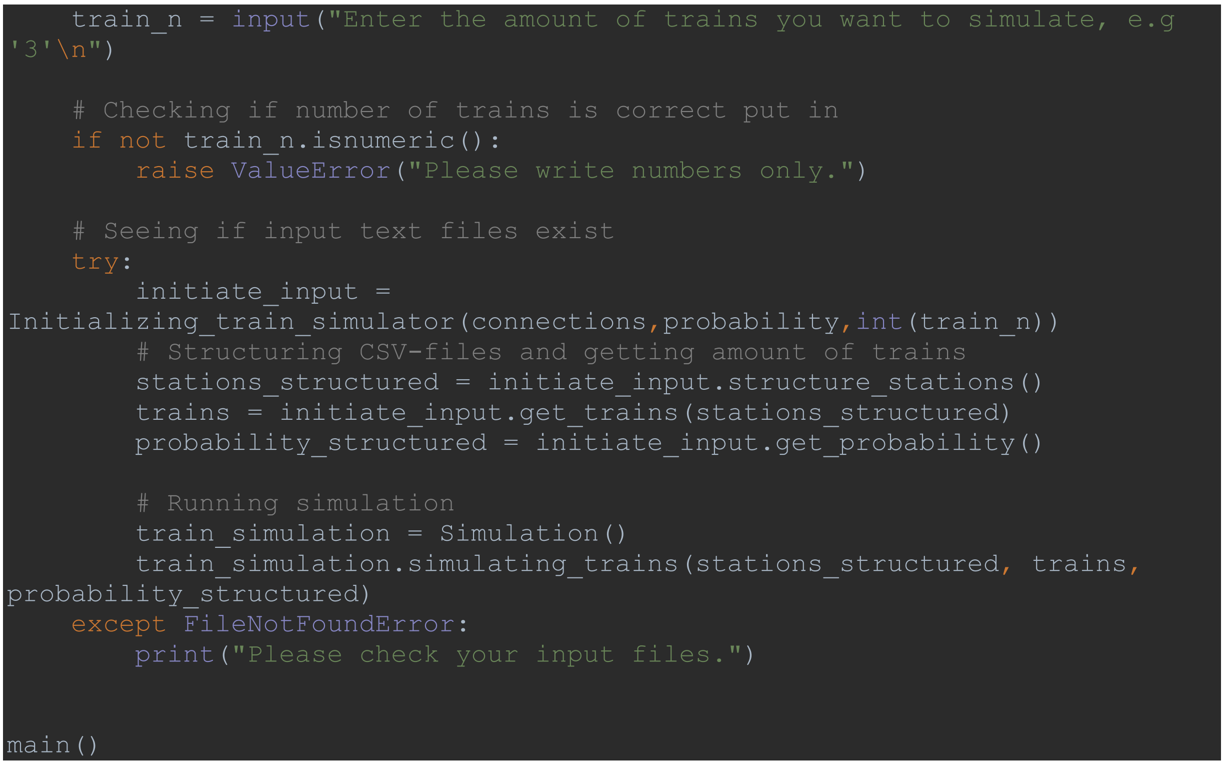
```
        :param lines: Dictionary of train lines
        :return: Set of stations in the CSV-file given
        """

        # Set comprehension of all stations
        set_stations = {station for stations in lines.values() for station
in stations}
        return set_stations

    def station_same_line(self, start_station, end_station, lines):
        """
        :param start_station: Station user put in to travel from
        :param end_station: Station user put in to travel to
        :param lines: Train lines as a list within a list of all lines, e.g
[["A", "B"], ["A", "C"]]
        :return: True, line
        """

        for station, line in lines.items():
            if (start_station in line) and (end_station in line):
                return True, (station, (line.index(start_station),
line.index(end_station)))

        return False, None

    def shortest_route(self, start_station, end_station, lines):
        """
        :param start_station: Start station provided by the user
        :param end_station: End station provided by the user
        :param lines: Train stations as a dictionary
        :return:
        """

        # Looking if start station and end station are on the same line, if
the stations are on the same line
        check_stations = Simulation()
        stations = check_stations.station_same_line(start_station,
end_station, lines)
        if stations[0]:
            return [stations[1]]

        # Different stations not being the start station or end station
        routes = [station for station in check_stations.get_stations(lines)
                  if station not in (start_station, end_station)]
        # Stations traveled. Starting station already checked for,
therefore it starts at 1
        for stations_traveled in range(1, 21):
            # All unique combinations of stations at a given line except
starting and end station
            intermediate_steps = combinations(routes, stations_traveled)
            for i_step in intermediate_steps:
                for steps in permutations(i_step):
                    travel_solution, possible_route = [], True
                    full_path = [start_station] + list(steps) +
[end_station]
                    print(full_path)
                    # Branching out all the different ways to travel
                    for path_1, path_2 in zip(full_path[:-1],
full_path[1:]):
                        # Checking if moving to the next station is
possible
```



```
                        same_line = Simulation()
                        valid = same_line.station_same_line(path_1, path_2,
lines)
                        possible_route *= valid[0]
                        travel_solution.append(valid[1])

                    if possible_route:
                        return travel_solution

        return False

    def travel_point_a_to_b(self, start_station, end_station,
train_stations, distance):
        """
        :param start_station: Start station put in by user
        :param end_station: End station put in by user
        :param train_stations: Dictionary of train stations
        :return: True if it´s possible to travel between stations, False if
not
        """
        getting_result = Simulation()

        # Calls methods for result
        result = getting_result.shortest_route(start_station, end_station,
train_stations)
        units_to_travel = 0

        # Counting stations to travel between each line and adding them up
        for travel_distance in result:
            units_to_travel += abs(int(travel_distance[1][0]) -
int(travel_distance[1][1]))

        # Returns boolean value if it´s possible to travel between given
stations
        if int(distance) >= units_to_travel:
            return True
        return False

    def train_mapping(self, stations):

        """
        :return: If input is faulty, send text message back,
        else returns if travel between the two stations is possible
        """

        station_mapping = Simulation()

        start_station = input("Please write the starting station: \n")
        end_station = input("Please write the end station: \n")
        n_stations = input("Please write number of jumps: \n")

        start_station, end_station = start_station.upper(),
end_station.upper()

        # Seeing if input of user is correct
        if not (start_station and end_station) in
station_mapping.get_stations(stations):
            return "Stations provided are not present in the CSV-list for
stations."

        if not n_stations.isnumeric():
```



```
            return "Please only use whole numbers, e.g '5'"

        if station_mapping.travel_point_a_to_b(start_station, end_station,
stations, n_stations):
            return f"Station {end_station} IS possible to reach from
station {start_station} within " \
                   f"{n_stations} jumps."
        else:
            return f"Station {end_station} is NOT possible to reach from
station {start_station} " \
                   f"within {n_stations} jumps."

    def simulating_trains(self, stations, trains, probability):
        """
        :param stations: Dictionary of stations e.g {green : ["A", "B"]}
        :param trains: Dictionary of trains current station e.g {1 :
["green", "C", "South"]}
        :param probability: Dictionary of delay whilst traveling, e.g
{"A":0.01, "B":0.2}
        """

        while True:
            user_input = input("Continue simulation [1], train info [2],
mapping train stations [3], exit [q].\n")
            # Simulates one "time unit" forward
            if user_input == "1":
                train_simulation = Simulation()
                trains = train_simulation.traveling(stations, trains,
probability)

            # Getting information about chosen train
            elif user_input == "2":
                # Presenting trains and getting information of train
                train_choice = Simulation()
                print(train_choice.train_information(trains))

            elif user_input == "3":
                train_map = Simulation()
                print(train_map.train_mapping(stations))

            elif user_input.lower() == "q":
                print("The program will now exit.")
                return False

            else:
                print("No correct input given. Please try again.")


def main():
    """
    Launches main body of program.
    1. Gather input
    2. Structure input
    3. Launch simulation with structured input
    """

    # Input for CSV-files and number of trains
    connections = input("Enter name of connections file, e.g:
'connections.txt'\n")
    probability = input("Enter name of file for probability of travel
between stations, e.g: 'probability.txt'\n")
```



```
    train_n = input("Enter the amount of trains you want to simulate, e.g
'3'\n")

    # Checking if number of trains is correct put in
    if not train_n.isnumeric():
        raise ValueError("Please write numbers only.")

    # Seeing if input text files exist
    try:
        initiate_input =
Initializing_train_simulator(connections,probability,int(train_n))
        # Structuring CSV-files and getting amount of trains
        stations_structured = initiate_input.structure_stations()
        trains = initiate_input.get_trains(stations_structured)
        probability_structured = initiate_input.get_probability()

        # Running simulation
        train_simulation = Simulation()
        train_simulation.simulating_trains(stations_structured, trains,
probability_structured)
    except FileNotFoundError:
        print("Please check your input files.")


main()
```