HUMAN-CENTRED OPTIMISATION FOR FLEET VEHICLE ROUTING

Jacob McCarthy

Department of Computer Science
Aston University
Birmingham, UK.

I. INTRODUCTION

The concept of fleet vehicle routing has been explored and implemented by some of the great minds within Computer Science and Mathematics. While intelligent fleet vehicle routing has paved the way for impressively fast delivery times on a mass scale, drivers of fleet vehicles report some of the lowest levels of job satisfaction amongst skilled professionals.

Following the aftermath of Brexit and the COVID-19 Pandemic, shortages of skilled drivers have been a pressing issue for many of the UK's largest companies. Fierce competition to employ drivers means that companies will need to do everything they can to hold onto their workforce. Furthermore, drivers have been turning to freelance work within the gig economy, which has been rapidly overtaking traditional employment within this sector.

The aim of this project is to develop a system which will be able to provide fleet vehicle routing solutions over a range of locations, as well as considering parameters which are necessary to ensuring the satisfaction of a human driver workforce. The system will include an intuitive GUI allowing for monitoring of drivers and destinations, alongside clear indicators of important information.

II. PREPARATION

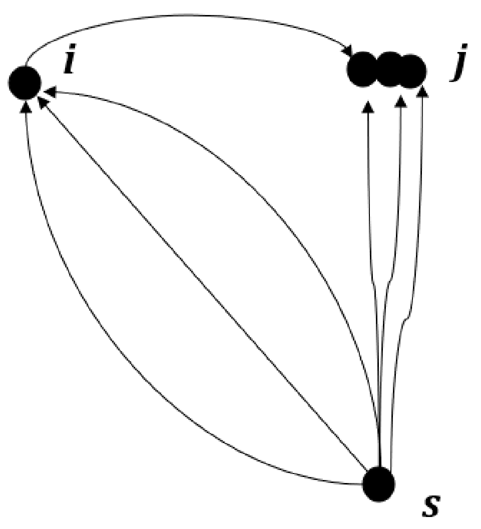
1. Research

a. Historical development of solutions to the FVRP

The world's fascination with Fleet Vehicle Routing Problems (FVRPs) is reflected in the vast amount of research which has been conducted on them over the past sixty years. FVRPs are not only of great economic importance, they also present a famously challenging combinatorial optimisation problem. These two properties have resulted in something of a 'holy grail' scenario; the perfect solution to the FVRP is almost impossibly elusive, yet many fervently seek it due to the untold riches it would bring.

The first implementation of the FVRP was introduced by Dantzig and Ramser in 1959 (Dantzig, 1959). Their algorithm examined the paths taken by lorries delivering petrol. Specifically, stations were assigned to lorries 'in such a manner that station demands are satisfied, and total mileage covered by the fleet is kept to a minimum'. As such, the precedent was set for FVRPs to be solved with the primary objective of minimising cost to the supplier. While their implementation was only illustrated on a toy-sized sample, it served as an introduction of the concept to many and laid foundations for future development and expansion.

During the following years, many heuristic approaches to the FVRP were designed– accepting imperfections in the hope of finding solutions which were approximately optimal. The most famous solution was the 'savings method' published by Clark and Wright (Clarke, 1964).

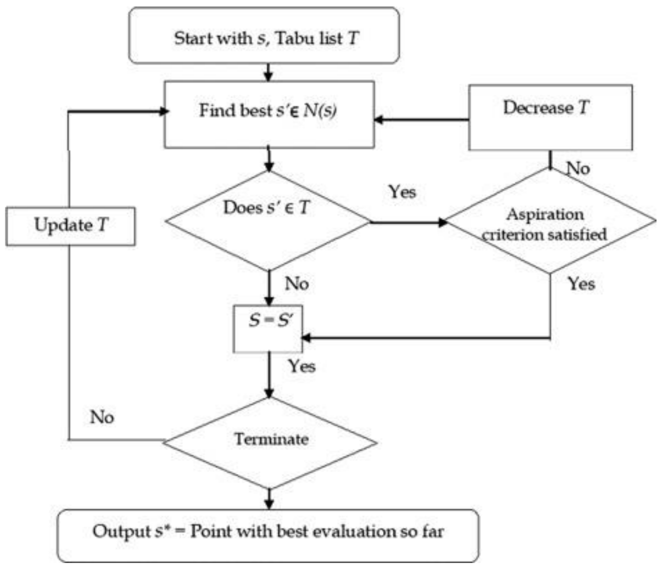


The savings method algorithm was defined as follows:

- Calculate S_{ij} for all *I*, *j*
- Consider cheapest S_{ij}
- If *j* can be appended to *i*
    - merge them to new *i*
    - update all S_{ij}
- Else
    - delete S_{ij}
- Repeat

This algorithm, although simple, is surprisingly effective in producing optimal routing solutions. Its speed and simplicity have allowed it to remain relevant, being used to solve VRPs as recently as 2019 (Kristina, 2019) (Mittal, 2017).

Following these developments, many attempts were made to further optimise FVRPs. However, no significant breakthroughs were made until the 1990s, which saw the introduction of metaheuristics. The earliest implementations of metaheuristics leaned heavily towards the use of tabu search. (Taillard, 1993) (Gendreau, 1994)


Start with s, Tabu list T
Find best s′ ∊ N(s)
Decrease T
No
Does s′ ∊ T
Yes
Aspiration criterion satisfied
Update T
No
Yes
S = S′
Yes
No
Terminate
Output s* = Point with best evaluation so far

In more recent years, FVRP algorithms have tended to use either a combination of multiple operators, such as Pisinger and Ropke's adaptive large neighbourhood search (Pisinger, 2007), or a combination of local and genetic search. (Ho, 2008) (Vidal, 2012)

In 2020, Vidal et al published '*A concise guide to existing and emerging vehicle routing problem variants*'. (Vidal, 2020) In this article, it was identified that "profitability or cost optimisation is the primary concern in the overwhelming majority of VRP studies". Vidal recognises that cost minimizing solutions do not typically possess properties such as ensuring a consistent level of service, which could be addressed through measures such as more rigorous employee training combined with a more satisfied workforce.

b. The changing economy and its effect on skilled drivers

i. The gig economy

In a time where the gig economy is growing rapidly, maintaining the job satisfaction of traditionally employed fleet delivery drivers is important. Gig economy workers are expected to comprise 6.5% of the workforce in London over the next few years (Johnes, 2019). A significant proportion of gig economy jobs involve driving, such as ride-hailing app Uber or Deliveroo's food delivery service.

While the gig economy's treatment of workers is often bemoaned in articles and viewed with moderate disdain by the general public (Healy, 2020), freelance workers reported higher levels of satisfaction in multiple dimensions of their work lives than those holding traditional jobs (Manyika, 2016).

Some of the benefits most frequently cited by gig workers as to why they have taken gig work over conventional employment are flexibility and independence. (Donovan, 2016) Whilst the elimination of a micro-managing boss is impossible to achieve through software, allowing drivers flexibility can certainly be attempted.

ii. The impact of Brexit

Compounding upon the growth of the gig economy is the rising pressure on companies to retain drivers owing to the labour shortage stemming from Brexit and COVID-19. The Road Haulage Association has stated that there is a 'shortfall of roughly 100,000 HGV Drivers' within the UK in August 2021. Subsequently, retailers Tesco, Aldi, M&S, John Lewis, Poundland, and Co-op have offered increased benefits to drivers, with up to £5,000 salary increases and £2,000 'welcome bonuses' (Nazir, 2021).

Despite these bonuses, almost 3000 hauliers are planning a strike on August 23rd over 'low pay and working conditions'. Evidently, skilled drivers are accustomed to poor working conditions and are capitalising on an opportunity to organise and demand better.

A report produced by the European Parliament (European Parliament, 2017) indicated that HGV drivers feel that the quality of various aspects of their role has deteriorated in recent years.

Specifically, the standard of their working environment, hours worked and income levels were below the expectations of drivers.

Clearly, more needs to be done in order to retain a skilled driver workforce. Whilst salary offers are being raised by companies, there is danger that the 'people dimension', as suggested by Sweeney (Sweeney, 2013), is being neglected. This project aims to aid in filling the gap in worker satisfaction through providing workers with greater agency over their assignments.

iii. How to make drivers happy?

Amongst all professions, salary is one of the - if not the - most important factors influencing job satisfaction. (Prockl, 2017) Since changing the salary of drivers falls beyond the scope of this project, focus will instead be held to the non-financial aspects of the job which can be affected by the software.

In a 2014 article titled *Psychological capital: A new tool for driver retention,* Schulz et al. explore the relationship between the positive 'PsyCap' of employees and desirable outcomes. (Schulz, 2014) Psychological capital refers to a set of resources which a person can use to improve their performance on the job: Self efficacy, optimism, hope, and resilience (Luthans, 2015). Self-efficacy refers to a person's confidence in their ability to influence outcomes and overcome hardships which may arise during the job. Increasing self-efficacy would be possible through allowing drivers greater agency over their job, for example allowing them to decide when to take a break, or accounting for personal stops during their shift. Optimism and hope are fairly similar, referring to a driver's expectation of positive outcomes, and ability to solve problems through various avenues. According to Schulz, these two attributes are best addressed through interpersonal connections to others within the organization, as peer reinforcement is key to maintaining a generally positive attitude within the workforce. The final resource, resilience, focuses on the ability to bounce back from challenges, risks, and failures. For drivers, challenges may often present themselves in the form of problems with their vehicle. A system which ensures that drivers' vehicles are kept adequately fuelled combined with regular servicing of fleet vehicles would help to improve the resilience of employees.

To further reduce the impact of challenging scenarios and maximise the drivers' ability to address problems, it is important that drivers do not become fatigued whilst on the job. Not only does this reduce the risk of accidents stemming from tired driving, it also gives the drivers some personal time and prevents them from feeling overworked. In a literature review covering the topic of rest breaks, Tucker identifies multiple studies which highlight the three-hour mark as the most effective interval for driver breaks, leading to an 'almost complete eradication of errors' in the drivers. (Tucker, 2003) (Stave, 1977) (Drory, 1985)

2. Requirements Elicitation

Throughout the development of this project, I have decided to split the creation of the software into five distinct stages. I had considered following the agile software development methodology, however agile is intended for scenarios in which a representative from a client or stakeholder is easily reachable and in close contact with the developer. For this particular project, the end user is unknown, however it is important that the product produced could be usable by a wide range of organizations for their fleet vehicle routing solutions. Instead of agile, the waterfall development methodology was chosen.

With this in mind alongside the elements identified during background research, it was possible to begin the first stage of the waterfall lifecycle: requirements elicitation. Throughout the process of requirements elicitation, a test plan will be derived for each type of testing that the project will be subjected to. The test plan will be able to measure how well the product satisfies the requirements outlined. Before creating a test plan, it is important that a set of both functional and non-functional requirements are identified.

a. Identifying Requirements

In order to elicit the requirements of the system, it was important to identify the stakeholders and model requirements accordingly. In this system, two main stakeholders were identified.

The Drivers- As this project is looking primarily to address the poor working conditions and job satisfaction levels of skilled drivers, it is important that their interests are kept at the forefront of development.

The Managers- Although the focus of this system is on the drivers, a successful system will also provide useful functionality and usability to managers. By making the system appealing to managers, drivers also directly benefit as their managers are more likely to adopt the system.

After identifying these two main groups of stakeholders, user stories were modelled for each group. These user stories can be found in **(Appendix A).** User stories are an important part of requirements analysis, as they help to put requirements in perspective in terms of the experiences of those who will actually be using the software.

For each user story, an either functional or non-functional requirement was created. This set of requirements was then used to create a test plan for many different types of testing, in order to provide a measure by which to establish whether the software project produced is of sufficient quality for its intended goal. This set of requirements can be found in **(Appendix B).**

b. Types of Testing

i. Black Box Testing

To test how the system handles state transitions of drivers, state transition testing will be used. State transition testing is a type of black box testing which focuses on the state that a program is in given various conditions.

Specifically, this testing will be examining the state that *Driver* objects are in during the program's execution. Since the program will effectively model a hub receiving orders from customers, this should reflect the tasks that drivers will be assigned to in real life implementations of the system. For this testing, different states which a *Driver* object may assume were identified alongside justifications. These may be found in **(Appendix C)***.*

A state graph was constructed to illustrate the intended state flow of *Driver* objects. This graph can be found in **(Appendix D)**.

Additionally, a state table was created. The table indicates how events will affect the state of the driver given their current state. For example, while resting, external events will not affect the driver. Conversely, while idling, all events will cause the driver to change state. While refuelling, the only events which will cause the driver to change state are the refuelling process being completed or the driver requesting a break.

All valid states are listed in the left column of the table, and the cells of each state's row denote what state will be assumed given the occurrence of the corresponding event (listed in the top row). The state table can be found in **(Appendix E).**

With the state graph and table constructed, the next step was to create a comprehensive set of black box test cases to be completed following development. This test table can be found in **(Appendix F).**

ii. White Box Testing

Another method of testing to be used within the system is white box testing, specifically branch coverage. Branch coverage aims to ensure that each possible branch from a decision point within a program is executed given the right circumstances. If some branches are not ever reached, it could result in significant portions of code being unreachable. This type of testing is also important as it identifies any abnormal behaviour which could result from some branches. (Frankl, 1993)

Akin to the focus of the black box testing, the branch coverage used within this project will primarily be investigating the code relating to the drivers' AI. Since this code will be ran on multiple threads concurrently, it is important that there are no errors within it.

As white box testing involves examining the internal workings of the code, specifics such as the test plan and program flow diagram will be constructed after the development of the code.

iii. Scenario Testing

Scenario testing focuses more on system and user acceptance levels, focusing more on the business-facing element of the system than white and black box testing methods. This type of testing involves the creation of a credible scenario which could occur within the system, and an exploration of how the system would respond to such a scenario. Kaner defines an ideal scenario as "a story that is motivating, credible, complex, and easy to evaluate" (Cem Kaner, 2013). To comply with these requirements, a scenario was created. The scenario is as follows:

"A customer had been waiting much longer than expected for their order. They called up the business, and to placate them, an employee assigned the order to a driver who was already delivering a different order"

Key characteristics of a scenario according to Kaner	**Justification**
A story	The scenario reflects a logical series of events: the motivations of actors within the system are understandable and their actions are realistic.
Motivating	The system needs to be able to handle this situation accordingly. The manager's attempt to override the system's process could result in the order not being processed, and the customer becoming even more disgruntled.
Credible	Customers often take their frustration out on business staff – particularly so when over the phone. It is believable that an inexperienced manager may try to override the system as a result of pressure from aggressive customers.
Complex	The scenario's implications would affect almost every area of the program and could result in widespread effects if not deal with appropriately
Easy to evaluate	To be successful, the system needs to return the order to the queue without compromising the driver's current delivery.

iv. Usability Testing

Usability testing involves taking some realistic target clients of the project, and getting them to use it. Prospective clients may be given tasks to complete within the system, and asked to give feedback on how the experience was of using the system to complete them. A table was constructed with some tasks that users should be able to do. Once the product is completed, remote testing with potential users will be conducted to establish the product's usability.

Task	**User Feedback**
Identify the location of the business hub, the drivers, the customers, and the petrol stations on the map	
Determine which orders have been seen to and which are still pending driver assignment	
For each driver, identify what their current task is.	

c. Design

The design stage will establish which tools will be used to create the system, as well as what the internal workings of the system may look like.

i. Tools

Before starting the development of any program, it is important to determine which language will be the most appropriate. At earlier stages of the project, Java was considered. This was due to Java's extreme versatility, as it was planned that drivers would be able to run the program on their mobile devices or dashboard 'infotainment' systems. Although this versatility would have been beneficial, there are few appropriate map-based frontend tools for Java. This absence would result in a significantly more complicated development process, and it was decided that the system could simply pass information to drivers who mostly already have their own GPS systems.

Instead, Python was chosen. This was primarily due to the vast array of libraries available for the python platform, as it is more suited to the creation of a program with a map-based GUI than Java.

Folium is an open-source library for Python which builds upon the mapping abilities of JavaScript's leaflet library. It contains a built-in tileset from OpenStreetMap, which can be used for the visual 'map' in this project. Support for markers is also included, which can be used to indicate the position of drivers and orders.

One downside of folium is that it cannot render in 'real time' and respond dynamically to inputs. To mediate this, Flask allows a folium map to be embedded in a webpage. In this way, the map could have a 'refresh' button adjacent to it, which managers could click whenever they need to view up to date information. It is likely that managers would not need to be checking the map all the time, so occasional refreshing should be a sufficient and less computationally expensive solution than real-time rendering.

Since the system will be using real-world geolocation data, it is necessary to find a way of retrieving the required geographic information and feeding it to the program. For this, Google Maps API will be utilised. Google Maps API offers all of the services required for the construction of the project, as well as a $300 USD allowance for new users. There are three main APIs which will be used within the project:

- **Geocoding:** Converting an address into map coordinates or vice versa. This will be useful for plotting the address of orders on the map.

- **Nearby Search:** Finding a specific kind of business or point of interest within a radius from a point. In this program, Nearby Search will be used to find the location of petrol stations.

- **Distance Matrix:** Returns information about how long it takes to drive from one point to another. This will be used to help determine which driver to assign to an order.

In summary, the application will be coded in python, the map will be constructed using Folium, and Flask will provide the necessary utility for a semi-dynamic map. Google Maps API will be used to provide geographical information.

ii. Structure

The program will be constructed with an object-oriented design philosophy in mind. Below is listed each module of the program and their intended purposes.

main: Initialises Flask and sets off program execution

cfg: Holds variables which are specific to the business using the system, such as the surrounding city and location of the delivery 'hub'. Also contains API authorisation keys and API URLs.

mapfunctions: Variety of functions relating to geographic queries such as geocoding, finding nearby petrol stations

mapsetup: Creates the folium map with markers showing the business location, petrol stations, and drivers

order: Contains the Order class. An instance of this object class will be generated to represent each order received by the business. Outside of the Order class will be a method to artificially generate orders for the purpose of product demonstration.

driver: Contains the Driver class. Each driver employed by the business will have an object representation, alongside properties denoting their fuel and fatigue levels amongst others.

orderhandler: Assigns order to drivers based on a combination of parameters.

To demonstrate the relationship between the different modules and classes, a UML class diagram was created. This diagram can be found in **(Appendix G).**

III. THE DELIVERABLE

1. Prerequisites

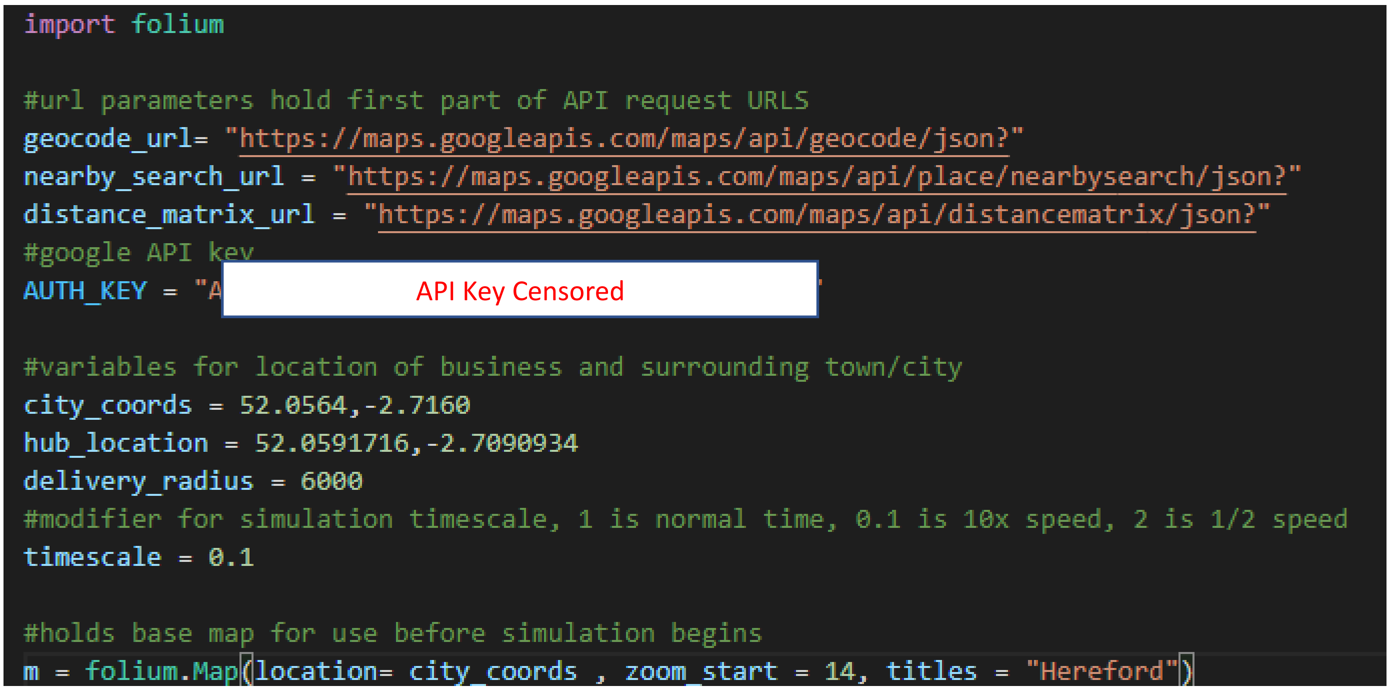
Before development of the system could begin, it was necessary to obtain an API key to be used on Google Cloud. To achieve this, an account was created, and its API key was stored within cfg as AUTH_KEY. The three APIs needed: geocoding, nearby search, and distance matrix had to all be manually enabled for use with the key.

Python 3.9.6 was installed, and PIP was used to install Folium and Flask, with the command 'pip install' followed by their names.

2. Creation of the Deliverable

a. cfg

The '**cfg**' module is an essential element of the system's design. Within this module, business- specific global variables are stored. These variables are used throughout the rest of the program, usually passed as parameters for any geographic function requiring a location.

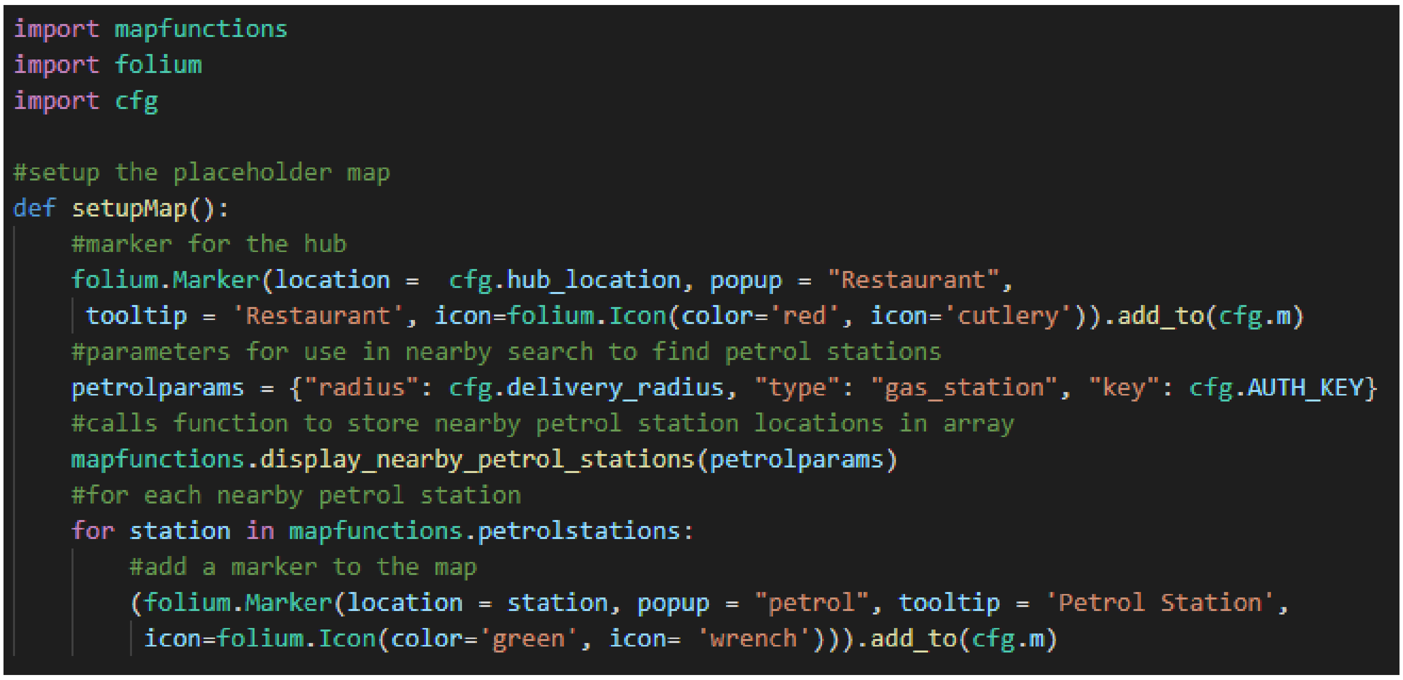


```
import folium

#url parameters hold first part of API request URLS
geocode_url= "https://maps.googleapis.com/maps/api/geocode/json?"
nearby_search_url = "https://maps.googleapis.com/maps/api/place/nearbysearch/json?"
distance_matrix_url = "https://maps.googleapis.com/maps/api/distancematrix/json?"
#google API key
AUTH_KEY = "A          API Key Censored          "

#variables for location of business and surrounding town/city
city_coords = 52.0564,-2.7160
hub_location = 52.0591716,-2.7090934
delivery_radius = 6000
#modifier for simulation timescale, 1 is normal time, 0.1 is 10x speed, 2 is 1/2 speed
timescale = 0.1

#holds base map for use before simulation begins
m = folium.Map(location= city_coords , zoom_start = 14, titles = "Hereford")
```

Cfg also stores the first part of the three API URLs that will be used to facilitate geographical queries within the system. This is partially to save space when typing out the URLs in their implementations, but also to allow for easy changing of API provider in case the service offered is no longer supported at a later date. Additionally, should a more cost-effective API provider be discovered, it would make it easy to switch to them. The **city_coords**, **hub_location**, and **delivery_radius** values are the only internal variables that would have to be changed to apply the system to a new location, all other business – specific data is taken from external files which can be fed into the system by managers. **Timescale** is a variable which will only be used for the purpose of running a simulation demonstrating the system in action using AI drivers, it can be changed to either speed up or slow down the pace of the simulation. Its default state of 1.0 results in a simulation using real-world timings for driving between locations.

b. mapsetup

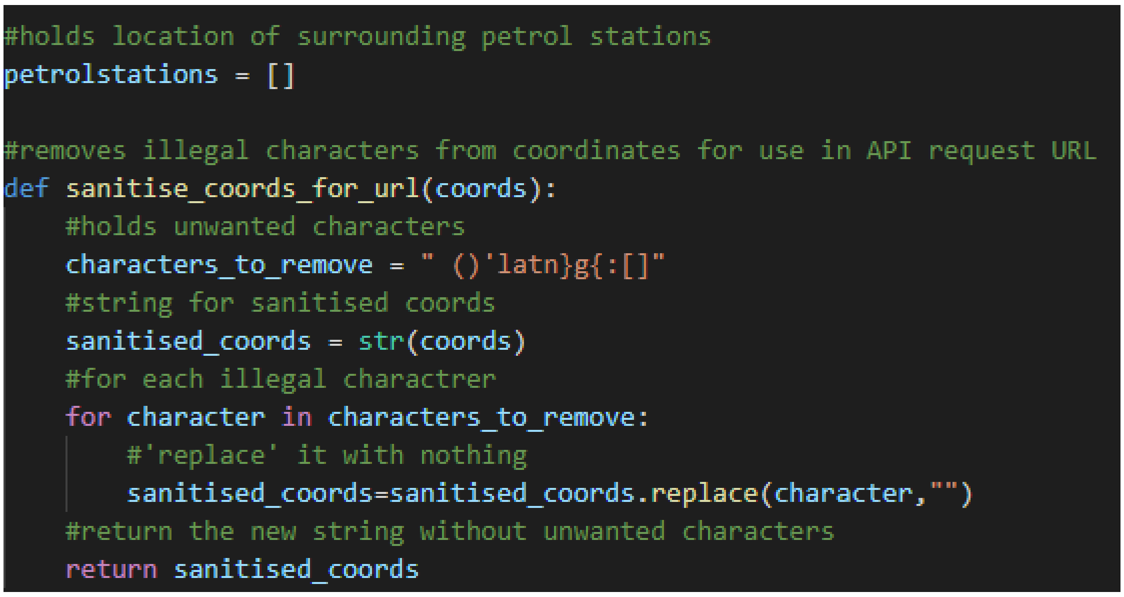
The **mapsetup** module contains only the **setupMap** method, which invokes the creation of a folium map featuring markers showing the position of various elements relevant to the business, such as the location of the business hub and all petrol stations within the delivery radius as defined in **cfg**. These petrol stations will be located through the use of a nearby search API query, which is handled by the **mapfunctions** module.



```python
import mapfunctions
import folium
import cfg

#setup the placeholder map
def setupMap():
    #marker for the hub
    folium.Marker(location =  cfg.hub_location, popup = "Restaurant",
     tooltip = 'Restaurant', icon=folium.Icon(color='red', icon='cutlery')).add_to(cfg.m)
    #parameters for use in nearby search to find petrol stations
    petrolparams = {"radius": cfg.delivery_radius, "type": "gas_station", "key": cfg.AUTH_KEY}
    #calls function to store nearby petrol station locations in array
    mapfunctions.display_nearby_petrol_stations(petrolparams)
    #for each nearby petrol station
    for station in mapfunctions.petrolstations:
        #add a marker to the map
        (folium.Marker(location = station, popup = "petrol", tooltip = 'Petrol Station',
         icon=folium.Icon(color='green', icon= 'wrench'))).add_to(cfg.m)
```

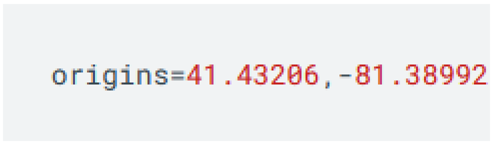
c. mapfunctions

mapfunctions is a very useful module containing a range of methods and functions relating to Folium and API queries. Its methods are frequently called by the other modules, and are key to providing accurate geographical information on which the rest of the system operates.



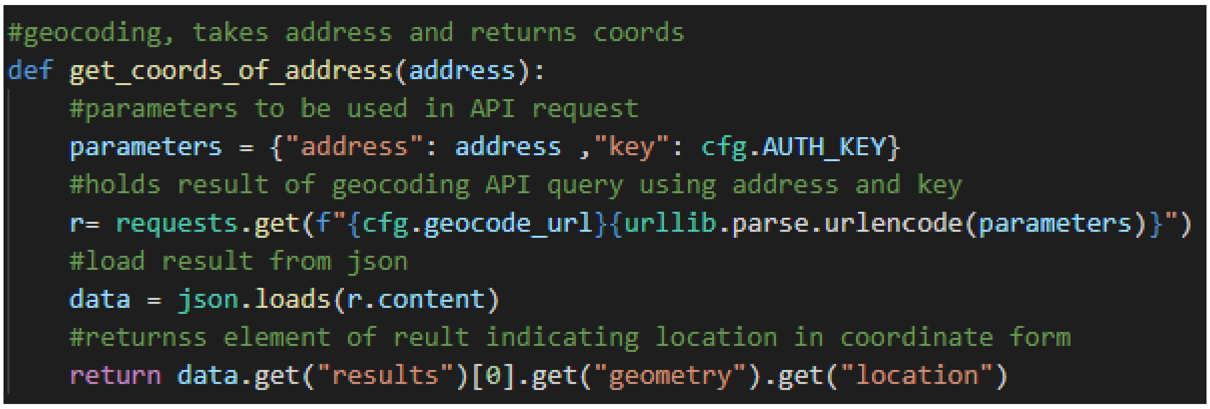
```python
#holds location of surrounding petrol stations
petrolstations = []

#removes illegal characters from coordinates for use in API request URL
def sanitise_coords_for_url(coords):
    #holds unwanted characters
    characters_to_remove = " ()'latn}g{:[]"
    #string for sanitised coords
    sanitised_coords = str(coords)
    #for each illegal charactrer
    for character in characters_to_remove:
        #'replace' it with nothing
        sanitised_coords=sanitised_coords.replace(character,"")
    #return the new string without unwanted characters
    return sanitised_coords
```

sanitise_coords_for_url is the module's first function. During development, a problem frequently arose where the program would attempt to make an API request using coordinates, however they would be passed in an illegal format. Google's API service requires coordinates to be passed as such:



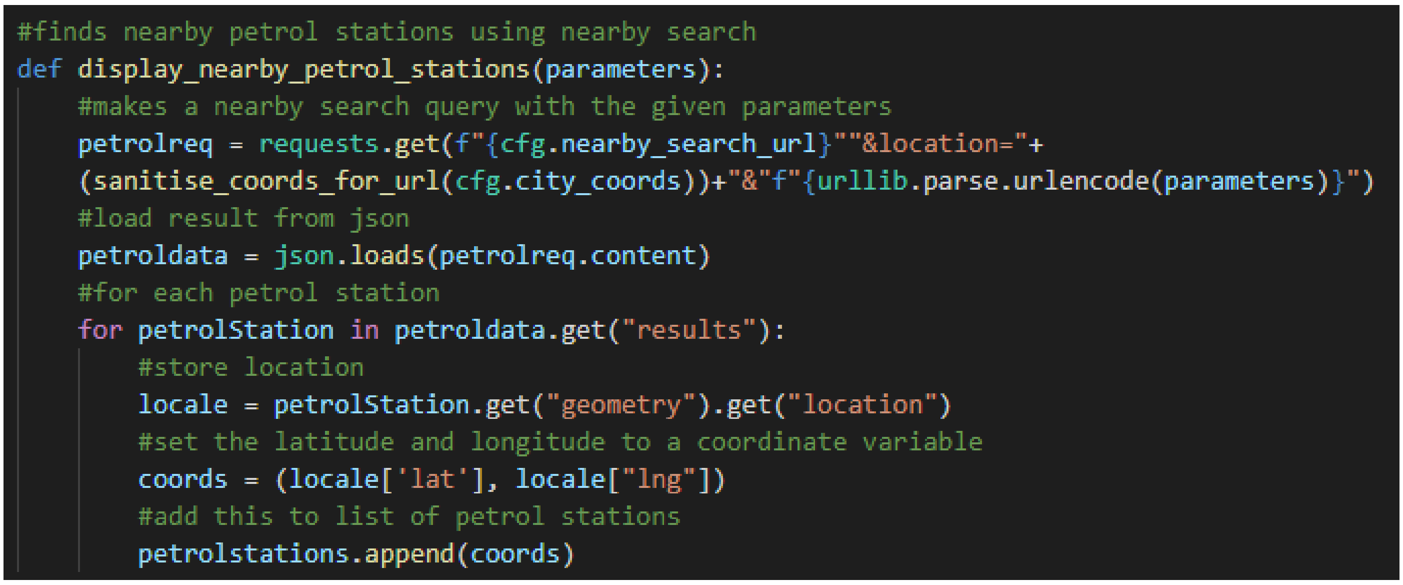
```
origins=41.43206,-81.38992
```

When getting coordinates from a geocode request, they would be returned in the format [lat:41.43206, lng: -81.382992]. Therefore, to pass the results of a geocode request straight into a distance matrix or nearby search, some sanitisation has to take place. **sanitise_coords_for_url** takes coordinates as a parameter, and then iterates through them searching for any of the illegal characters ()'latn}g{:[]. Should it discover one of these, it simply replaces it with nothing. The function then returns the freshly sanitised set of coordinates.



```python
#geocoding, takes address and returns coords
def get_coords_of_address(address):
    #parameters to be used in API request
    parameters = {"address": address ,"key": cfg.AUTH_KEY}
    #holds result of geocoding API query using address and key
    r= requests.get(f"{cfg.geocode_url}{urllib.parse.urlencode(parameters)}")
    #load result from json
    data = json.loads(r.content)
    #returnss element of reult indicating location in coordinate form
    return data.get("results")[0].get("geometry").get("location")
```

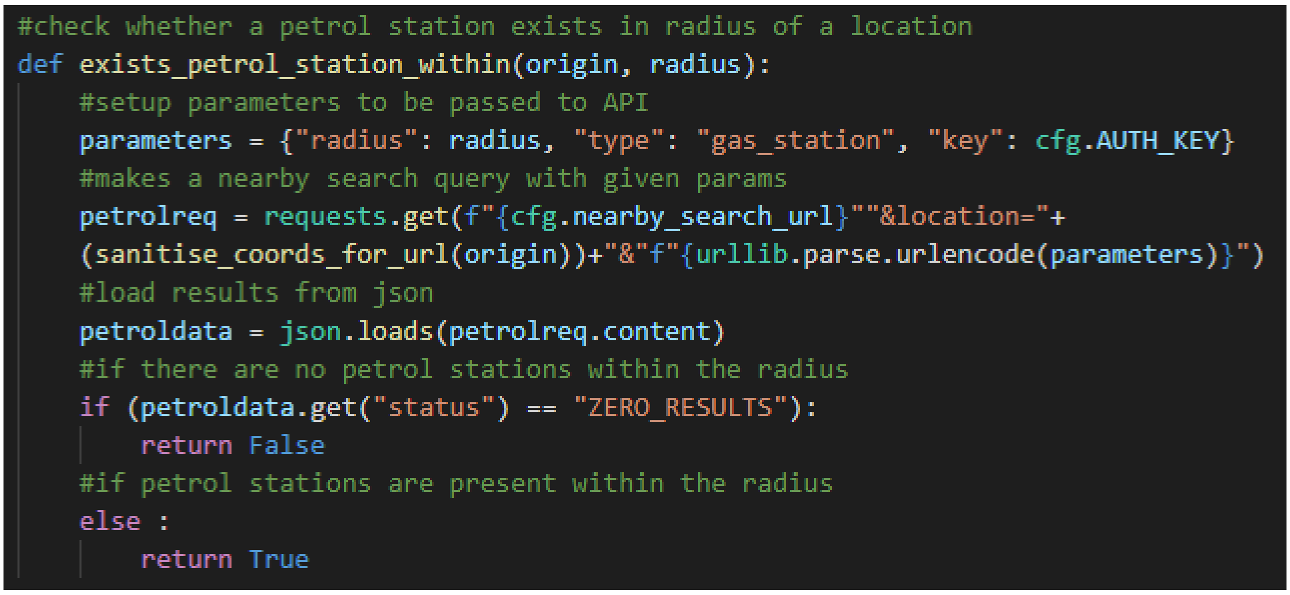
Geocoding is an essential part of the system, as customers ordering will provide their address but not the geographic coordinates needed for distance matrix queries. This function simply takes an address and passes it alongside the API key to Google's geocoding service. Notably, the packages urllib and requests are used for this functionality. Urllib's urlencode function converts the list of parameters into a query string, and requests allows for easy sending of HTTP requests. **get_coords_of_address** requests the result in JSON format, as this can easily be parsed to find the necessary element, which in this case is the 'location' value, nested inside 'geometry', which itself is part of 'results'.



```
#finds nearby petrol stations using nearby search
def display_nearby_petrol_stations(parameters):
    #makes a nearby search query with the given parameters
    petrolreq = requests.get(f"{cfg.nearby_search_url}""&location="+
    (sanitise_coords_for_url(cfg.city_coords))+"&"f"{urllib.parse.urlencode(parameters)}")
    #load result from json
    petroldata = json.loads(petrolreq.content)
    #for each petrol station
    for petrolStation in petroldata.get("results"):
        #store location
        locale = petrolStation.get("geometry").get("location")
        #set the latitude and longitude to a coordinate variable
        coords = (locale['lat'], locale["lng"])
        #add this to list of petrol stations
        petrolstations.append(coords)
```

display_nearby_petrol_stations is the first method to utilise the nearby search query. This method is to be called by **mapsetup** at system initialisation; it will find and store the coordinates of all petrol stations within the delivery radius. Whenever the map is rendered, Folium markers will be created for each petrol station and displayed on the map.

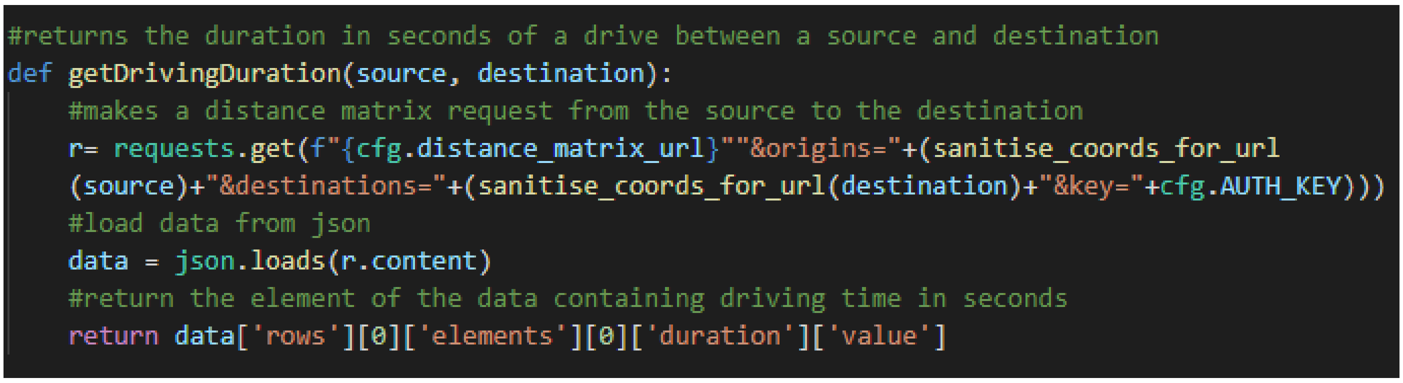
The nearby search query passes a sanitised version of **city_coords** as the origin, from which the search will begin. Within 'parameters' is the **delivery_radius,** the stipulation **type = gas_station** as well as the **AUTH_KEY**. The results are then iterated through by a for each loop, and each petrol station's coordinates are added to the list **petrolstations**. Whenever the map is rendered, instead of making new API queries, these locations will be reused to save on API billing charges as well as processor time.



```
#check whether a petrol station exists in radius of a location
def exists_petrol_station_within(origin, radius):
    #setup parameters to be passed to API
    parameters = {"radius": radius, "type": "gas_station", "key": cfg.AUTH_KEY}
    #makes a nearby search query with given params
    petrolreq = requests.get(f"{cfg.nearby_search_url}""&location="+
    (sanitise_coords_for_url(origin))+"&"f"{urllib.parse.urlencode(parameters)}")
    #load results from json
    petroldata = json.loads(petrolreq.content)
    #if there are no petrol stations within the radius
    if (petroldata.get("status") == "ZERO_RESULTS"):
        return False
    #if petrol stations are present within the radius
    else :
        return True
```

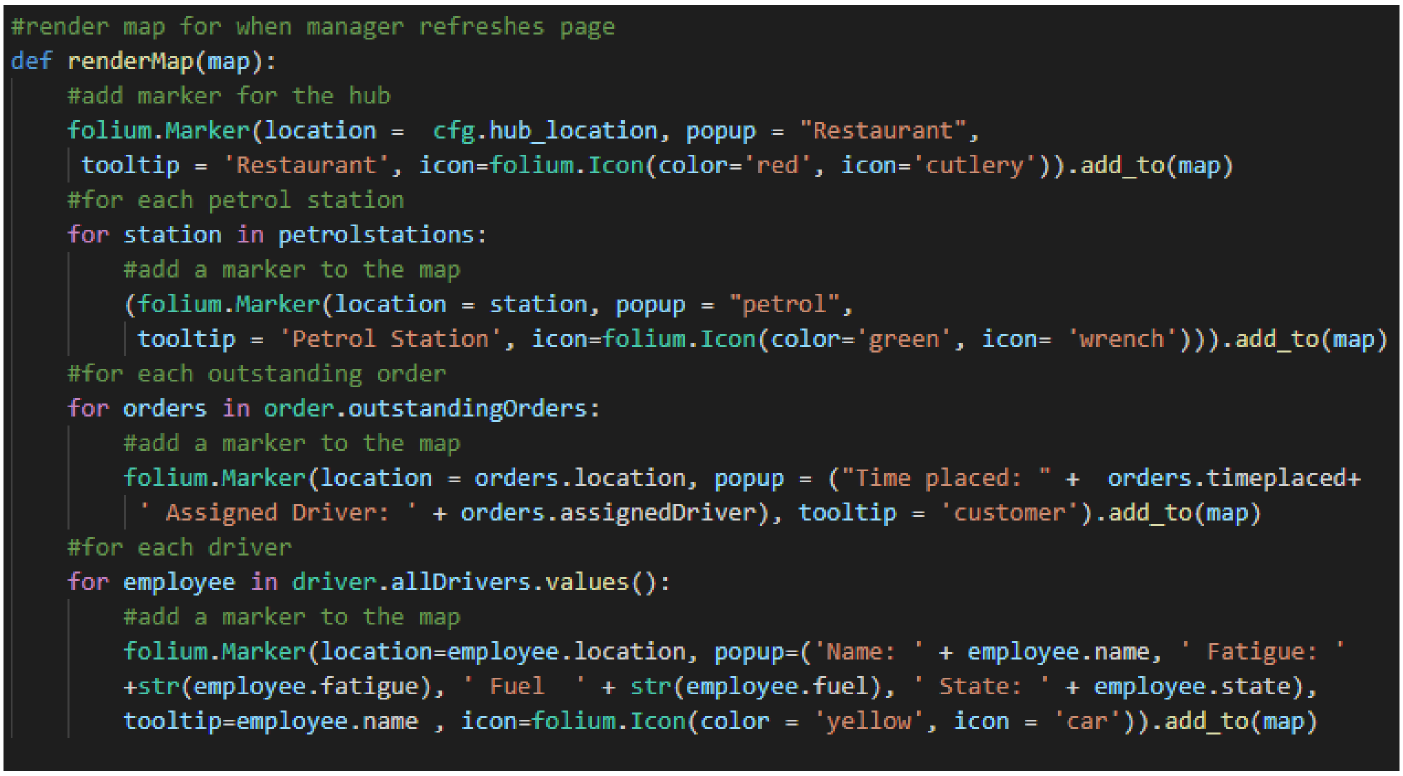
Another use of the nearby search API can be found in the **exists_petrol_station_within** function. This function will be used to determine whether there is a petrol station within close proximity to the location of an order. This is useful, as drivers who are running low on fuel can be prioritised for deliveries where a petrol station exists in close proximity.

The first part of this function is almost identical to **display_nearby_petrol_stations**, except it takes the origin as a parameter instead of using **city_coords.** Should the JSON returned by the API contain a status indicating 'ZERO_RESULTS', the function returns false, indicating that there are no petrol stations. Otherwise, True is returned. The use of this function will be demonstrated within **orderhandler.**



```python
#returns the duration in seconds of a drive between a source and destination
def getDrivingDuration(source, destination):
    #makes a distance matrix request from the source to the destination
    r= requests.get(f"{cfg.distance_matrix_url}""&origins="+(sanitise_coords_for_url
    (source)+"&destinations="+(sanitise_coords_for_url(destination)+"&key="+cfg.AUTH_KEY)))
    #load data from json
    data = json.loads(r.content)
    #return the element of the data containing driving time in seconds
    return data['rows'][0]['elements'][0]['duration']['value']
```

Since the system is focused on managing a workforce of drivers, it is important to be able to determine how long it would take drivers to travel between two points. This function provides this ability through its use of Google's Distance Matrix API. It simply takes a source and destination as parameters, and passes sanitised versions of these coordinates to the API. The result is returned as the time in seconds it takes to drive between the locations.



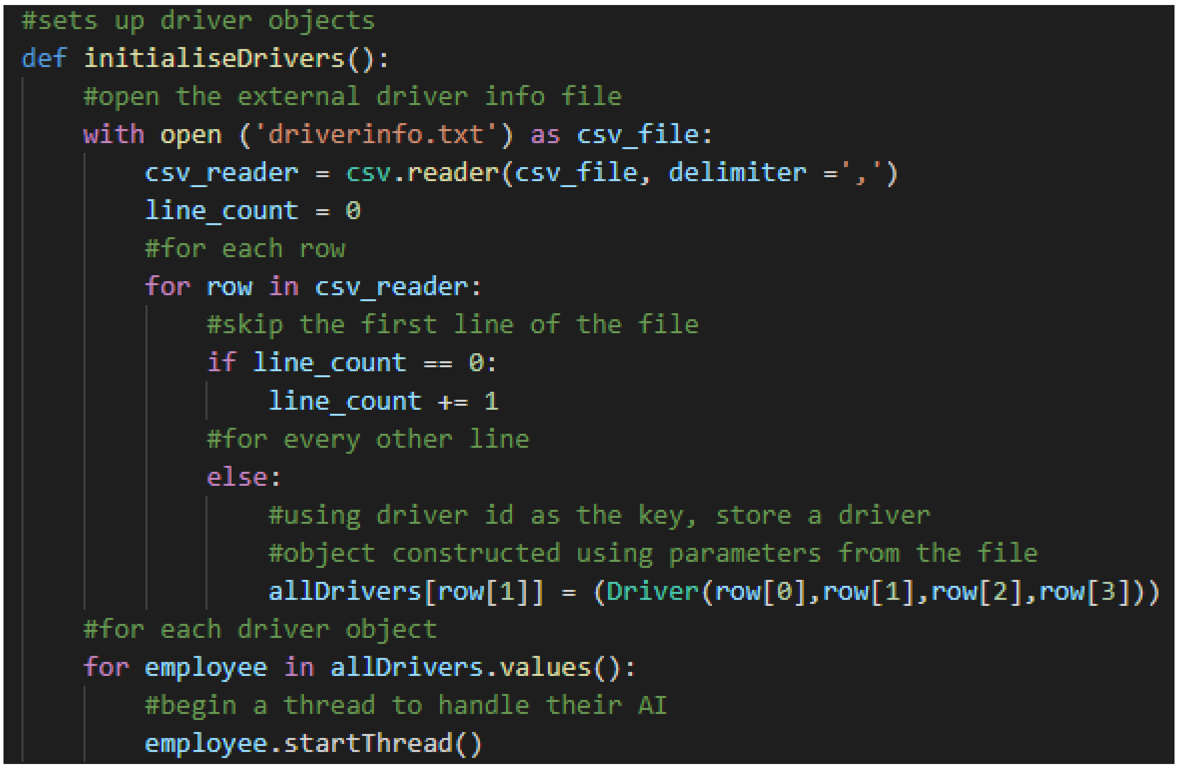
```python
#render map for when manager refreshes page
def renderMap(map):
    #add marker for the hub
    folium.Marker(location =  cfg.hub_location, popup = "Restaurant",
     tooltip = 'Restaurant', icon=folium.Icon(color='red', icon='cutlery')).add_to(map)
    #for each petrol station
    for station in petrolstations:
        #add a marker to the map
        (folium.Marker(location = station, popup = "petrol",
         tooltip = 'Petrol Station', icon=folium.Icon(color='green', icon= 'wrench'))).add_to(map)
    #for each outstanding order
    for orders in order.outstandingOrders:
        #add a marker to the map
        folium.Marker(location = orders.location, popup = ("Time placed: " +  orders.timeplaced+
         ' Assigned Driver: ' + orders.assignedDriver), tooltip = 'customer').add_to(map)
    #for each driver
    for employee in driver.allDrivers.values():
        #add a marker to the map
        folium.Marker(location=employee.location, popup=('Name: ' + employee.name, ' Fatigue: '
        +str(employee.fatigue), ' Fuel  ' + str(employee.fuel), ' State: ' + employee.state),
        tooltip=employee.name , icon=folium.Icon(color = 'yellow', icon = 'car')).add_to(map)
```

This method will be called whenever the map is refreshed, in order to provide an updated map. It is necessary to do this due to Folium not supporting the removal of markers, or the ability to animate markers. During development, the limitations of Folium proved to be rather restrictive; it was planned that the driver markers would update themselves every time they moved, however the inability to remove the markers indicating their previous position made this infeasible.

The location of the business hub is displayed, as well as the nearby petrol stations (taken from the **petrolstations** list). Every driver and outstanding order are represented with a marker and a relevant icon, and the 'popup' includes important information about each to be displayed when the marker is clicked, which the manager may wish to see when using the system.

d. driver

The driver module contains a dictionary **allDrivers** to hold each driver object, with the key as their id. This dictionary is populated when **initialiseDrivers** is ran, which is done during the system's start-up.

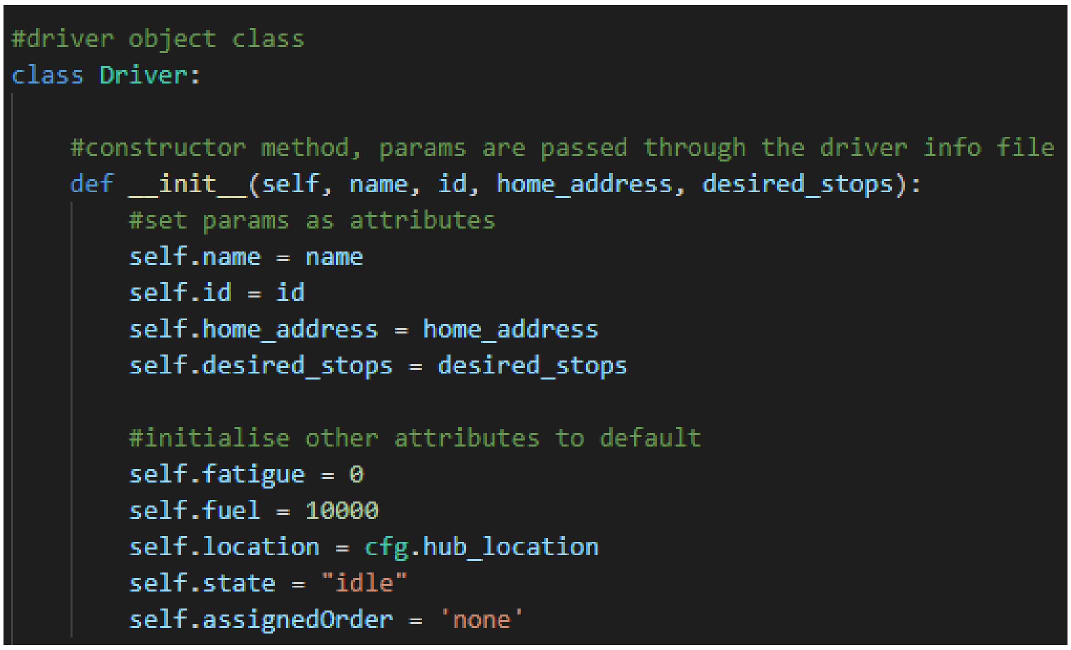


```python
#sets up driver objects
def initialiseDrivers():
    #open the external driver info file
    with open ('driverinfo.txt') as csv_file:
        csv_reader = csv.reader(csv_file, delimiter =',')
        line_count = 0
        #for each row
        for row in csv_reader:
            #skip the first line of the file
            if line_count == 0:
                line_count += 1
            #for every other line
            else:
                #using driver id as the key, store a driver
                #object constructed using parameters from the file
                allDrivers[row[1]] = (Driver(row[0],row[1],row[2],row[3]))
    #for each driver object
    for employee in allDrivers.values():
        #begin a thread to handle their AI
        employee.startThread()
```

initialiseDrivers takes information from the external file **driverinfo.txt**, which managers can substitute with their own documents containing important information about drivers. The csv reader will skip over the first line, as this line contains column names – in this case name, id, home_address, and desired_stops. For each row, a driver object is constructed with parameters set as the data read from the external file. After the drivers have all been constructed and added to the dictionary, each driver's **startThread** method is called. This will spawn a new thread to process each driver's AI.

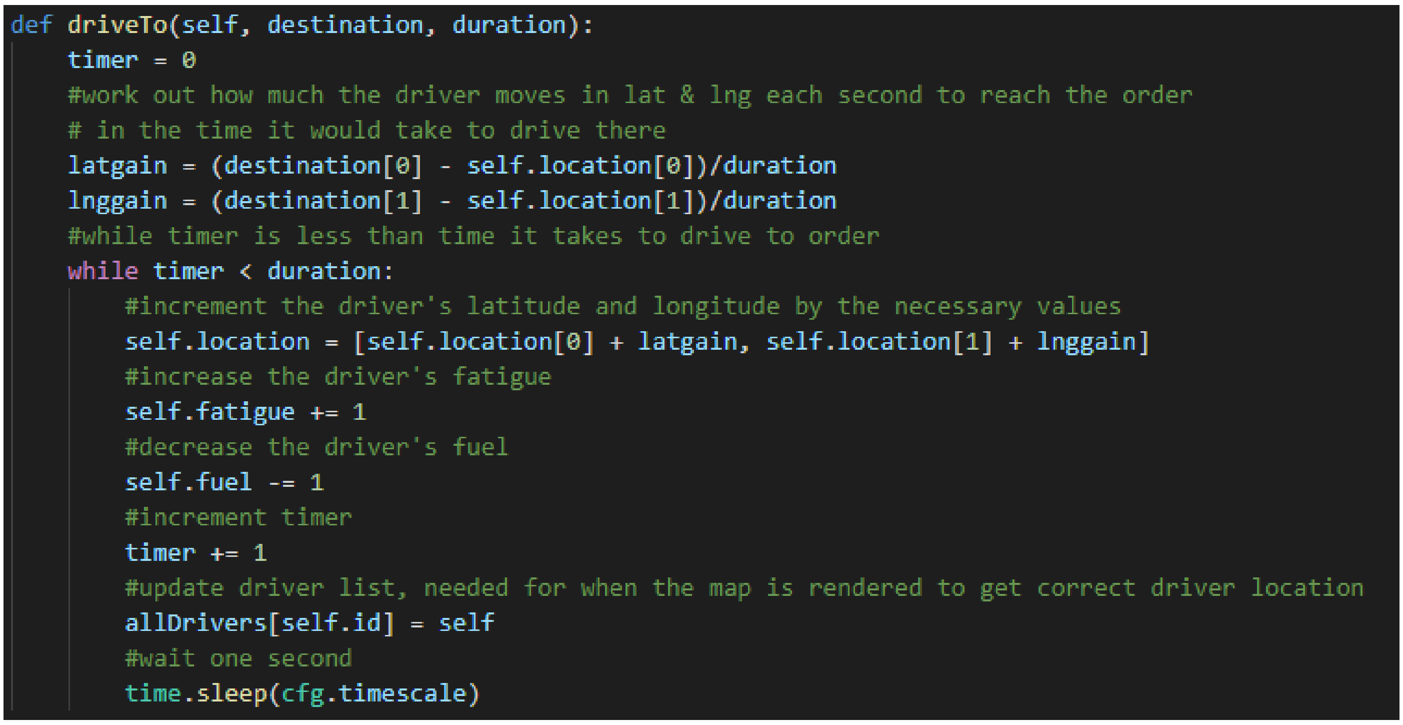
Multithreading is appropriate for this system, as every driver's AI needs to be running simultaneously. A single-threaded approach would be infeasible, as starting the AI of one driver would cause the other drivers' AIs to have their execution delayed until the first driver's process was terminated. Another solution offered by Python to this type of problem is multiprocessing, although in multiprocessing, the processes have separate memory. In this implementation, the driver threads all need to access shared data such as **outstandingOrders**, so multithreading's shared memory solution is appropriate.

Within multithreading, the different threads are not 'truly' running in parallel, instead the processor will service the next thread in the queue when it encounters downtime such as a **time.sleep** statement. In practice, the threads appear to be running simultaneously due to the speed at which the processor is able to execute commands prior to downtime.



```python
#driver object class
class Driver:

    #constructor method, params are passed through the driver info file
    def __init__(self, name, id, home_address, desired_stops):
        #set params as attributes
        self.name = name
        self.id = id
        self.home_address = home_address
        self.desired_stops = desired_stops

        #initialise other attributes to default
        self.fatigue = 0
        self.fuel = 10000
        self.location = cfg.hub_location
        self.state = "idle"
        self.assignedOrder = 'none'
```

__init__ is Python's keyword for a constructor method. Here, the Driver object class's constructor method is defined, and the parameters specified are taken from the **driverinfo.txt** file during **initialiseDrivers**. The second set of attributes are default values, as each driver should start with a full tank of fuel and zero fatigue. Their location will be identical to the business hub location, they will be in the idle state and have no assigned order.



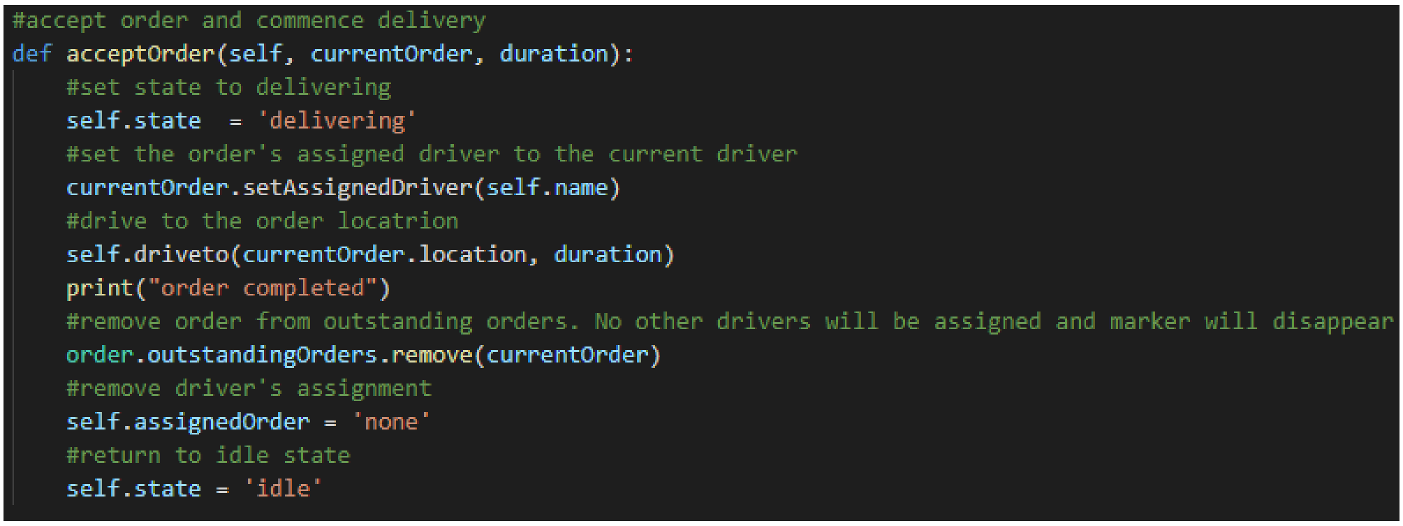
```python
def driveTo(self, destination, duration):
    timer = 0
    #work out how much the driver moves in lat & lng each second to reach the order
    # in the time it would take to drive there
    latgain = (destination[0] - self.location[0])/duration
    lnggain = (destination[1] - self.location[1])/duration
    #while timer is less than time it takes to drive to order
    while timer < duration:
        #increment the driver's latitude and longitude by the necessary values
        self.location = [self.location[0] + latgain, self.location[1] + lnggain]
        #increase the driver's fatigue
        self.fatigue += 1
        #decrease the driver's fuel
        self.fuel -= 1
        #increment timer
        timer += 1
        #update driver list, needed for when the map is rendered to get correct driver location
        allDrivers[self.id] = self
        #wait one second
        time.sleep(cfg.timescale)
```

driveto does exactly what it says on the tin: sets the driver on their way to a destination. This destination is provided as a set of coordinates. The **duration** parameter is how long in seconds it should take to drive to the destination as calculated by the distance matrix API. It was considered to include this calculation as part of the **driveto** method, however this would result in duplicate API queries in some scenarios, for example queued orders already have the distance for the driver calculated by the **orderhandler**.

A timer is initialised as a means of counting how long the driver has been driving to the destination, so that the driving process can be completed when the driver should have arrived at the destination. **Latgain** and **lnggain** function as a vector which is added to the driver's location, each of them is the difference between the driver's and customer's coordinate divided by the time it takes to drive there. Effectively, they represent how much the driver should move each second to arrive at the destination at the time estimated by the distance matrix API.

It is worth noting that this solution is purely for the AI representation of the system; in a real-world implementation it would be more accurate and less computationally expensive to set the driver's location to coordinates returned by a GPS affixed to their vehicle. For demonstration purposes, this solution gives a rough approximation of where a driver should be, and a visible indication of their progress towards a destination.

Each tick of the timer, the driver's location is incremented by the vector, their fatigue increases and fuel decreases. **allDrivers** is also updated to reflect the current attributes of the driver, so that if a manager refreshes the map during a delivery, the location shown by the driver's marker is up to date.

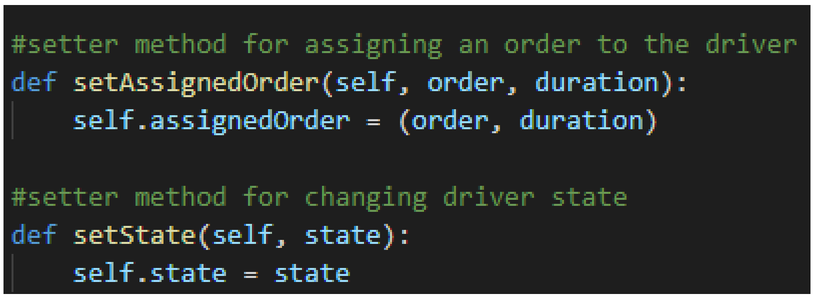


```
#accept order and commence delivery
def acceptOrder(self, currentOrder, duration):
    #set state to delivering
    self.state  = 'delivering'
    #set the order's assigned driver to the current driver
    currentOrder.setAssignedDriver(self.name)
    #drive to the order locatrion
    self.driveto(currentOrder.location, duration)
    print("order completed")
    #remove order from outstanding orders. No other drivers will be assigned and marker will disappear
    order.outstandingOrders.remove(currentOrder)
    #remove driver's assignment
    self.assignedOrder = 'none'
    #return to idle state
    self.state = 'idle'
```

The **acceptOrder** method will be called whenever the driver is assigned an order. Self is passed as a parameter referring to the driver themselves, so any changes to attributes will be reflected in the corresponding driver alone. The other two parameters are the order object as well as the duration – the time it will take the driver to reach the order from their current location.

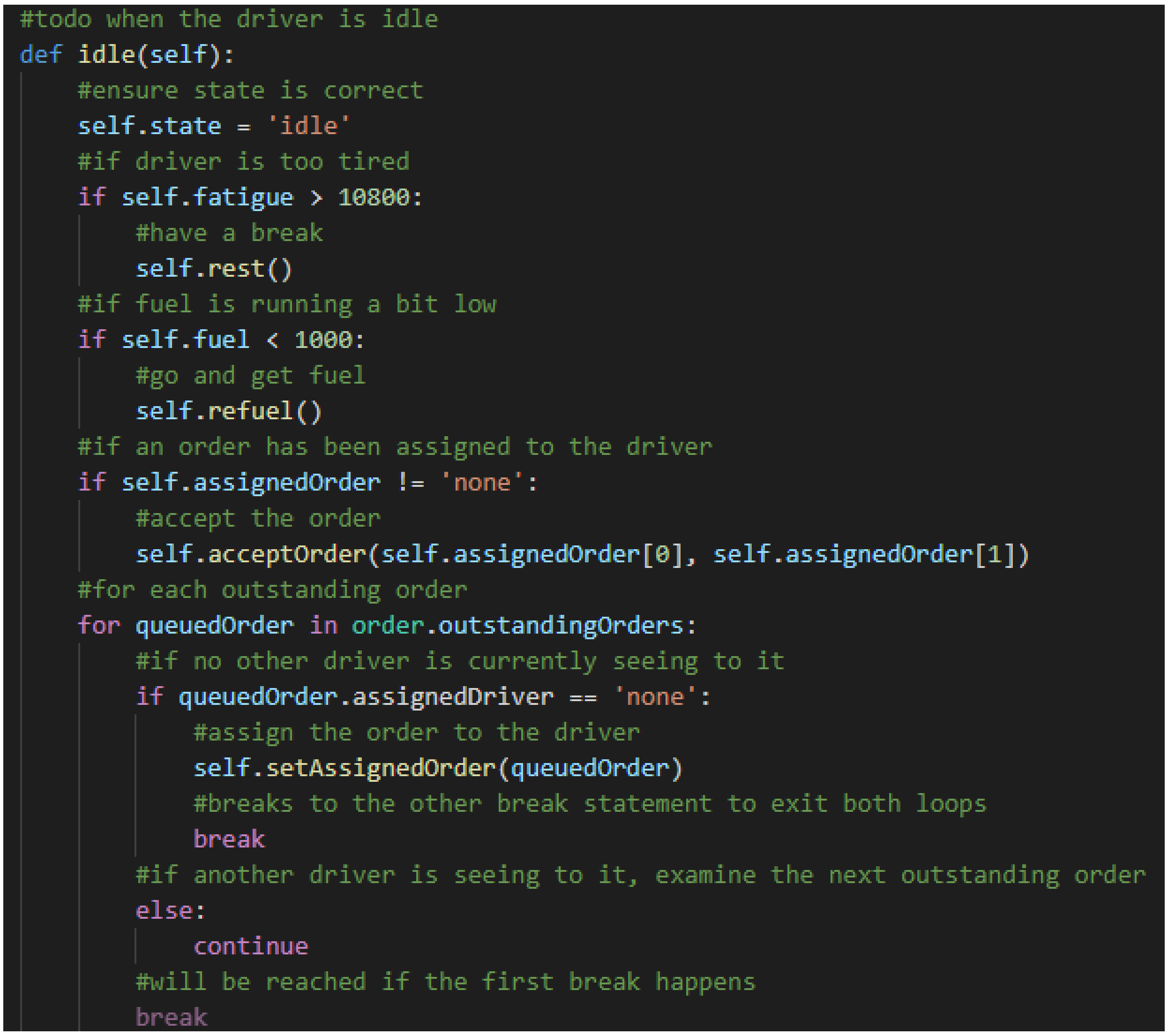
First, the driver's state is set to delivering. This is so that any other processes can see that the driver is occupied, and will not assign other tasks to them. The order object's assigned driver is also set to the driver's name, this is so that other drivers will not accept the order and, if the map is rendered, the manager can click on the order's marker and see which driver is assigned to the order. Next, the driver is sent on their way to the delivery through the use of the **driveto** method.

After the completion of a delivery, the order object is removed from **outstandingOrders** to prevent it being assigned to another driver. The driver's attributes are updated accordingly: they no longer have an assigned order and can return to the 'idle' state.



```python
#setter method for assigning an order to the driver
def setAssignedOrder(self, order, duration):
    self.assignedOrder = (order, duration)

#setter method for changing driver state
def setState(self, state):
    self.state = state
```

Two setter methods are defined for the Driver object, for the **assignedOrder** and **state** attributes. These attributes will need to be changed by the **orderhandler**, yet they are private by default, necessitating setter methods.



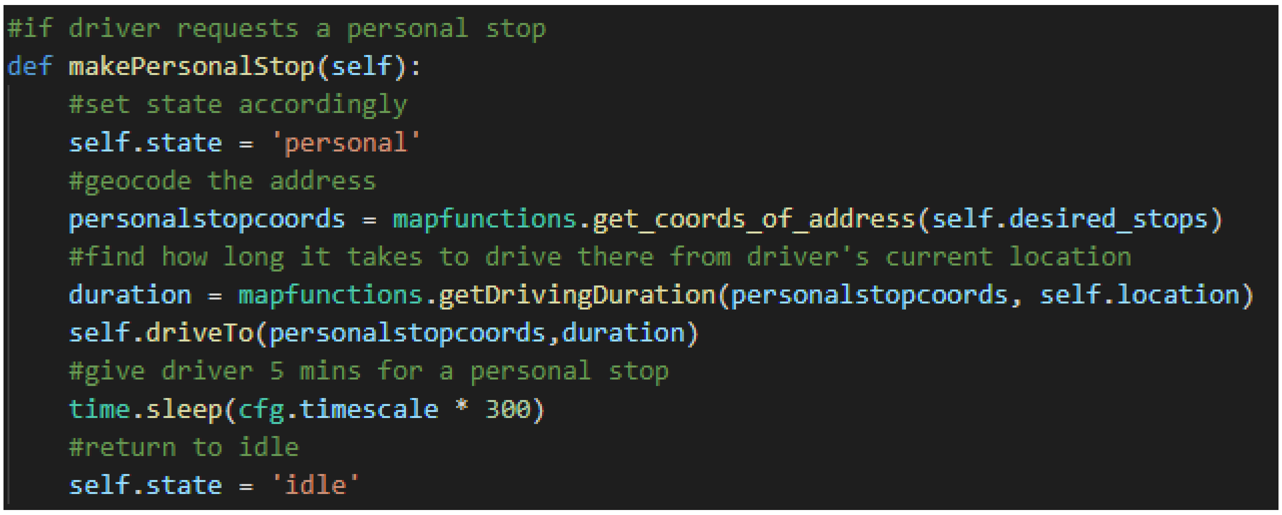
```python
#todo when the driver is idle
def idle(self):
    #ensure state is correct
    self.state = 'idle'
    #if driver is too tired
    if self.fatigue > 10800:
        #have a break
        self.rest()
    #if fuel is running a bit low
    if self.fuel < 1000:
        #go and get fuel
        self.refuel()
    #if an order has been assigned to the driver
    if self.assignedOrder != 'none':
        #accept the order
        self.acceptOrder(self.assignedOrder[0], self.assignedOrder[1])
    #for each outstanding order
    for queuedOrder in order.outstandingOrders:
        #if no other driver is currently seeing to it
        if queuedOrder.assignedDriver == 'none':
            #assign the order to the driver
            self.setAssignedOrder(queuedOrder)
            #breaks to the other break statement to exit both loops
            break
        #if another driver is seeing to it, examine the next outstanding order
        else:
            continue
        #will be reached if the first break happens
        break
```

The **idle** method is a series of checks to be carried out every second whilst the driver is in the 'idle' state, each of which is designed to check whether there is another task they could be doing instead of idling. The most important element is driver fatigue, owing to health and safety reasons and also the 'human-centered' aspect of the system. Even if a driver has not requested a break, the system will still ensure that they have a break should their fatigue reach over 10800. This reflects three hours of work, as this is the ideal break interval as identified during background research (Tucker, 2003).

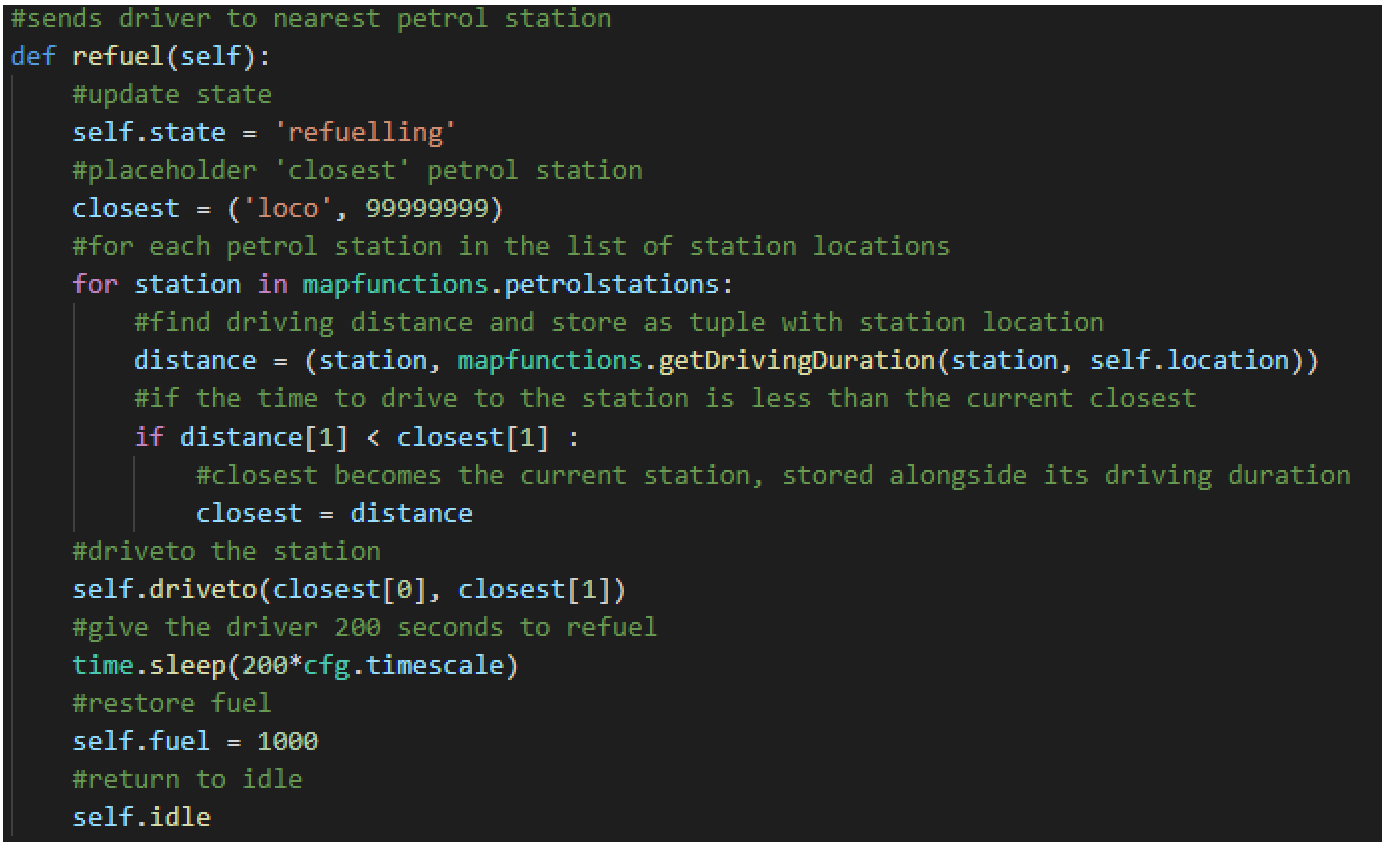
To ensure that drivers have enough petrol, drivers will be sent to refuel should their fuel readings drop below 10%. When their fuel levels are below 15%, the **orderhandler** will start to prioritise them for deliveries which are nearby fuel stations, allowing for a minimal detour when refuelling.

Any orders which have been assigned to the driver will be accepted, and the **acceptOrder** method will be invoked for whatever order object was set to the driver's **assignedOrder.** Finally, should all of these checks be passed without invoking a different method, **idle** will examine the list of **outstandingOrders** to see if any should be assigned to the driver. Each order's **assignedDriver** is checked, and if it is 'none', then the driver will take the order.

To account for the case of **outstandingOrders** containing more than one unassigned order, a slightly unorthodox use of break has been employed. Python does not provide an easy way to break from a nested loop, however a workaround is possible by using an else: continue statement. In this way, once an order is found with no driver, the order will be assigned to the current driver, and then the break statement is triggered. This break statement means that the else: continue is not executed, and the interpreter moves on to the second break statement, exiting the for loop**.** Interestingly, the IDE fades the colour of the second break due to it being 'unreachable', however this is not the case.

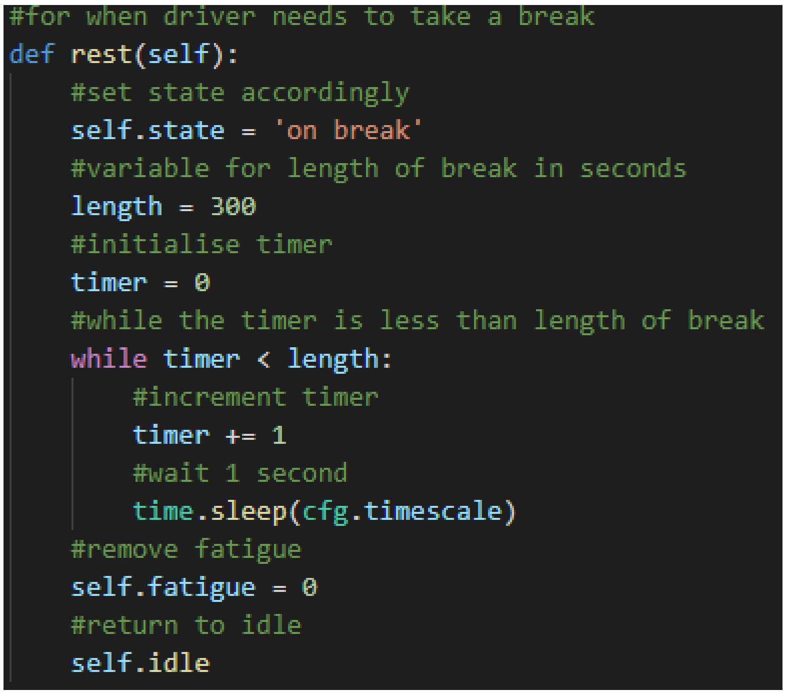

```
#if driver requests a personal stop
def makePersonalStop(self):
    #set state accordingly
    self.state = 'personal'
    #geocode the address
    personalstopcoords = mapfunctions.get_coords_of_address(self.desired_stops)
    #find how long it takes to drive there from driver's current location
    duration = mapfunctions.getDrivingDuration(personalstopcoords, self.location)
    self.driveTo(personalstopcoords,duration)
    #give driver 5 mins for a personal stop
    time.sleep(cfg.timescale * 300)
    #return to idle
    self.state = 'idle'
```

The method **makePersonalStop** will be invoked only by the driver themselves. Allowing drivers the freedom to make short personal stops during downtime is an important part of the system's design; drivers having this freedom contributes to their self-efficacy, an aspect tied to improving psychological capital (Luthans, 2015).



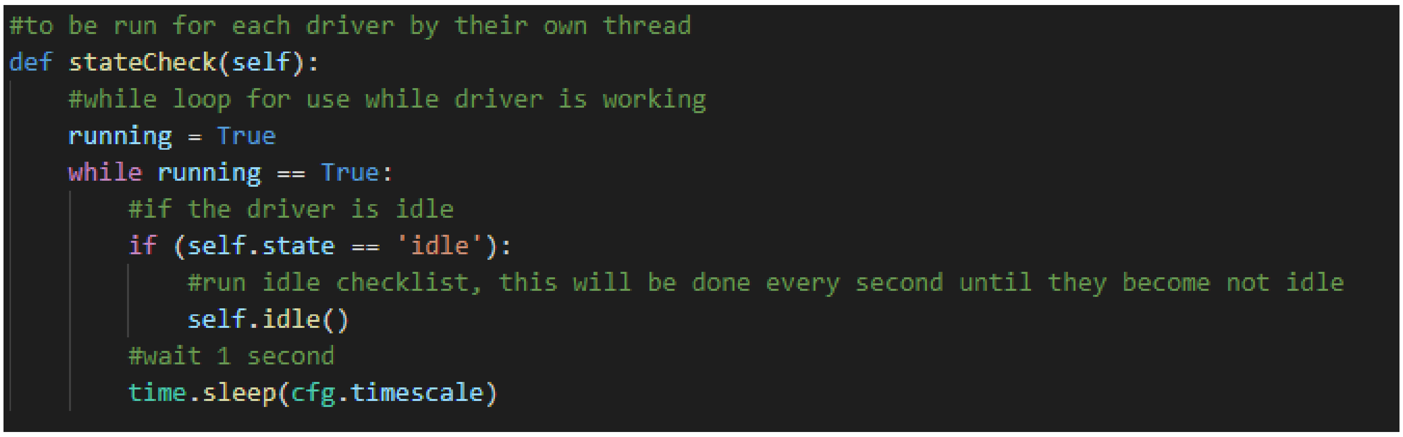
```
#sends driver to nearest petrol station
def refuel(self):
    #update state
    self.state = 'refuelling'
    #placeholder 'closest' petrol station
    closest = ('loco', 99999999)
    #for each petrol station in the list of station locations
    for station in mapfunctions.petrolstations:
        #find driving distance and store as tuple with station location
        distance = (station, mapfunctions.getDrivingDuration(station, self.location))
        #if the time to drive to the station is less than the current closest
        if distance[1] < closest[1] :
            #closest becomes the current station, stored alongside its driving duration
            closest = distance
    #driveto the station
    self.driveto(closest[0], closest[1])
    #give the driver 200 seconds to refuel
    time.sleep(200*cfg.timescale)
    #restore fuel
    self.fuel = 1000
    #return to idle
    self.idle
```

When drivers are refuelling, it makes sense that they should go to the nearest fuel station. **Refuel** first finds the coordinates of the nearest station, then routes the driver accordingly. **Closest** is initialised to a dummy value holding a very high driving duration. A for each loop iterates through **petrolstations,** finding the driving duration between the driver and the station. If it finds a petrol station which is a shorter drive away than the current shortest, it will make said station the new shortest. Once the loop completes, the driver will drive to the petrol station at **shortest**. Their fuel is refilled and time is given to account for filling up, before resetting the driver to idle.



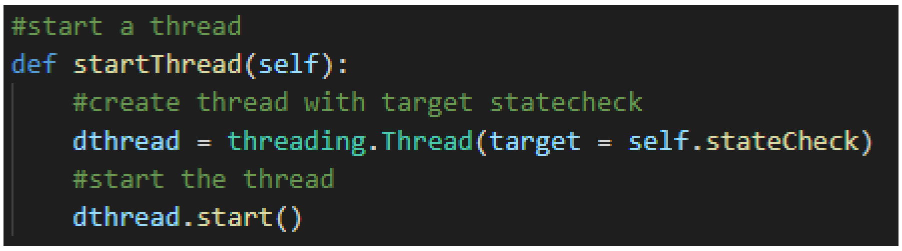
```
#for when driver needs to take a break
def rest(self):
    #set state accordingly
    self.state = 'on break'
    #variable for length of break in seconds
    length = 300
    #initialise timer
    timer = 0
    #while the timer is less than length of break
    while timer < length:
        #increment timer
        timer += 1
        #wait 1 second
        time.sleep(cfg.timescale)
    #remove fatigue
    self.fatigue = 0
    #return to idle
    self.idle
```

Resting will be triggered by either the driver's fatigue reaching a predetermined threshold, or the driver requesting a break themselves. There's nothing too fancy here, just a 5-minute timer running while a driver's state is 'on break' so that other activities cannot interfere. After the break, the driver's fatigue is removed and they are returned to idle.



```
#to be run for each driver by their own thread
def stateCheck(self):
    #while loop for use while driver is working
    running = True
    while running == True:
        #if the driver is idle
        if (self.state == 'idle'):
            #run idle checklist, this will be done every second until they become not idle
            self.idle()
        #wait 1 second
        time.sleep(cfg.timescale)
```

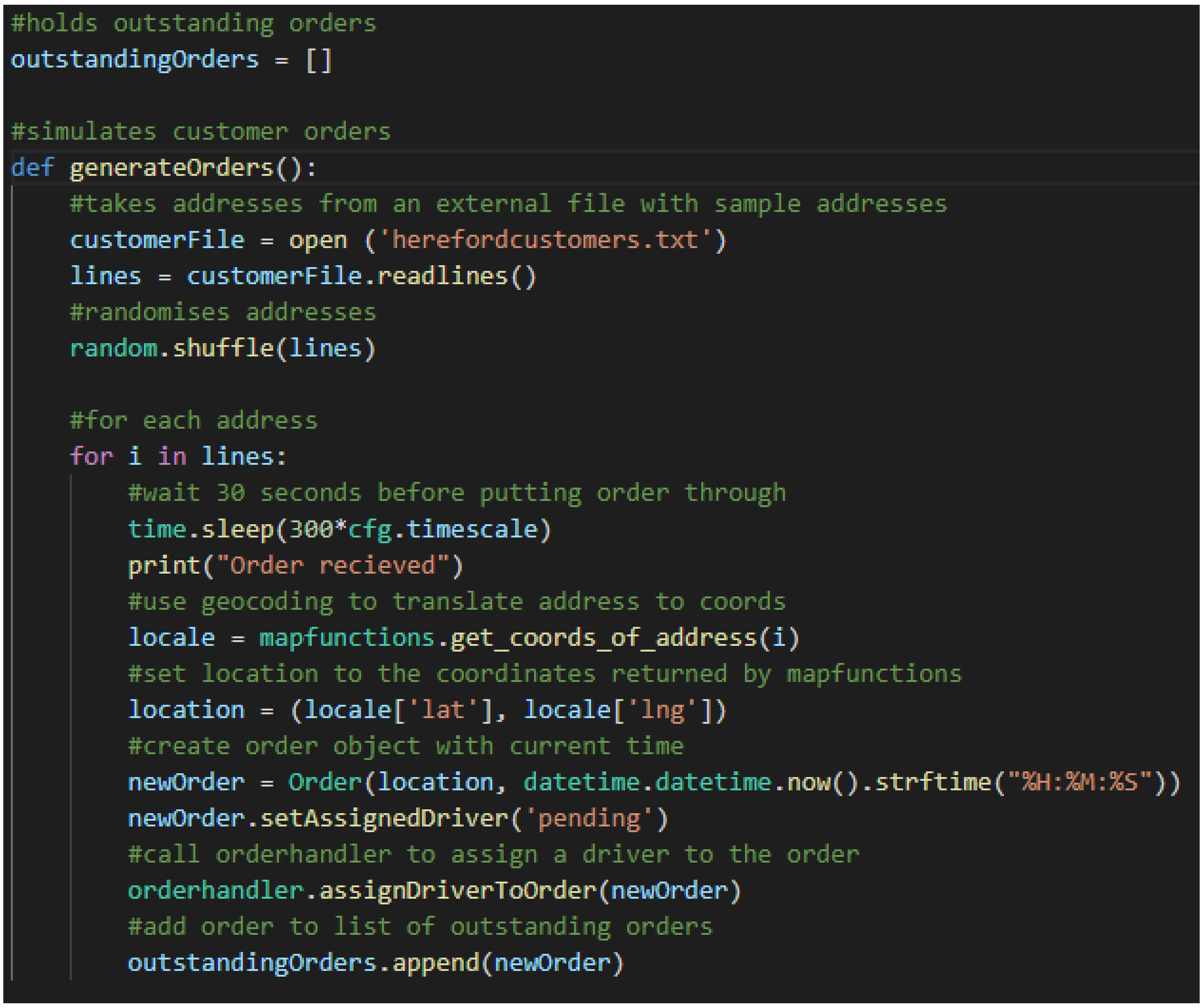
Each driver's thread is set to run **stateCheck**. This method is an infinite loop which will run the **idle** checklist when drivers are in the idle state, once per second. If the driver is not idle, it will do nothing to allow them to complete their current task. Since drivers revert to idle at the end of each task, this loop will run after any task's completion.



```
#start a thread
def startThread(self):
    #create thread with target statecheck
    dthread = threading.Thread(target = self.stateCheck)
    #start the thread
    dthread.start()
```

Finally, this method is invoked for each driver by **initialiseDrivers**. A new thread is spawned, set to target **stateCheck**, and started.

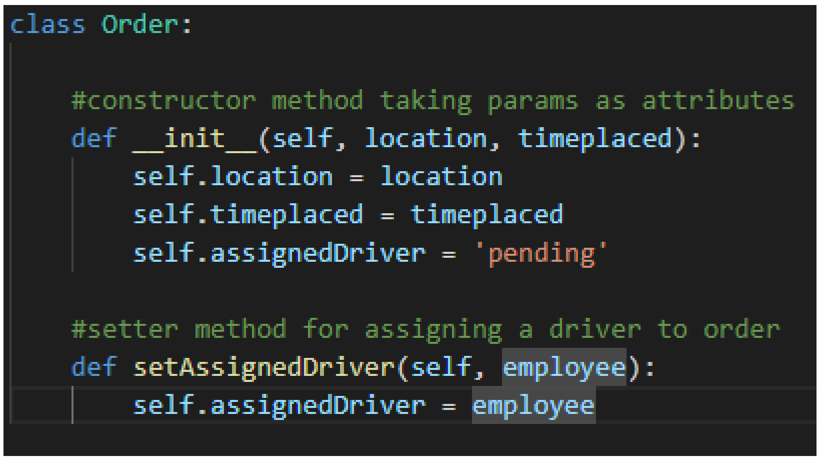
e. Order



```python
#holds outstanding orders
outstandingOrders = []

#simulates customer orders
def generateOrders():
    #takes addresses from an external file with sample addresses
    customerFile = open ('herefordcustomers.txt')
    lines = customerFile.readlines()
    #randomises addresses
    random.shuffle(lines)

    #for each address
    for i in lines:
        #wait 30 seconds before putting order through
        time.sleep(300*cfg.timescale)
        print("Order recieved")
        #use geocoding to translate address to coords
        locale = mapfunctions.get_coords_of_address(i)
        #set location to the coordinates returned by mapfunctions
        location = (locale['lat'], locale['lng'])
        #create order object with current time
        newOrder = Order(location, datetime.datetime.now().strftime("%H:%M:%S"))
        newOrder.setAssignedDriver('pending')
        #call orderhandler to assign a driver to the order
        orderhandler.assignDriverToOrder(newOrder)
        #add order to list of outstanding orders
        outstandingOrders.append(newOrder)
```

Outside of the **Order** object class, the **order** module has a list to hold outstanding orders, as well as the **generateOrders** function which is used for simulation purposes to model customers ordering from the business. An external file with addresses around Hereford is used, as this small city was used to test the system.

For each line in the file, the address will be geocoded by **mapfunctions** and an **Order** object instantiated using the coordinates returned alongside the time at which it was placed, courtesy of Pythons **datetime** package. The order's **assignedDriver** is set to 'pending', so that two drivers will not accept the same order. Only once **orderhandler** has attempted (and failed) to find a driver to assign the order to can the **assignedDriver** be set to 'none', and the order become 'up for grabs' by any idle drivers. Finally, the order is added to **outstandingOrders**, to be removed once it is completed.

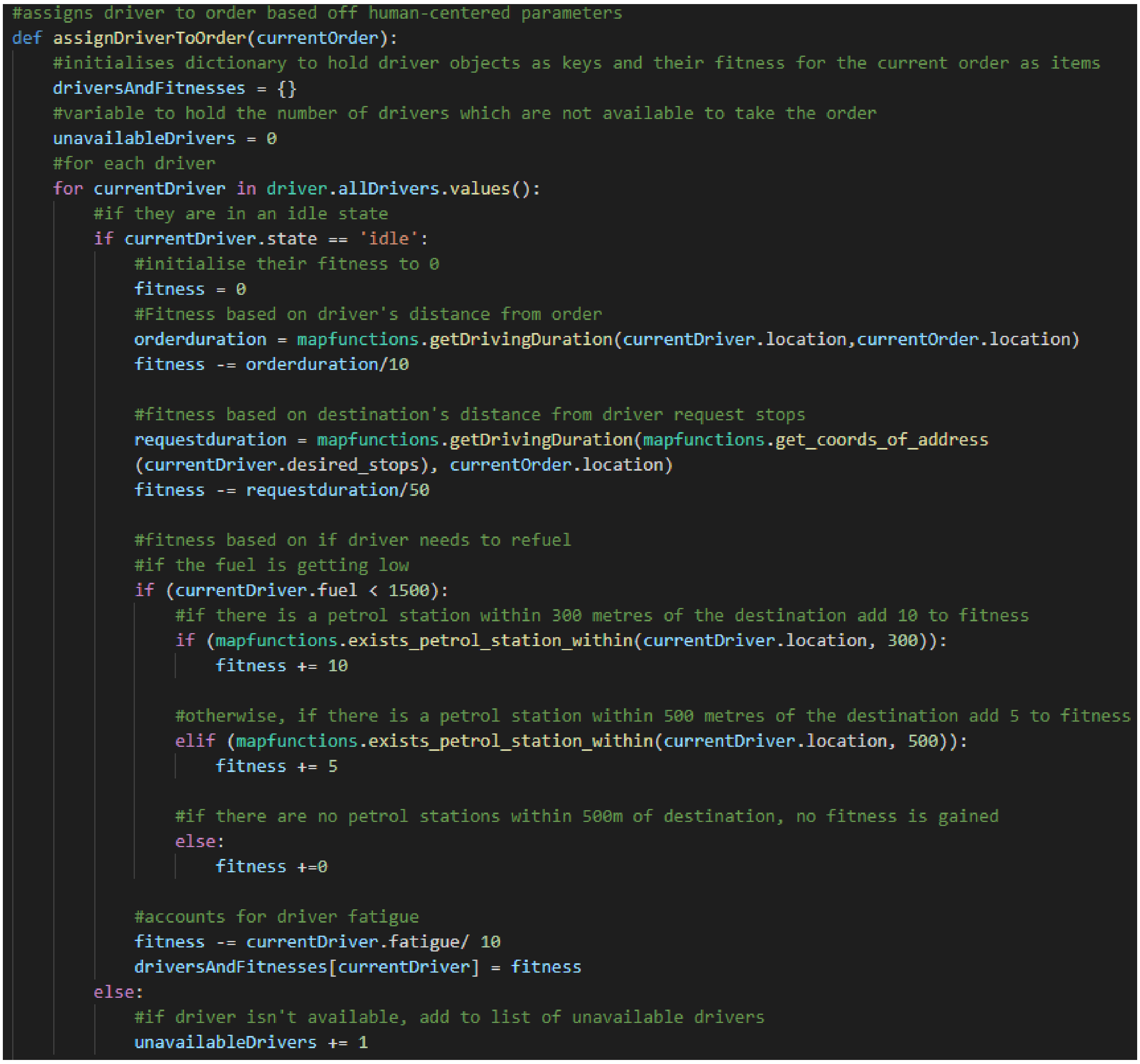


```python
class Order:

    #constructor method taking params as attributes
    def __init__(self, location, timeplaced):
        self.location = location
        self.timeplaced = timeplaced
        self.assignedDriver = 'pending'

    #setter method for assigning a driver to order
    def setAssignedDriver(self, employee):
        self.assignedDriver = employee
```

The **Order** object class is fairly uncomplicated, including a constructor method with a default assignment, as well as a setter class for **assignedDriver**.

f. orderhandler

orderhandler contains only one method: **assignDriverToOrder**.



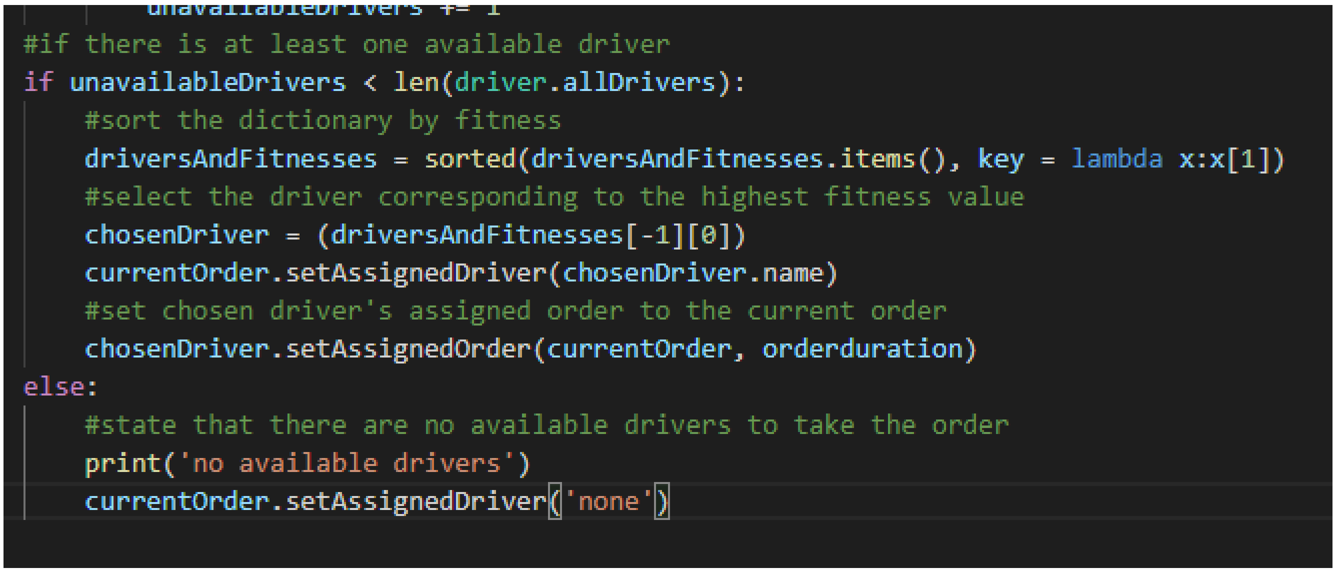
```
#assigns driver to order based off human-centered parameters
def assignDriverToOrder(currentOrder):
    #initialises dictionary to hold driver objects as keys and their fitness for the current order as items
    driversAndFitnesses = {}
    #variable to hold the number of drivers which are not available to take the order
    unavailableDrivers = 0
    #for each driver
    for currentDriver in driver.allDrivers.values():
        #if they are in an idle state
        if currentDriver.state == 'idle':
            #initialise their fitness to 0
            fitness = 0
            #Fitness based on driver's distance from order
            orderduration = mapfunctions.getDrivingDuration(currentDriver.location,currentOrder.location)
            fitness -= orderduration/10

            #fitness based on destination's distance from driver request stops
            requestduration = mapfunctions.getDrivingDuration(mapfunctions.get_coords_of_address
            (currentDriver.desired_stops), currentOrder.location)
            fitness -= requestduration/50

            #fitness based on if driver needs to refuel
            #if the fuel is getting low
            if (currentDriver.fuel < 1500):
                #if there is a petrol station within 300 metres of the destination add 10 to fitness
                if (mapfunctions.exists_petrol_station_within(currentDriver.location, 300)):
                    fitness += 10

                #otherwise, if there is a petrol station within 500 metres of the destination add 5 to fitness
                elif (mapfunctions.exists_petrol_station_within(currentDriver.location, 500)):
                    fitness += 5

                #if there are no petrol stations within 500m of destination, no fitness is gained
                else:
                    fitness +=0

            #accounts for driver fatigue
            fitness -= currentDriver.fatigue/ 10
            driversAndFitnesses[currentDriver] = fitness
        else:
            #if driver isn't available, add to list of unavailable drivers
            unavailableDrivers += 1
```

The main objective of this method is to decide which driver to assign an order to, based on many factors. The heart of the system relies on this method. First, a dictionary **driversAndFitnesses** is created, to store driver objects alongside their ‘fitness’ score for a given order. A variable holding the number of unavailable drivers is also initialised, for use in case all drivers are busy with other tasks.

A for each loop iterates through the driver objects in the valueset of **allDrivers**, and checks whether they are 'idle'. The driver's fitness is initialised to 0, then some calculations are made to determine their fitness. First, the time it would take the driver to reach the order is considered. This consideration is informed using **getDrivingDuration**. Another consideration is made based upon the distance of the order from the driver's desired stops. If the order address is close to somewhere a driver wanted to make a personal stop, then the driver's fitness to this order is increased.

Fuel is another consideration: any drivers with less than 15% fuel have their fitness boosted for orders which take them close to petrol stations. Using the **exists_petrol_station_within** function, addresses with orders within 300m of a station are boosted by 10 fitness, those within 500m are boosted by 5, and those with no stations within 500m do not receive a boost.

More fatigued drivers are of lower priority for being assigned orders, this is to take the pressure off them while drivers who have had a break more recently pick up the slack. Once the calculations have been made, an entry is made into **driversAndFitnesses** with the driver object as the key, and their fitness as the value.



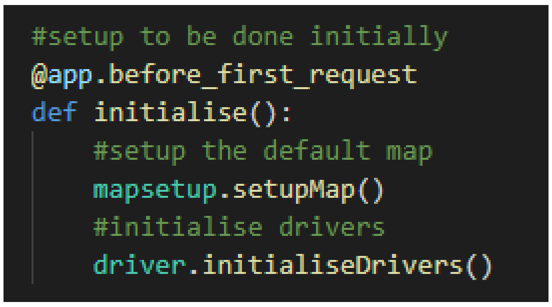
```
        unavailableDrivers += 1
#if there is at least one available driver
if unavailableDrivers < len(driver.allDrivers):
    #sort the dictionary by fitness
    driversAndFitnesses = sorted(driversAndFitnesses.items(), key = lambda x:x[1])
    #select the driver corresponding to the highest fitness value
    chosenDriver = (driversAndFitnesses[-1][0])
    currentOrder.setAssignedDriver(chosenDriver.name)
    #set chosen driver's assigned order to the current order
    chosenDriver.setAssignedOrder(currentOrder, orderduration)
else:
    #state that there are no available drivers to take the order
    print('no available drivers')
    currentOrder.setAssignedDriver('none')
```

Should all drivers be unavailable, **unavailableDrivers** will be equal to the length of **allDrivers**, and a message will display so that the manager knows there are no drivers available to fulfil the order. The order's **assignedDriver** is set to 'none' effectively adding it to a queue of orders. The next idle driver will discover this order and have it assigned to them.

If at least one driver is available to take the order, **driversAndFitnesses** is sorted in ascending order of its values, in this case the fitness of each driver. **chosenDriver** is set to the corresponding key for the last entry (with the highest fitness) in **driversAndFitnesses**. List index [-1] is used here to refer to the highest index. Setter methods are called accordingly to reflect the driver assignment.

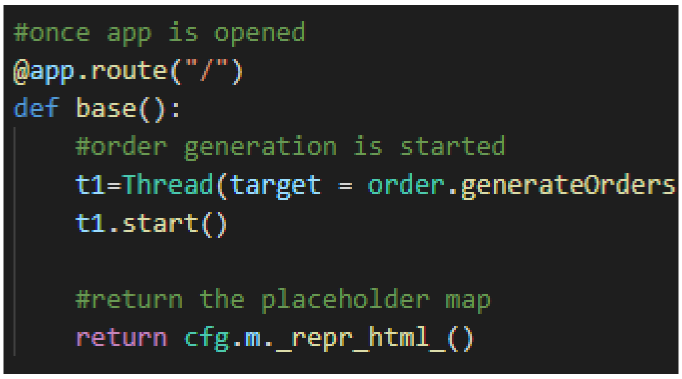
g. main

main is the module which ties all others together, including the Flask framework which will be used to house and serve the Folium map.



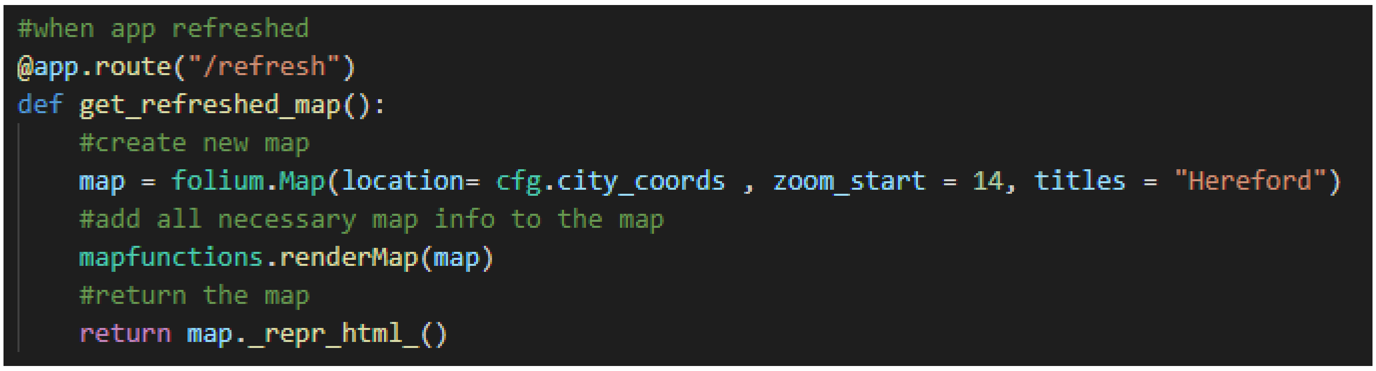
```python
#setup to be done initially
@app.before_first_request
def initialise():
    #setup the default map
    mapsetup.setupMap()
    #initialise drivers
    driver.initialiseDrivers()
```

Using the before_first_request statement allows for code to be ran prior to the app launching fully. Here, it is used to construct a map as well as initialise driver objects, so that the business is primed and ready to begin accepting orders.



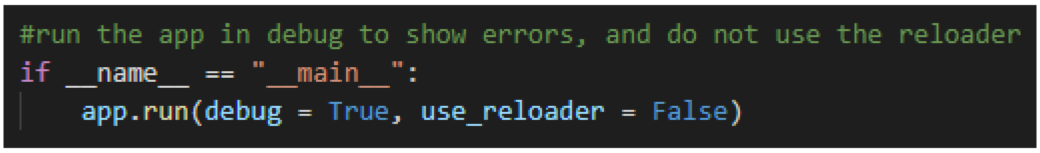
```python
#once app is opened
@app.route("/")
def base():
    #order generation is started
    t1=Thread(target = order.generateOrders
    t1.start()

    #return the placeholder map
    return cfg.m._repr_html_()
```

The above block will be ran once the Flask application is opened, but after the preceding code. A thread is used to handle order generation, this is because if a thread were not used, the map would never be returned as it would attempt to wait for all orders to be generated, which is an infinite process. The map is returned in html form, which Flask serves to the specified route.



```python
#when app refreshed
@app.route("/refresh")
def get_refreshed_map():
    #create new map
    map = folium.Map(location= cfg.city_coords , zoom_start = 14, titles = "Hereford")
    #add all necessary map info to the map
    mapfunctions.renderMap(map)
    #return the map
    return map._repr_html_()
```

When the system is refreshed by a manager, this code will be executed. As previously explained, the constraints of a lightweight framework such as Folium make it necessary to render a new map each time. Here, **mapfunctions** renders a new map given up-to-date information about orders and drivers.



```python
#run the app in debug to show errors, and do not use the reloader
if __name__ == "__main__":
    app.run(debug = True, use_reloader = False)
```

Flask requires this code; it effectively makes sure that the system is still running. Debug mode is used to allow for appropriate error messages to be returned, and the reloader set to False. During development, a problem was encountered where the system was being ran twice, however disabling the reloader fixed this problem.

IV. EVALUATION

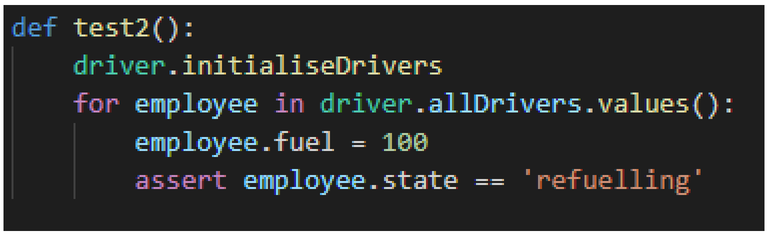
Now that the system had been developed, it was possible to conduct testing to ascertain whether or not the system satisfied the requirements as identified during the preparation stage. Analysis will also be performed of the system's suitability and usability for business applications as planned.

1. Testing

a. Black Box Testing

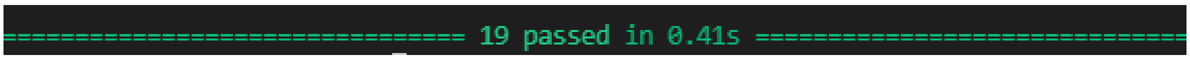
To conduct the state transition testing for driver objects, the **pytest** framework was used. This framework allows for the construction of tests in Python. Pytest has the ability to run multiple tests in parallel, which reduces the execution time of testing. It can also detect extraneous tests and skip them during execution. Because Pytest has very simple syntax, it is ideal to be applied to smaller projects, however it scales well to work with larger programs.

For each of the tests in the test table **(Appendix F)**, a test was written using the **pytest** framework. As an example, test case 2 stipulates that *"When the driver is 'idle' and their fuel drops below 15%, they should be set to 'refuelling' state".* When written as a test, this test case appears as follows:



```python
def test2():
    driver.initialiseDrivers
    for employee in driver.allDrivers.values():
        employee.fuel = 100
        assert employee.state == 'refuelling'
```

After a test had been written for every test case, the **test_module** was ran via simply typing **pytest** into the terminal. **Pytest** will automatically run all files of the form test_*.py or *_test.py in the current directory and its subdirectories.



```
=============================== 19 passed in 0.41s ==============================
```

Every test was passed; therefore, the system had successfully withstood state transition testing. From this result, it is clear that the behaviour of the **Driver** objects aligns with the desired behaviou0r specified during the preparation stage. The complete set of tests are included in **(Appendix J)**

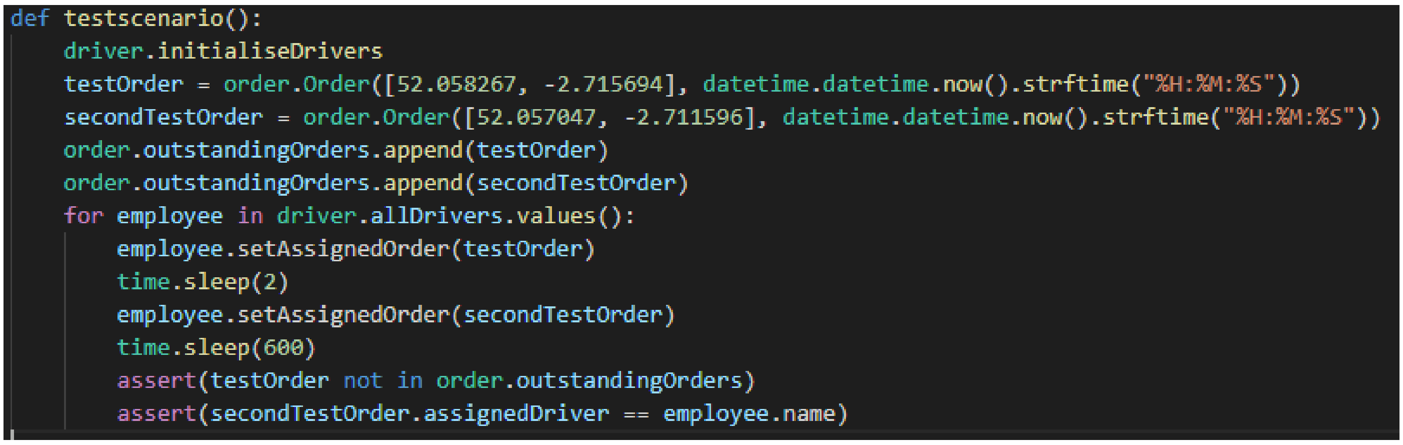
b. White Box Testing

Since program had been constructed, it became possible to design a test plan for white box testing and subject the system to branch coverage analysis. The stated objective of the white box testing was to ensure that the drivers' AI branching does not cause any unintended behaviour, and also that each branch is reachable. Before making a test plan, a program flow diagram was constructed to illustrate the different paths and branches which may be taken by each driver thread over the **stateCheck** method. This diagram can be found in **(Appendix H).** Test cases which covered every possible branch in this diagram were derived, and a test table containing these cases can be found in **(Appendix I).**

Similarly to the state transition testing, pytest tests were constructed for each test case identified within the test table. These tests can be found in **(Appendix K).** When running pytest, each of these tests passed. Therefore, it is safe to assume that each of these branches taken by the program function as intended, and additionally do not cause the program to crash.

c. Scenario Testing

To determine whether the system responded appropriately to the scenario, a test was written. For this test, multiple assertions were necessary. This is due to the complex nature of the scenario, especially in relation to the way in which the program is written. Since the program is multithreaded, it can be easy for erroneous values to be handled appropriately in one thread, but have undesirable results in another.

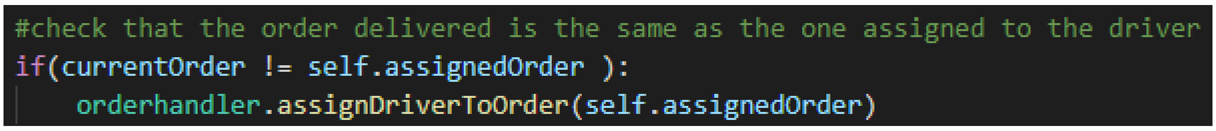


```
def testscenario():
    driver.initialiseDrivers
    testOrder = order.Order([52.058267, -2.715694], datetime.datetime.now().strftime("%H:%M:%S"))
    secondTestOrder = order.Order([52.057047, -2.711596], datetime.datetime.now().strftime("%H:%M:%S"))
    order.outstandingOrders.append(testOrder)
    order.outstandingOrders.append(secondTestOrder)
    for employee in driver.allDrivers.values():
        employee.setAssignedOrder(testOrder)
        time.sleep(2)
        employee.setAssignedOrder(secondTestOrder)
        time.sleep(600)
        assert(testOrder not in order.outstandingOrders)
        assert(secondTestOrder.assignedDriver == employee.name)
```

The test first initialises the drivers, then creates two dummy orders. Both are appended to **outstandingOrders**, as they would be during real execution. The driver is assigned the first order, and then a timer waits to allow for the employee's thread to set them to deliver it. Next, the employee under duress from an impatient customer assigns the second order to the driver. The sleep statement is there to give the driver's thread time to deliver the order. The first assertion determines whether the original order has been delivered. If it had been delivered, then it would no longer be in the **outstandingOrders** list. The second assertion determines whether the second order has now been assigned to the driver, as it should still have been in **outstandingOrders** and subsequently be assigned to the driver following the completion of their original delivery.

The first time this test was ran, it failed. This is because there was no built-in support for ensuring that erroneously assigned orders are not 'lost'. What had happened is although the drivers' assigned order was changed, they were still on a **driveTo** task to the original order. Once the drive had finished, the second test order was removed from **outstandingOrders** even though the driver was at the location of the original order. In this way, the system thought that the second order had been delivered instead of the first, causing a driver to be re-assigned to the first order and nobody to take the second order. In effect, the employee trying to speed up a customer's order had made it so that they would never receive it.

To remedy this problem, the following code was added to **acceptOrder**. This code ensures that once a driver reaches the location of an order, a check is made to see if the order they are currently assigned matches the order they are delivering. If it does not, then the assigned order is passed back to **orderhandler** to be either queued or assigned to another driver.



```
#check that the order delivered is the same as the one assigned to the driver
if(currentOrder != self.assignedOrder ):
    orderhandler.assignDriverToOrder(self.assignedOrder)
```

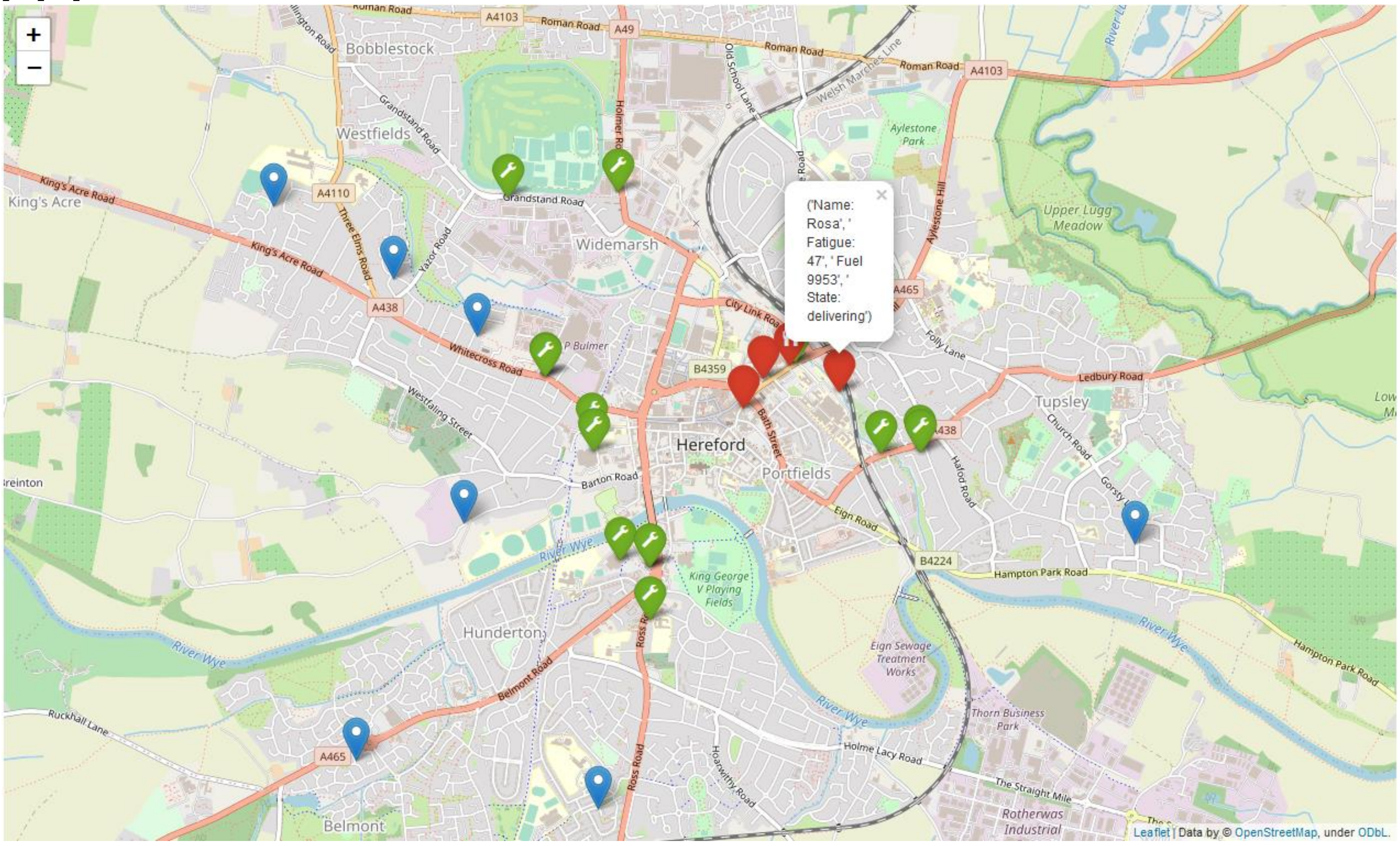
d. Usability Testing

The product was remotely tested with a group of friends who are managers of various businesses, all of which involve some form of home delivery. Copies of the tables returned by the testers can be found in **(Appendix M)**. Overall, the users found the look of the map to be good. They liked how the markers were obvious and different types contrasted each other. In terms of discerning relevant information from the map, a lot of testers stated that the manner in which they had to click on markers to see the information was not optimal. Unfortunately, Folium does not offer an alternative way to have information displayed. They also recommended that all markers should have icons, which is easily fixable through passing an icon design as a parameter when constructing the marker.

2. Requirements Analysis

This stage of the evaluation entails examining whether the deliverable sufficiently meets the functional and non-functional requirements outlined during requirements elicitation. **(Appendix L)** contains an updated requirements table with a decision on whether the program meets each requirement and why.

Notably, the program met every requirement except for number 7: *"The program's interface should be attractive and easily readable"*. Due to the fact that markers must be clicked on to display relevant information, the interface cannot be considered easily readable. Additionally, the fact that the interface could not be animated from its overall attractiveness. The awkward nature of the marker popups can be seen below.



```
('Name: Rosa', ' Fatigue: 47', ' Fuel 9953', ' State: delivering')
```

3. Reflections and Lessons Learned

The choices made throughout the project have all had an effect on the state of the outcome, some positive and some negative. Exploring the strengths and weaknesses of software projects is essential to personal improvement and development, to make sure that the next project's development is informed accordingly.

a. What went well

In terms of how the project met its goals, I am satisfied that the product is a solid system that could be used to route orders for real businesses, given a few modifications. I also believe that drivers working for businesses using the system would report higher rates of job satisfaction, due to increased agency over their working lives and higher psychological capital.

Technically speaking, I am very happy with the product. I have not used APIs or multithreading in a program before, and have not programmed with Python since 2015. The API requests worked exactly as intended, and it is satisfying that the product is able to use real-world geographic values. Multithreading worked perfectly in this implementation; I was concerned that the added level of complexity to the product would cause major roadblocks in development, however these were all readily overcome.

b. What could have gone better

The largest source of problems during development was the tools chosen to create the project with. I was initially drawn to Folium due to its easy setup and straightforward syntax, however it was evidently a library more suited for less ambitious projects. Instead, I could have opted for Leaflet, a JavaScript library for interactive maps, upon which Folium is already built. To avoid using suboptimal tools, I will ensure that any future projects are conducted with a greater emphasis on identifying appropriate software packages. The usability of the product and its appeal to potential clients was harmed as a result of this, and it would take a lot of work to move the system over onto another mapping framework.

Regarding the business aspect of the project, it is true that the system is easily adaptable to different locations and business types, however I am not convinced that many businesses would adopt the system due to its increase in delivery times compared to traditional systems. To remedy this, I could have made a comparison between the system and an example traditional system, to examine exactly how much it impacted delivery times. This would make the product more marketable, as businesses could be convinced that the difference in times is worth the increased employee satisfaction.

V. CONCLUSION

In conclusion, the product is successful in that it provides a human-centered solution to fleet vehicle routing, alongside a GUI and AI system for demonstration purposes. Technically, the product is solid and makes good use of some relatively advanced programming techniques, although its scope was hindered by use of suboptimal tools.

VI. REFERENCES

Cem Kaner, J., 2013. *An introduction to scenario testing,* Melbourne: Florida Institute of Technology.

Clarke, G. a. W. J., 1964. Scheduling of vehicles from a central depot to a number of delivery points. *Operations research,* 12(4), pp. 568-581.

Dantzig, G. a. R. J., 1959. The truck dispatching problem. *Management science,* 6(1), pp. 80-91.

Donovan, S. B. D. a. S. J., 2016. *What does the gig economy mean for workers?,* s.l.: s.n.

Drory, A., 1985. Effects of rest and secondary task on simulated truck-driving task performance. *Human Factors,* 27(2), pp. 201-207.

European Parliament, 2017. *Research for TRAN Committee -Road Transport Hauliers in the EU: Social and Working Conditions.,* Brussels: Policy Department for Structural and Cohesion Policies.

Frankl, P. a. W., 1993. Provable improvements on branch testing. *IEEE Transactions on Software Engineering,* 19(10), pp. 962-975.

Gendreau, M. H. A. a. L. G., 1994. A tabu search heuristic for the vehicle routing problem.. *Management science,* 40(10), pp. 1276-1290.

Healy, J. P. A. a. V. A., 2020. Sceptics or supporters? Consumers’ views of work in the gig economy. *New Technology, Work and Employment,* 35(1), pp. 1-19.

Ho, W. H. G. J. P. a. L. H., 2008. A hybrid genetic algorithm for the multi-depot vehicle routing problem. *Engineering applications of artificial intelligence,* 21(4), pp. 548-557.

Johnes, G., 2019. *The gig economy in the UK: a regional perspective.,* s.l.: Journal of Global Responsibility.

Kristina, S., 2019. Minimize transportation cost with clark and wright algorithm saving heuristic method with considering traffic congestion factor. *IOP Conference Series: Materials Science and Engineering,* 673(1), p. 012080.

Luthans, F. Y. C. a. A. B., 2015. *Psychological capital and beyond,* USA: Oxford University Press.

Manyika, J. L. S. B. J. R. K. M. J. a. M. D., 2016. *Independent-Work-Choice-necessity-and-the-gig-economy,* s.l.: Mckinsey Global Institute.

Mittal, P. G. N. A. H. a. M. C., 2017. Solving VRP in an Indian Transportation Firm through Clark and Wright Algorithm: A Case Study. *International journal of emerging technologies in engineering research,* 5(10).

Nazir, S., 2021. *6 Retailers that have addressed the lorry driver shortage.* [Online]
Available at: https://www.retailgazette.co.uk/blog/2021/08/6-retailers-that-have-addressed-the-lorry-driver-shortage/
[Accessed 05 August 2021].

Pisinger, D. a. R. S., 2007. A general heuristic for vehicle routing problems. *Computers & operations research,* 34(8), pp. 2403-2435.

Prockl, G. T. C. K. H. a. A. R., 2017. Antecedents of truck drivers’ job satisfaction and retention proneness. *Journal of Business Logistics,* 38(3), pp. 184-196.

Schulz, S. L. K. a. M. J., 2014. Psychological capital: A new tool for driver retention. *International Journal of Physical Distribution & Logistics Management..*

Stave, A., 1977. The effects of cockpit environment on long-term pilot performance. *Human Factors,* 19(5), pp. 503-514.

Sweeney, E., 2013. The People Dimension in Logistics & Supply Chain Management – It’s Role and Importance.. In: R. P. a. A. Thomas, ed. *Supply Chain Management: Perspectives, Issues and Cases.* Milan: McGraw-Hill, pp. 73-82.

Taillard, E., 1993. Benchmarks for basic scheduling problems. *european journal of operational research,* 64(2), pp. 278-285.

Tucker, P., 2003. The impact of rest breaks upon accident risk, fatigue and performance: a review.. *Work & Stress,* 17(2), pp. 123-137.

Vidal, T. C. T. G. M. L. N. a. R. W., 2012. A hybrid genetic algorithm for multidepot and periodic vehicle routing problems. *Operations Research,* 60(3), pp. 611-624.

Vidal, T. L. G. a. M. P., 2020. A concise guide to existing and emerging vehicle routing problem variants. *European Journal of Operational Research,* 286(2), pp. 401-416.

VII. APPENDICES

Appendix A

Story ID	As a ...	I want the system to ...	Because ...
1	Driver	Allow me to choose personal stops and route me near them	This will allow me to complete personal tasks during my downtime, such as collecting an order from a store
2	Manager	Display the location of drivers on a local map	Seeing where the drivers are reassures me that the orders are being processed effectively
3	Manager	Display the location and time outstanding of orders	I can see if any orders have been unfulfilled and take action, I can also gain greater understanding of the nature of our customers' locations and order patterns.
4	Driver	Know when I need to refuel and route me near to a fuel station	This means that refuelling will take me on less of a detour and put less pressure on me to hurry up
5	Driver	Take into account my fatigue and allow me to take rest breaks when requested	I do not want to be overworked and being able to choose rest breaks would make me more content with my job
6	Manager	Be applicable to any location	Chain locations using the software require operability in any location
7	Manager	Have a pleasant and easy-to-read GUI	Fast-paced work environments mean that I need to be able to quickly and easily see any information I require
8	Manager	Work with an external file holding driver information	This will allow the system to integrate readily with my existing employee records, no need to enter all the details again
9	Manager	Take an address of an order and display it on the map	I will be able to see how many orders are outstanding and visualise how far away drivers are on the map

Appendix B

Requirement ID	Relevant User Story ID	Requirement	Functional / Non-Functional?	Justification
1	1	The system should accept personal stop requests from drivers and attempt to route them around these stops where possible	Functional	This allows drivers to make any personal stops they need during their downtime, saving them from having to take time out after finishing work

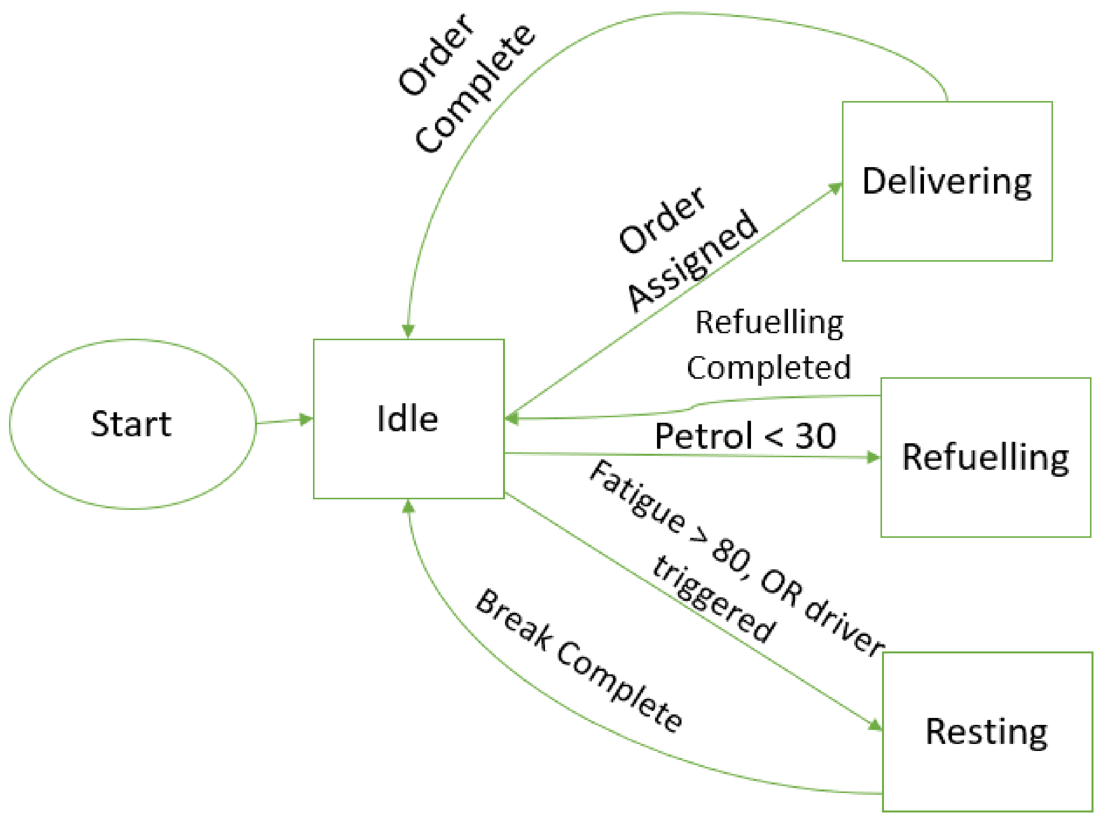
2	2	The system should display the location of drivers on top of a map	Functional	This allows the manager to keep track of the drivers and ensure that the orders are being dealt with
3	3	The system should display the location of orders and time they have been waiting	Functional	Any orders which have been waiting too long can be addressed by the manager
4	4	The system should keep an eye on driver fuel levels and route them near to a petrol station when they are running low	Functional	This saves drivers' time being taken to drive themselves to a petrol station, giving them more time to themselves
5	5	The system should monitor driver fatigue prioritise orders to drivers that aren't as fatigued	Functional	Drivers who are exhausted are given a break. This avoids tired driving, which is both unpleasant for drivers and dangerous.
6	6	The system should be able to work in any given location	Functional	The system should be flexible with its localisation to allow compatibility with organizations and chains spanning across the UK
7	7	The system's interface should be attractive and easily readable	Non-Functional	Managers should be able to easily see information through a GUI.
8	8	The system should be able to accept driver information from an external file	Functional	Having an external driver file means that companies do not have to enter driver information that is already on their system
9	9	The system should be able to accept addresses of orders and display them on the map	Functional	This allows the manager to be able to visualise current orders and their locations

Appendix C

State Name	**Description**	**Triggers**	**Justification**

Idle	Driver has no current task	Any task ends and the delivery queue is empty, driver is not fatigued and has enough fuel At program initialisation	Once a task is completed, the driver is able to have some 'idle' time. Drivers will begin in an idle state as there are no tasks straight away
Delivering	Driver has accepted a delivery and is en route	Order handler assigns driver a delivery	Drivers will not be able to complete other tasks whilst on a delivery
Refuelling	Driver is in the process of refuelling their vehicle	Driver's fuel drops below 30% whilst they are 'idle'	From time to time, drivers will need to refuel their vehicle and cannot complete tasks as their fuel is too low. Refuelling should be done only once current task has been completed
Resting	Driver is taking a break	Driver triggered, or when fatigue reaches 80%	Drivers must be allowed to rest undisturbed. It is important that drivers do not work when too tired, for their own happiness and also for road safety.

Appendix D



Appendix E

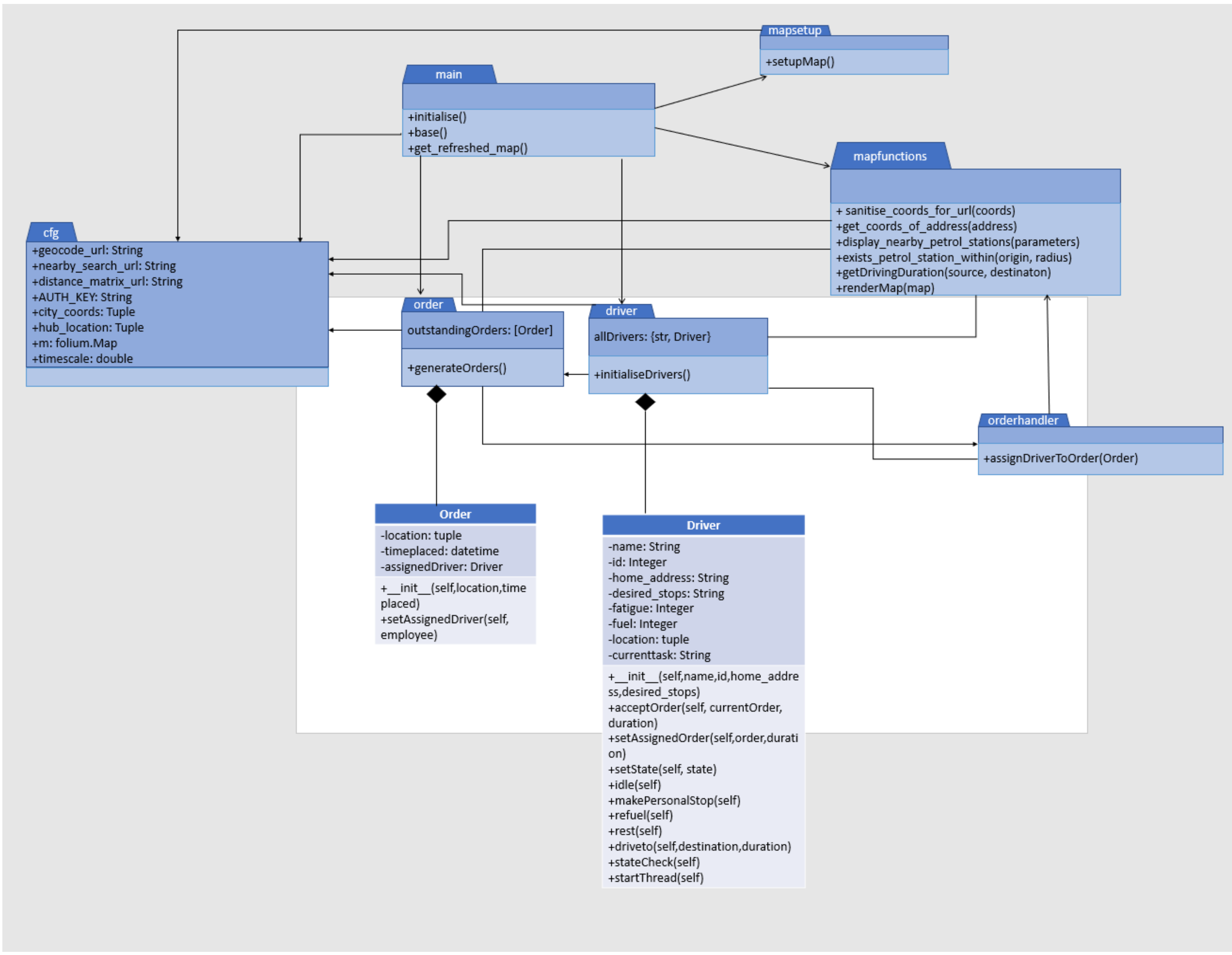
State	Order Assigned	Petrol < 30	Fatigue > 80, or driver requests break	Task Completed
S1) Idle	S2	S3	S4	S1
S2) Delivering	S2	S2	S2	S1
S3) Refuelling	S3	S3	S4	S1
S4) Resting	S4	S4	S4	S1

Appendix F

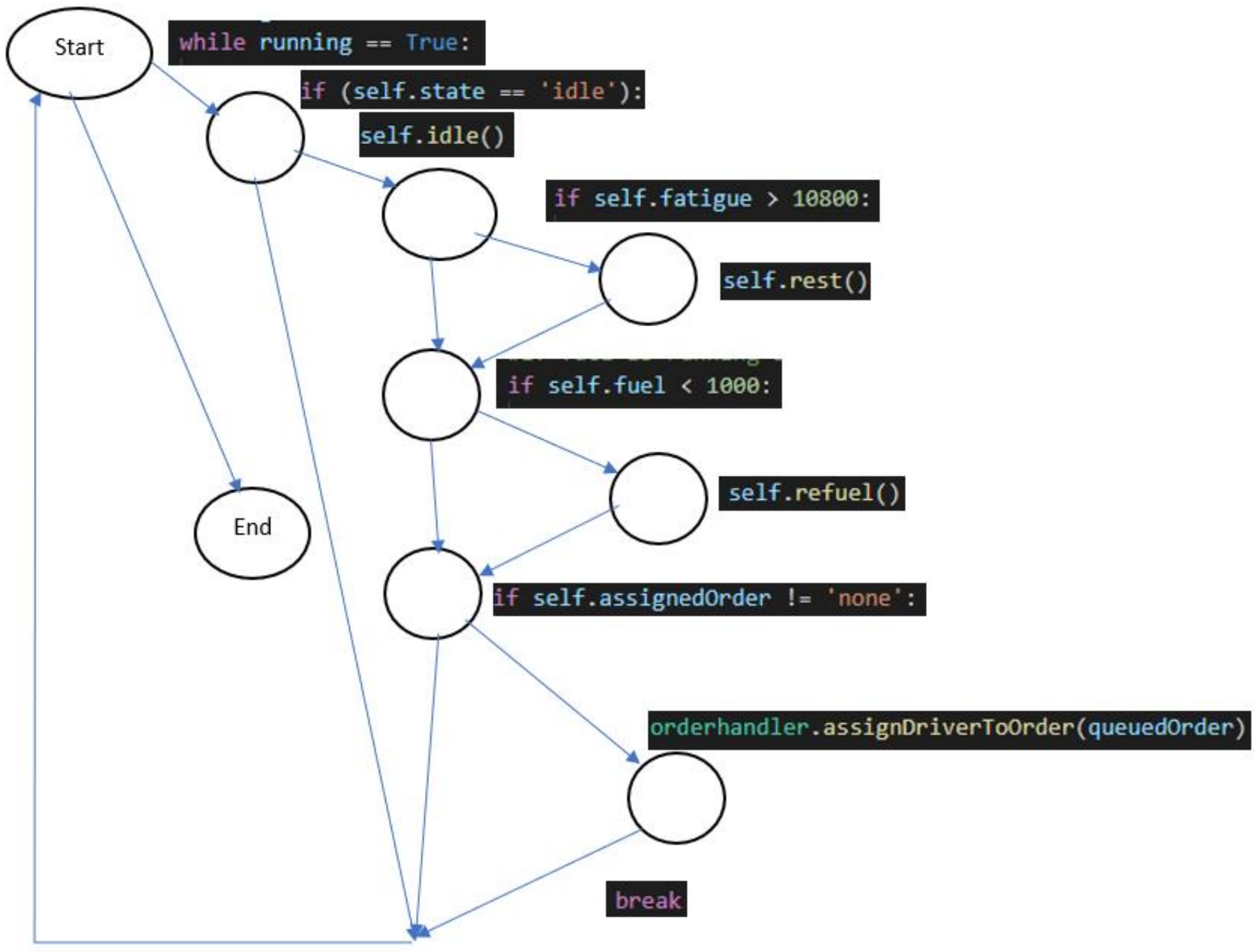
Test Case #	Test Case Description	Test Data	Expected Result	Actual Result	Pass/Fail
1	When the program is started, the driver object should be in 'idle' state	Start the program	Driver state = 'Idle'		
2	When the driver is 'idle' and their fuel drops below 15%, they should be set to 'refuelling' state	Set driver state to idle and fuel to below 30	Driver state = 'refuelling'		
3	When the driver is 'idle' and they are assigned a delivery, they should be set to 'delivering' state	Set driver state to idle and assign a delivery to them	Driver state = 'delivering'		
4	When the driver is 'idle' and their fatigue is above 10800, they should be set to 'resting' state	Set driver state to idle and fatigue to above 10800	Driver state= 'on break'		
5	When the driver is 'idle' and their request a break, they should be set to 'resting' state	Set driver state to idle and request a break from driver	Driver state = 'on break'		
6	When the driver is 'delivering' and they are assigned an order, they should remain in 'delivering' state	Set driver state to delivering and assign a delivery to them	Driver state = 'delivering'		
7	When the driver is 'delivering' and their fuel drops below 15%, they should remain in 'delivering' state	Set driver state to delivering and set their fuel to below 15%	Driver state = 'delivering'		
8	When the driver is 'delivering' and their fatigue goes above 80, they should remain in 'delivering' state	Set driver state to delivering and fatigue to over 80	Driver state = 'delivering'		
9	When the driver is 'delivering' and they request a break, they should remain in 'delivering' state	Set driver state to delivering and request a break from driver	Driver state = 'delivering'		

10	When the driver is ‘delivering’ and they complete their delivery, they should return to ‘idle’ state	Set driver state to delivering and their current order to complete	Driver state = ‘idle’		
11	When the driver is ‘refuelling’ and they are assigned an order, they remain ‘refuelling’	Set driver state to refuelling and assign them an order	Driver state = ‘refuelling’		
12	When the driver is ‘refuelling’ and their fuel goes below 15%, they should remain refuelling	Set driver state to refuelling and fuel level below 15%	Driver state = ‘refuelling’		
13	When the driver is ‘refuelling’ and their fatigue reaches over 10800, they should be set to ‘resting’	Set driver state to refuelling and fatigue to over 10800	Driver state = ‘resting’		
14	When the driver is ‘refuelling’ and they request a break, they should be set to ‘resting’	Set driver state to refuelling and request break from driver	Driver state = ‘resting’		
15	When the driver is ‘refuelling’ and they complete their task, they should return to ‘idle’	Set driver state to refuelling and indicate that their task is complete	Driver state = ‘idle’		
16	When the driver is ‘resting’ and they are assigned an order, they should remain resting	Set driver state to resting and assign them an order	Driver state = ‘resting’		
17	When the driver is ‘resting’ and their petrol drops below 30, they should remain resting	Set driver state to resting and set fuel to below 30	Driver state = ‘resting’		
18	When the driver is ‘resting’ and their fatigue is above 10800, they should remain resting	Set driver state to resting and fatigue to above 80	Driver state = ‘resting’		
19	When the driver is ‘resting’ and they request a break, they should remain resting	Set driver state to resting and request break from driver	Driver state = ‘resting’		

Appendix G


mapsetup
+setupMap()
main
+initialise()
+base()
+get_refreshed_map()
mapfunctions
+ sanitise_coords_for_url(coords)
+get_coords_of_address(address)
+display_nearby_petrol_stations(parameters)
+exists_petrol_station_within(origin, radius)
+getDrivingDuration(source, destinaton)
+renderMap(map)
cfg
+geocode_url: String
+nearby_search_url: String
+distance_matrix_url: String
+AUTH_KEY: String
+city_coords: Tuple
+hub_location: Tuple
+m: folium.Map
+timescale: double
order
outstandingOrders: [Order]
+generateOrders()
driver
allDrivers: {str, Driver}
+initialiseDrivers()
orderhandler
+assignDriverToOrder(Order)
Order
-location: tuple
-timeplaced: datetime
-assignedDriver: Driver
+__init__(self,location,time placed)
+setAssignedDriver(self, employee)
Driver
-name: String
-id: Integer
-home_address: String
-desired_stops: String
-fatigue: Integer
-fuel: Integer
-location: tuple
-currenttask: String
+__init__(self,name,id,home_address,desired_stops)
+acceptOrder(self, currentOrder, duration)
+setAssignedOrder(self,order,duration)
+setState(self, state)
+idle(self)
+makePersonalStop(self)
+refuel(self)
+rest(self)
+driveto(self,destination,duration)
+stateCheck(self)
+startThread(self)

Appendix H

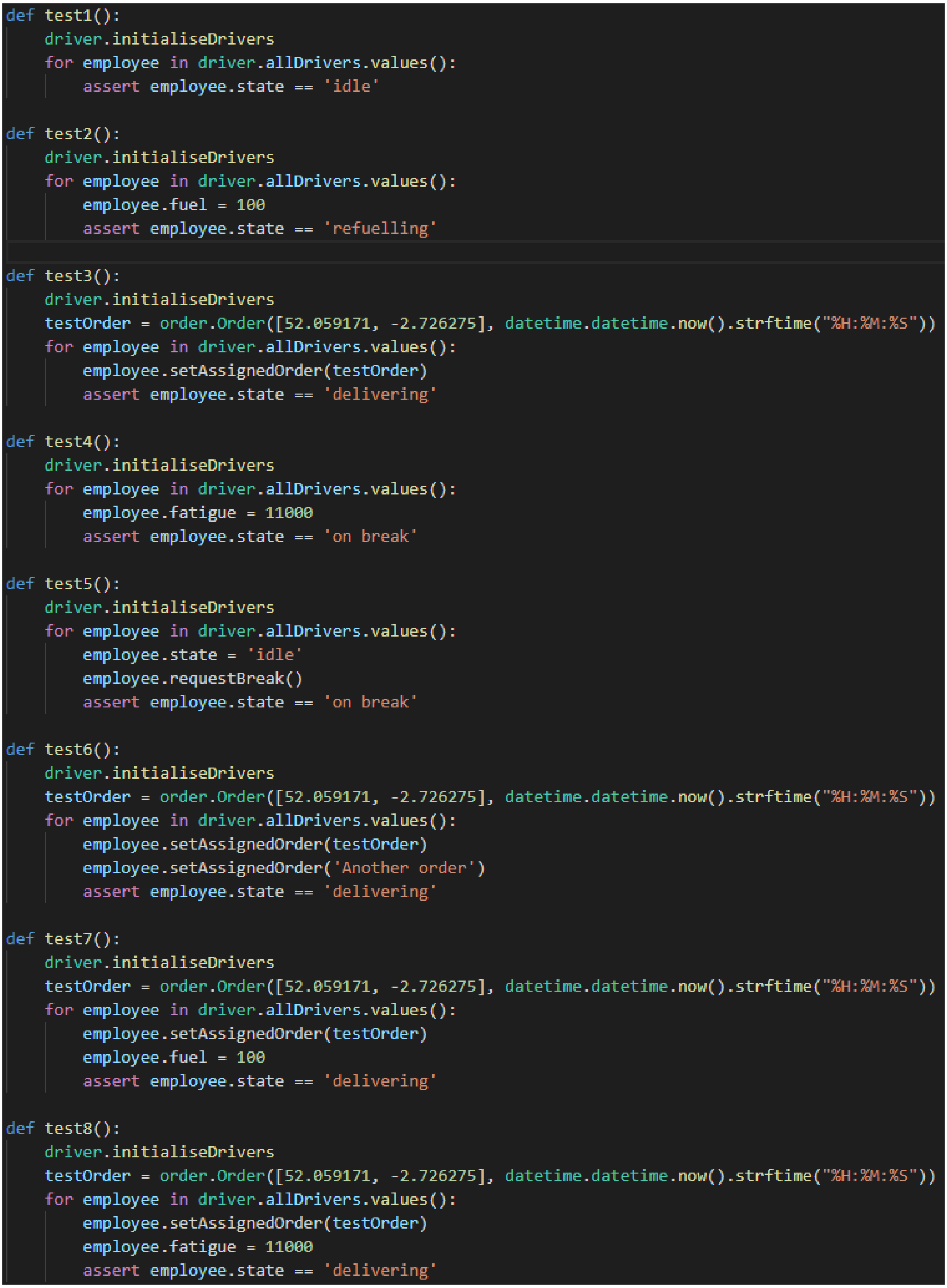


```
while running == True:
if (self.state == 'idle'):
self.idle()
if self.fatigue > 10800:
self.rest()
if self.fuel < 1000:
self.refuel()
if self.assignedOrder != 'none':
orderhandler.assignDriverToOrder(queuedOrder)
break
```

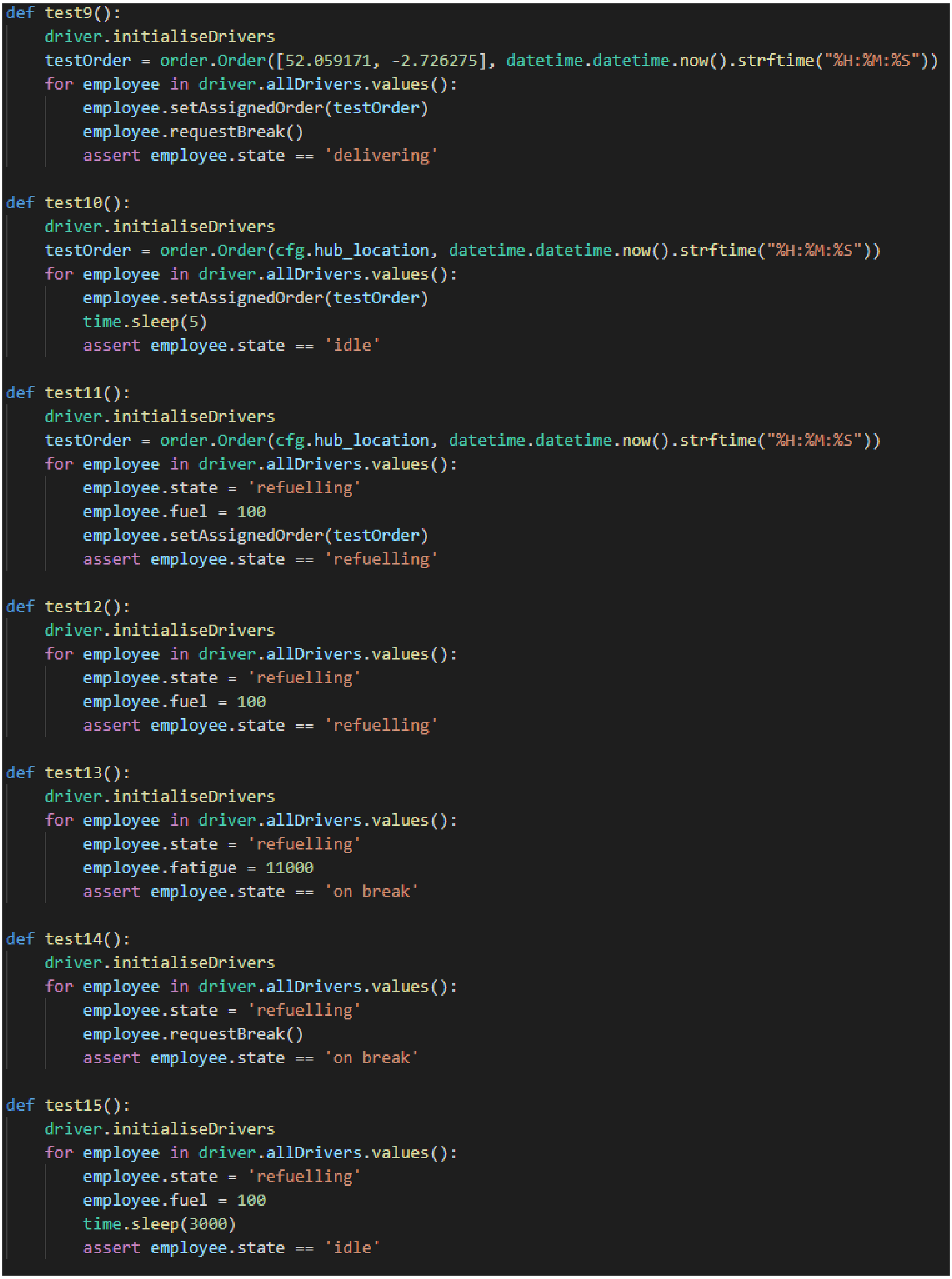
Appendix I

Running is true	**Driver state is idle**	**Fatigue is greater than 10800**	**Fuel is lower than 1000**	**assignedOrder is not equal to 'none'**	**Outcome**
False	-	-	-	-	Process Ends
True	False	-	-	-	Process restarts
True	True	True	-	-	Driver Rests
True	True	False	True	-	Driver refuels
True	True	False	False	True	Driver delivers
True	True	False	False	False	Process restarts

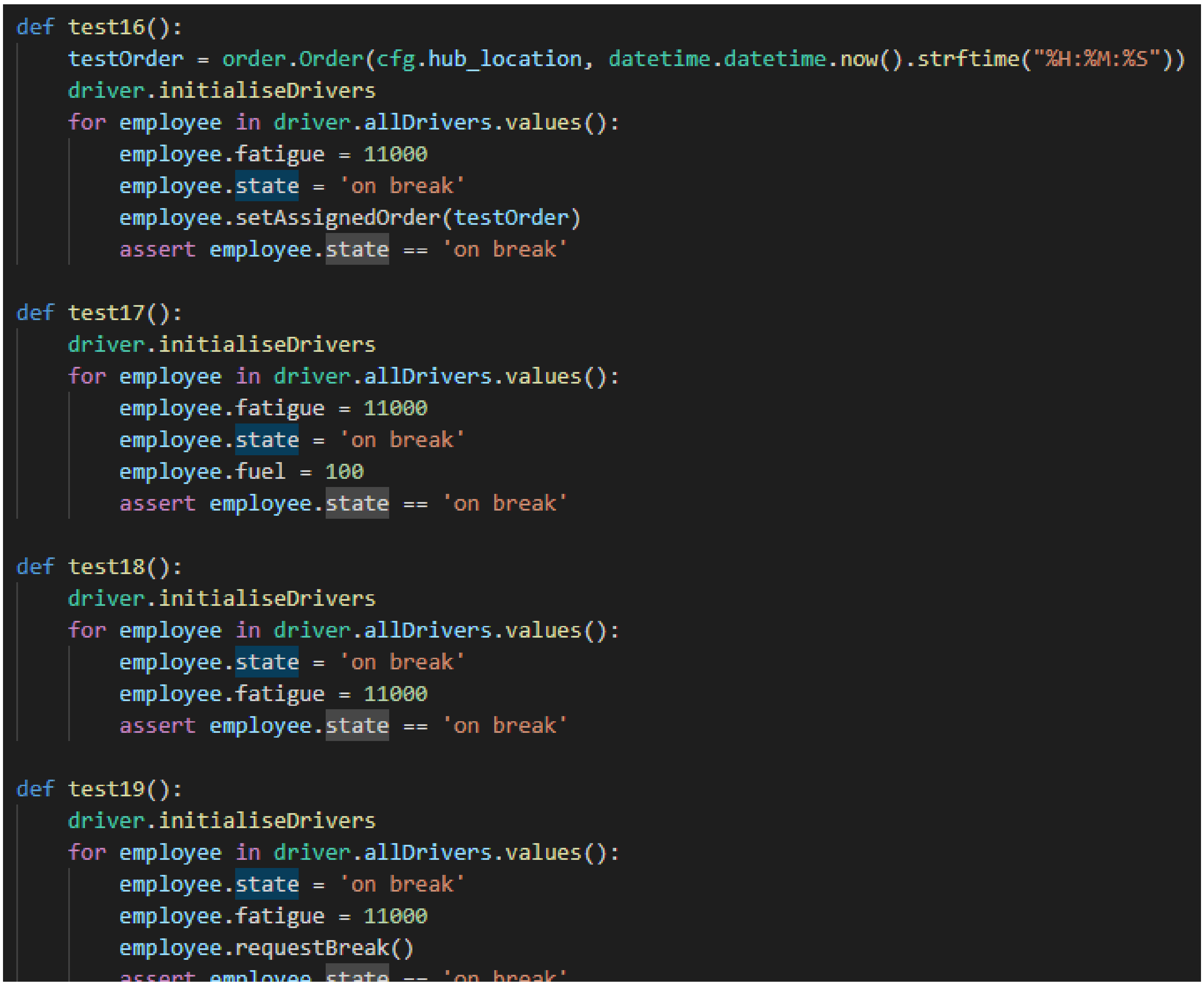
Appendix J



```python
def test1():
    driver.initialiseDrivers
    for employee in driver.allDrivers.values():
        assert employee.state == 'idle'

def test2():
    driver.initialiseDrivers
    for employee in driver.allDrivers.values():
        employee.fuel = 100
        assert employee.state == 'refuelling'

def test3():
    driver.initialiseDrivers
    testOrder = order.Order([52.059171, -2.726275], datetime.datetime.now().strftime("%H:%M:%S"))
    for employee in driver.allDrivers.values():
        employee.setAssignedOrder(testOrder)
        assert employee.state == 'delivering'

def test4():
    driver.initialiseDrivers
    for employee in driver.allDrivers.values():
        employee.fatigue = 11000
        assert employee.state == 'on break'

def test5():
    driver.initialiseDrivers
    for employee in driver.allDrivers.values():
        employee.state = 'idle'
        employee.requestBreak()
        assert employee.state == 'on break'

def test6():
    driver.initialiseDrivers
    testOrder = order.Order([52.059171, -2.726275], datetime.datetime.now().strftime("%H:%M:%S"))
    for employee in driver.allDrivers.values():
        employee.setAssignedOrder(testOrder)
        employee.setAssignedOrder('Another order')
        assert employee.state == 'delivering'

def test7():
    driver.initialiseDrivers
    testOrder = order.Order([52.059171, -2.726275], datetime.datetime.now().strftime("%H:%M:%S"))
    for employee in driver.allDrivers.values():
        employee.setAssignedOrder(testOrder)
        employee.fuel = 100
        assert employee.state == 'delivering'

def test8():
    driver.initialiseDrivers
    testOrder = order.Order([52.059171, -2.726275], datetime.datetime.now().strftime("%H:%M:%S"))
    for employee in driver.allDrivers.values():
        employee.setAssignedOrder(testOrder)
        employee.fatigue = 11000
        assert employee.state == 'delivering'
```

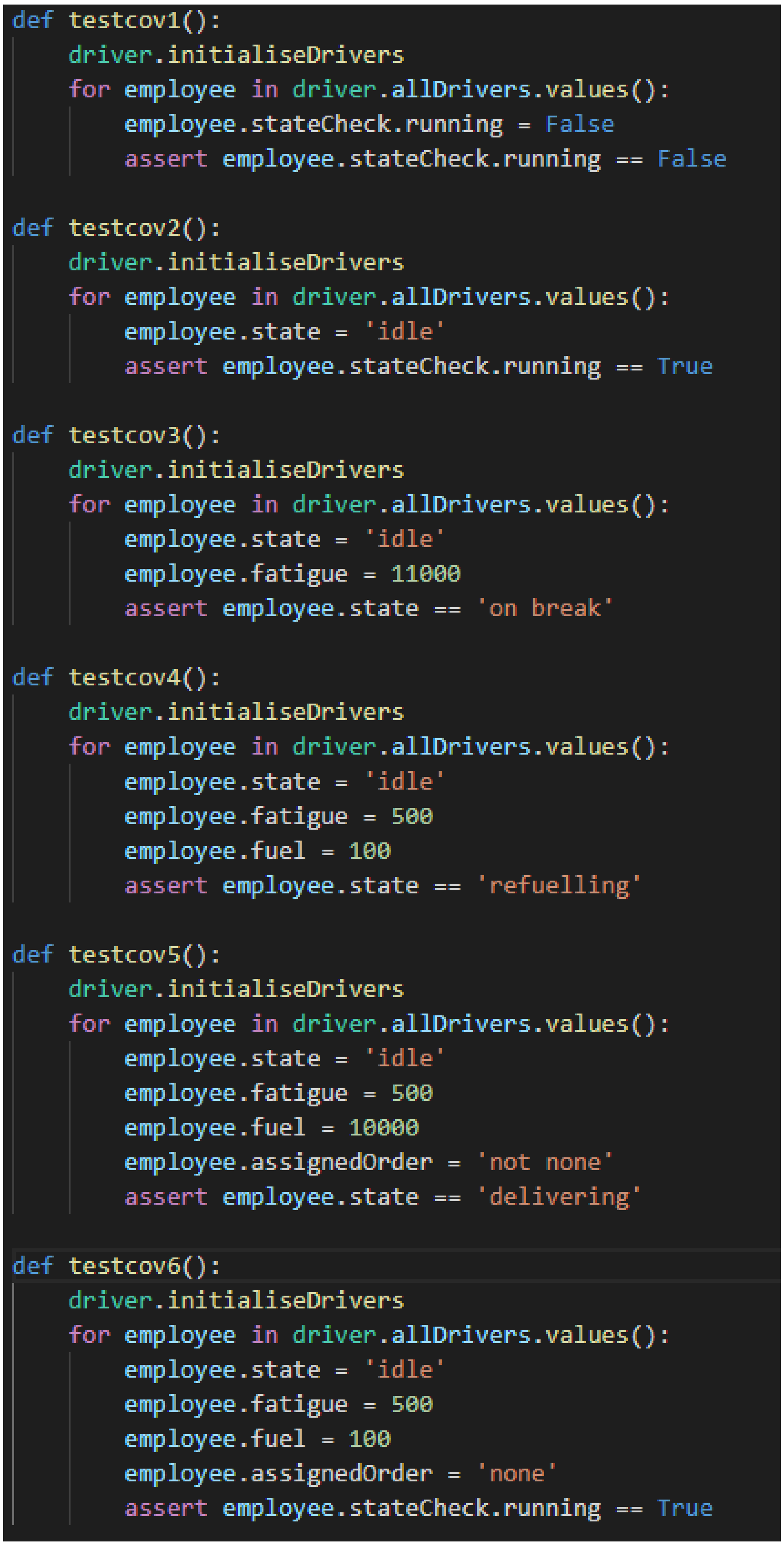


```python
def test9():
    driver.initialiseDrivers
    testOrder = order.Order([52.059171, -2.726275], datetime.datetime.now().strftime("%H:%M:%S"))
    for employee in driver.allDrivers.values():
        employee.setAssignedOrder(testOrder)
        employee.requestBreak()
        assert employee.state == 'delivering'

def test10():
    driver.initialiseDrivers
    testOrder = order.Order(cfg.hub_location, datetime.datetime.now().strftime("%H:%M:%S"))
    for employee in driver.allDrivers.values():
        employee.setAssignedOrder(testOrder)
        time.sleep(5)
        assert employee.state == 'idle'

def test11():
    driver.initialiseDrivers
    testOrder = order.Order(cfg.hub_location, datetime.datetime.now().strftime("%H:%M:%S"))
    for employee in driver.allDrivers.values():
        employee.state = 'refuelling'
        employee.fuel = 100
        employee.setAssignedOrder(testOrder)
        assert employee.state == 'refuelling'

def test12():
    driver.initialiseDrivers
    for employee in driver.allDrivers.values():
        employee.state = 'refuelling'
        employee.fuel = 100
        assert employee.state == 'refuelling'

def test13():
    driver.initialiseDrivers
    for employee in driver.allDrivers.values():
        employee.state = 'refuelling'
        employee.fatigue = 11000
        assert employee.state == 'on break'

def test14():
    driver.initialiseDrivers
    for employee in driver.allDrivers.values():
        employee.state = 'refuelling'
        employee.requestBreak()
        assert employee.state == 'on break'

def test15():
    driver.initialiseDrivers
    for employee in driver.allDrivers.values():
        employee.state = 'refuelling'
        employee.fuel = 100
        time.sleep(3000)
        assert employee.state == 'idle'
```



```python
def test16():
    testOrder = order.Order(cfg.hub_location, datetime.datetime.now().strftime("%H:%M:%S"))
    driver.initialiseDrivers
    for employee in driver.allDrivers.values():
        employee.fatigue = 11000
        employee.state = 'on break'
        employee.setAssignedOrder(testOrder)
        assert employee.state == 'on break'

def test17():
    driver.initialiseDrivers
    for employee in driver.allDrivers.values():
        employee.fatigue = 11000
        employee.state = 'on break'
        employee.fuel = 100
        assert employee.state == 'on break'

def test18():
    driver.initialiseDrivers
    for employee in driver.allDrivers.values():
        employee.state = 'on break'
        employee.fatigue = 11000
        assert employee.state == 'on break'

def test19():
    driver.initialiseDrivers
    for employee in driver.allDrivers.values():
        employee.state = 'on break'
        employee.fatigue = 11000
        employee.requestBreak()
        assert employee.state == 'on break'
```

Appendix K



```
def testcov1():
    driver.initialiseDrivers
    for employee in driver.allDrivers.values():
        employee.stateCheck.running = False
        assert employee.stateCheck.running == False

def testcov2():
    driver.initialiseDrivers
    for employee in driver.allDrivers.values():
        employee.state = 'idle'
        assert employee.stateCheck.running == True

def testcov3():
    driver.initialiseDrivers
    for employee in driver.allDrivers.values():
        employee.state = 'idle'
        employee.fatigue = 11000
        assert employee.state == 'on break'

def testcov4():
    driver.initialiseDrivers
    for employee in driver.allDrivers.values():
        employee.state = 'idle'
        employee.fatigue = 500
        employee.fuel = 100
        assert employee.state == 'refuelling'

def testcov5():
    driver.initialiseDrivers
    for employee in driver.allDrivers.values():
        employee.state = 'idle'
        employee.fatigue = 500
        employee.fuel = 10000
        employee.assignedOrder = 'not none'
        assert employee.state == 'delivering'

def testcov6():
    driver.initialiseDrivers
    for employee in driver.allDrivers.values():
        employee.state = 'idle'
        employee.fatigue = 500
        employee.fuel = 100
        employee.assignedOrder = 'none'
        assert employee.stateCheck.running == True
```

Appendix L

Requirement no	Requirement	Requirement Met?	Justification
1	The system should accept personal stop	Yes	Personal stops are part of the considerations made by the

	requests from drivers and attempt to route them around these stops where possible		program when assigning drivers to orders
2	The system should display the location of drivers on top of a map	Yes	Folium is used to provide this functionality.
3	The system should display the location of orders and time they have been waiting	Yes	Folium marker popups satisfy this requirement
4	The system should keep an eye on driver fuel levels and route them near to a petrol station when they are running low	Yes	Driver fuel levels are part of the considerations made by the algorithm when assigning drivers to orders
5	The system should monitor driver fatigue and prioritise orders to drivers that aren't as fatigued	Yes	The algorithm considers fatigue when assigning orders
6	The system should be able to work in any given location	Yes	Business coordinates are configurable in cfg. Geocoding can take any address and return coordinates, not limited to location.
7	The system's interface should be attractive and easily readable	No	Folium has its limitations; the manager must click on icons to see details. The interface is not animated, requiring refreshing to see an updated view.
8	The system should be able to accept driver information from an external file	Yes	Information from driverinfo.txt is used to create the driver objects
9	The system should be able to accept addresses of orders and display them on the map	Yes	Geocoding allows addresses to be translated into coordinates and then placed on the map with a Folium marker.

Appendix M

Task	User Feedback
Identify the location of the business hub, the drivers, the customers, and the petrol stations on the map	The locations are marked nicely, I like the contrast between the different colours
Determine which orders have been seen to and which are still pending driver assignment	This was a little difficult – I think that orders which have no driver should be a different colour to distinguish
For each driver, identify what their current task is.	Easy enough – the information is clear.
Task	**User Feedback**
Identify the location of the business hub, the drivers, the customers, and the petrol stations on the map	This task was easy, I don't see why some markers are missing icons though
Determine which orders have been seen to and which are still pending driver assignment	I don't like that you have to click on the markers. Would be better to have the information hovering over them
For each driver, identify what their current task is.	Again, it would be better to have the information always available
Task	**User Feedback**
Identify the location of the business hub, the drivers, the customers, and the petrol stations on the map	All are laid out clearly. Map looks very nice
Determine which orders have been seen to and which are still pending driver assignment	A bit tucked away, but intuitive enough
For each driver, identify what their current task is.	Its good that you can see the tasks, as a manager I would like to know what the drivers are currently doing for my peace of mind
Task	**User Feedback**
Identify the location of the business hub, the drivers, the customers, and the petrol stations on the map	It should be that every marker has an icon, some do and some don't
Determine which orders have been seen to and which are still pending driver assignment	This was easy enough – it does seem that orders are generated quite slowly in this simulation though!
For each driver, identify what their current task is.	I didn't understand this at first, then I realised that the task was listed as 'state'