# driver-manager

### Environment

\*\*Language\*\*: Python 3.6.2

\*\*Operating System\*\*: Windows

### Approach

#### Assumptions

Apart from the assumptions included in the problem statement I kept the following assumptions on mind while designing the application:

1. The driver’s name uniquely identifies each driver.

2. The input file name (or complete path) should be provided for a successful execution of the application. The output file’s name must be provided.

3. The input and output files should have full read/write permissions.

4. The contents of the input file should be in adherence to the input samples given in the problem statement.

#### Modelling

This application could have been designed without using an object-oriented approach. But by going with an object-oriented design approach I was able to give the code more readability and was able to divide the sub-tasks in a more meaningful way. The object-oriented approach led to the division of the two principle tasks of differentiating between drivers and trips to creating a new driver object and filling a driver objects with details respectively.

The ```Driver``` class represents a driver for this application. The attributes of the driver are represented in the form of three class attributes: ```name```, ```miles``` and ```time```. Where ```name``` is the name of the driver, ```miles``` is the total miles driven by the driver and finally ```time``` is the total minutes spent by the driver in covering the miles. The speed of the driver is calculated by invoking a class method named ```get\_speed()```. The speed is not an attribute of the Driver class because it is a function of the attributes and not a direct property of the Driver class.

To store the ```Driver``` objects, a dictionary has been used because search operations on a dictionary could be performed in a average case complexity of \*\*O(1)\*\*. To sort the dictionary according to the miles traveled by the drivers, an inbuilt function named ```sorted()``` has been used, which has a worst case complexity of \*\*O(n log n)\*\*.

### Testing

The application has gone through a comprehensive unit testing. Python's inbuilt ``unittest`` library has been used to develop all unit tests. The test cases has been implemented to handle the maximum number of edge cases that came to my mind. For the methods returning the dictionary I could only come up with a test which checks whether the return value is a dictionary or not. This is an area in the testing module which might have a future scope of improvement.

\*\*Running the tests:\*\* From the command line inside the root directory run this command: ``python -m unittest test\_rootmain.TestRootmain``

### Running the application

The application must be run from the command line. On the command line from the directory containing the file \*\*rootmain.py\*\*, run: ```python rootmain input\_file.txt output\_file.txt```.

Where \*\*input\_file.txt\*\* and \*\*output\_file.txt\*\* are the respective files to read from and write to.