Project Summary

This application is designed to simulate and calculate the distance between two vehicles moving in the same direction, plus the time required for the following (interceptor) vehicle to catch up to the leading (target) vehicle, given varying rates of speed for both vehicles. The project consists of 3 files: `interceptor.py`, which contains the `Intercept` class, `test_values.py`, containing the `TestValues` class, and `simulation.py`, containing the `Simulation` class.

The simulation works by first generating and calculating values for the speed of both vehicles and the distance between them for every second the simulation will be ran. Each value for the target speed, interceptor speed, and distance at every second will be passed into an instance of the `Intercept` class in order to return the final values to the user for that second in time.

Once every value has already been calculated for the duration of the simulation, the simulation runs as if it was being used in real-time. For real-life application, the values would be calculated in real-time, once per second, using live GPS data. Preparing this program for real-life application would require much further development and is beyond the scope of what this project was created for.

Project Background

I came up with this project while following a few miles behind my parents on the freeway to Los Angeles. I wanted to know how long it would take to catch up with them, and developed an application to calculate the time required given the rates of speed for both vehicles, and the current distance between the two. The intention of this project was to practice and apply my programming skills by creating a solution to a fairly simple problem that can be found in real life. However, I quickly realized that this solution would be useless if I couldn't simulate the problem in code. This led to developing the `TestValues` and `Simulation` classes in order to recreate the problem digitally.

Technical Information

`interceptor.py`

Within this file is the Interceptor class, which is used to calculate the remaining distance between both vehicles (`gap`), the time required to intercept the target at the

current rate of speed (`time`), and the distance gained/lost every second (`net_distance`). In this version, `net_distance` is not used. For real-life application, live GPS data would be passed to the `Intercept` class once every second to calculate the time and distance values for each second the program is left running.

`interceptor.py` Methods

- `__init__(self)`: Stores distance and speed values of both vehicles for calculation
- `get_time(self)`: Calculates time until interception given current distance & speed rates. Speed rates are converted from Km/h to m/s. Time is calculated by dividing current distance (in meters) by the difference of the rates of speed.
- `get_distance(self)`: Calculates the total distance currently between both vehicles (`gap`). `net_distance` == distance gained/lost each second. `net_distance` is not used in this version.

`test_values.py`

This file contains the `TestValues` class, which is used to generate simulated speed and distance vales that would otherwise be calculated from GPS data. The user is prompted for inputs for the initial distance between both vehicles (in Km), the number of seconds (*n*) the simulation will run for, and the min/max speed values for both vehicles (in Km/h). First, *n* speed values for both vehicles are calculated randomly with constraints in order to mimic the ebb and flow of traffic speed on a freeway. Subsequently, values for the velocity and acceleration are calculated for each second. Once speed and acceleration has been calculated, they are used to calculate the new distance between the vehicles at each second. Values input as Km & Km/h are converted to meters and meters/second for calculation.

`test_values.py` Methods

- `__init__(self)`: creates the `values` dictionary used to store the values needed for subsequent calculations.
- `get_input(self)`: User inputs values for starting distance, the min/max speed range (Km/h) of both vehicles, and the number of seconds (*n*) simulation runs for.
- `generate_values(self)`: Generates the first speed values by randomly choosing a value between the user-defined speed ranges for both vehicles.

- `s1_speed_values(self, speed_range=3)`: Calculates speed values for target vehicle. Subsequent speed values are calculated by first randomly determining if the vehicle will accelerate, decelerate, or remain at constant speed. If accelerating or decelerating, the next value will be within the min/max speed range defined by the user in `generate_values()`, and not greater/less than the current value +- the absolute value of the `speed_range` parameter. I.e: the vehicle will not accelerate/decelerate faster than `speed_range` Km/h in one second (`3` by default).
- `s2_speed_values(self, speed_range_2=3)`: Calculates speed values for interceptor vehicle.
- `acceleration_values(self)`: Calculates the initial and final velocities (m/s) of both vehicles after each second. Used to calculate the acceleration/deceleration of each vehicle after each second.
- `calc_a_vals(self)`: Calculates acceleration in m/s from initial/final velocities to account for differences in the vehicles rates of speed between each second.
- `calc_d_gap(self)`: Calculates distance gap between vehicles by first totaling the distances traveled within each second, then subtracting the total distance traveled by the interceptor from the sum of the total distance traveled by the target and the initial distance. Further refactoring/design changes will need to be implemented in order to make calculations with a time interval greater than 1 second.
- `clear_used_values(self)`: Deletes the obsolete values used to calculate the final speed and distance values that will be passed to the `Intercept` class.
- `convert_km(self, raw_values)`: Converts meters to kilometers. Practical application of this function within the project will require further refactoring and/or design changes. It is not used in this version.
- `create_test(self)`: The functions above are run as one for simplicity and ease of use.

`simulation.py`

This file contains the `Simulation` class, which generates *n* values with an instance of `TestValues`, then passes those values to *n* instances of the `Intercept` class. The data calculated by the `Intercept` class instances is displayed to the user and updated once every second, for *n* seconds. Along with the current distance and

time until interception, the user is shown the percentage of the initial distance closed, the number of seconds that have passed, and moving averages for the distance (meters) and time (seconds) of the last *k* seconds (*k* == `10` by default).

`simulation.py` Methods

- `__init__(self)`: Moving averages are stored in a `defaultdict(list)` object. Keys are the variable whose average is being calculated, and each key-value in the list is the new moving average value after each second.
- `get_values(self)`: Creates instance of `TestValues` class and returns generated vales
- `run_sim(self)`: Simulation is created by passing generated values into *n* instances of `Intercept` class and returning results for each instance
- `calculate_moving_avgs(self, values, key_name='default', start=0, multiple=10)`: Calculates moving averages for a dataset in array ordered by ascending chronological order. Values are stored under `key_name` in `moving_avgs` created in `__init__()`. Averages are calculated by the last `10` items by default. Will require refactoring to support live data and new values being appended to array.
- `get_percentage(self)`: Calculates and returns percentage of initial distance closed by interceptor vehicle. Percentage can be greater than 100 if current distance exceeds initial distance.
- `convert_time(self)`: Converts time returned by each `Intercept` instance from seconds to minutes:seconds.milliseconds.
- `display_stats(self, mavg_period=10)`: Displays live stats to the user for values generated by the simulation for each second. `time.sleep(1)` is used here to simulate one set of calculations per second.