On Board Vehicle Companion Reflection

For my senior project I created a vehicle communication and trip logging tool using the Python-OBD library along with the PyQt5 library for the user interface. The goal of this project was to create an application that provides a similar functionality as the consumer grade vehicle diagnostic tools available for purchase while also gaining a better understanding of vehicle programming and communication protocols. My application offers users the option to monitor a dashboard of vehicle variables, log and graph a trip, and query information about the check engine light as well as any present trouble codes.

I have always been interested in the computer side of vehicles and it was a goal of mine to incorporate this into my senior project. After brainstorming a variety of different ideas, I found that it was unrealistic to attempt to do anything with the source code responsible for making my car run. Instead I chose a safer alternative, communicating with the on board diagnostic system. My goal was to make this application a vehicle companion. I decided to use a raspberry pi with a small display, which allows me to keep everything in my vehicle at all times. However, much of this project was developed on my laptop, therefore the application functions on both windows and Linux platforms. I also must credit Tinkernut on YouTube whose DIY smartcar project helped cement this idea in my head and highlight the abilities of Python-OBD.

**Design**

I have never been artistically inclined nor do I have an eye for design. I decided it was more important to focus on usability and available features when designing my application. The process of deciding on a list of features that would be useful for the user while also being attainable with the technologies on hand was a bit overwhelming. After a lot of researching online and testing the different features of BlueDriver OBD, a consumer grade diagnostic tool, I eventually decided on four tabs that the user is able to switch from at their leisure. Providing a tabbed architecture allowed me to separate functionality that requires different types of connections and keep each tab clutter free.

The first tab is a dashboard, which provides the user with live displays of the vehicle speed, rpms, engine load, and intake air temperature. The second tab allows the user to check if there are any present trouble codes, read any codes that exist, get freeze frame data, and clear the trouble codes. The third tab is a trip logging feature with buttons for starting and stopping a trip and graphs for displaying the information. The fourth tab contains a link to the trip report, a text file that is appended after every logged trip. Every logged trip contains the date, time, trip duration in seconds, estimated distance, information about the check engine light and any codes if they exist, and a list of the vehicle speed, rpms, and engine load throughout the trip.

**Organization**

The skeleton of my code is a large class that builds and initializes the GUI, handles connection changes, and interacts with the OBD2 connector. I then create an instance of that class in the main method, which loads the GUI and keeps it running until the user quits the application or the code execution is stopped.

**Getting Started**

Since I already had an old iPad on hand, I decided to use it as the raspberry pi display. I was initially successful in using the iPad as a display by connecting both the pi and the iPad to the same Wi-Fi network and then establishing a VNC connection. However this was very limiting, as I would need a Wi-Fi connection at all times to use my application.

After doing some research online on alternative ways to use an iPad as a display, I stumbled upon an article detailing how to turn your raspberry pi into its own wireless access point(WAP). Putting two and two together I realized that if I was successful in turning the pi into a WAP I could establish a VNC connection to the iPad, even if I don’t have internet access. After many attempts and different tutorials, I was unsuccessful in emitting a Wi-Fi signal from the Raspberry Pi. I eventually found RaspAP, a software, which automatically creates a WAP from your pi. This software, by automatically disabling the pi’s ability to connect to Wi-Fi when installed, does not offer the same control as doing it yourself (and thus knowing how to quickly revert any changes). However, it still provided me with the necessary utility, establishing a VNC connection even when there is no underlying internet connection.

After figuring out how to create an offline display for the pi, I moved on to testing the Bluetooth communication between the pi and the $10 ELM327 OBD2 adapter I purchased on Amazon. Since the OBD adapter communicates through serial, it is first necessary to bind it to a serial port on the Raspberry Pi. To establish the initial connection I followed the tutorial Tinkernut had online. After this, I was able to communicate with the vehicle using Parameter IDs (PIDs). PIDs are hexadecimal codes used to request diagnostic data from your car, a complete list can be found on https://en.wikipedia.org/wiki/OBD-II\_PIDs. Once initial connectivity and communication were established I felt confident that this project would be attainable and enjoyable.

My first big hiccup came when I was attempting to establish a connection between the OBD2 adapter and the Python-OBD library. After an hour of messing around with different COM ports, logging debug info, and trying both manual and automatic connection protocols, nothing seemed to work. I finally found a forum post which suggested setting all “AT” commands in the OBD\elm327.py file of the Python-OBD library to have a delay of 1 (adding delay=1 to the arguments). AT commands allow execution control of the currently enabled python script. After manually changing all of these commands, I was able to establish a successful synchronous connection and get responses for both the vehicle speed and RPMs. With all of this set up, I was ready to construct my project.

**Python-OBD**

The bulk of the functional code in this project comes in the form of calls to the Python-OBD library. Python-OBD offers two different communication modes with a vehicles on board diagnostic system (OBD), synchronous and asynchronous. Per the documentation, “Synchronous mode allows you to query vehicle information in a request reply type fashion… Since the standard query() function is blocking, it can be a hazard for UI event loops. To deal with this, python-OBD has an Async connection object that can be used in place of the standard OBD object." Learning how the library works and how to use it was slightly laborious but as simple as reading through the online documentation. The documentation is quite thorough and well written which is great for getting started. Still, python-OBD is a pretty niche library and the user-base is relatively small. Therefore the error resources available online are quite limited. Thankfully, the library developers had the forethought to include a debugging log.

Python-OBD allowed me to move away from using hexadecimal encoded PIDs and instead accomplish the same result from simple library calls. Instead of sending the command “010C” and getting a hex response I was now able to write “cmd = OBD.commands.RPM” and obtain the numerical result by writing “response = connection.query(cmd)”. This same level of simplicity transferred across all python-OBD library calls, making communication with the vehicle a seemingly straightforward and easy task.

When I did run into difficulties, I would often have to read numerous forum posts and comments to piece together a solution that resolved my issue. One example that highlights this well is when I was trying to test out switching between a synchronous connection and an asynchronous connection using event triggers in the application. Initial attempts were unsuccessful and according to the debug logs, this had something to do with a response taking too long. Worried that this might be the downfall of my goal to make this application seamlessly switch between connections, I began to investigate the elm327.py file that I had previously updated. After ctrl-f searching for all of the “AT” commands and comparing them, I found one that made a write call rather than a send call. Going out on a limb, I removed the “delay=1” from write call only.

After testing a connection switch again with the debugger log active, I generated a different error. The connection was now failing because the auto connect built into the python-OBD library was unable to choose a baud rate for the Bluetooth connection. At this point, I ran a few sample programs with both connection modes and stored the information from the debugger log. Looking through the output, I found that the same baud rate was always being chosen for both asynchronous and synchronous connections. With this information, I manually added the appropriate baud rate to the arguments of any connection calls made after the initial connection on application startup and successfully switched between asynchronous and synchronous connections. Debugging an issue this obscure would have been impossible without help from numerous forum posts and the built in error log. However, there was also no way to find this solution by simply googling my error and copying the results I found.

**PyQt5**

Since my experience with python was very limited leading up to this project I asked Dr. Appleton for suggestions on GUI libraries. He suggested that I play around with PyQt. After going through some online tutorials and building sample user interfaces, I found that everything felt intuitive and easy to use. Knowing my project advisor had experience with the library and feeling confident I could implement it, I decided it would be a great fit.

Starting to build the GUI was a combined process of learning the tools available to me with PyQt5 while refining the design of the UI to give the user the experience I desired. In the end, I went with a multi-tabbed window view. Each tab is comprised of a widget with multiple elements and organized using a grid layout. A widget in this case is a container for anything the user can see or interact with. It can refer to something as small as a button or as big as a whole tab layout. The process of setting this up has a strange nested composition to it but offers a very organized structure.

I first construct a layout and populate it with different widgets (buttons, labels, etc.) without it visually existing in the GUI. I then create a QWidget and set its layout to the appropriate already constructed layout. At this point I am able to add these QWidgets to a QTabWidget, creating tabs in the main window.

The only graphical tool that I was unable to achieve with PyQt5 was the graphs for the trip logging section. I found a library called pyqtgraph, which claimed to be compatible with PyQt5. I ran into issues when first attempting to do any graphing as a result of certain function calls from the time module being deprecated in python 3.8 that were used in the source code for pyqtgraph. By downloading the most recent version of the library directly from github I was able to avoid this issue and successfully plot graphs inside a PyQt5 window.

**Hardest Part**

The most challenging part of my project, after the initial learning curve of utilizing new technologies, was trying to get PyQt5 and Python-OBD to work as seamlessly as possible together. Thinking of these technologies as front-end(pyqt) and back-end(pythonOBD) helps illustrate their functions and the difficulties I ran into. When I first began developing my project I kept these functions completely separate. This allowed me to more easily debug any issues I ran across and figure out how exactly to achieve the design and usability that I wanted. Once I was content with the functionality from the Python-OBD code and the look of the PyQt5 GUI, I began meshing the two together.

I quickly realized I would need to handle one glaring problem, scope and access. My first step in alleviating this issue was to reorganize my code and provide natural access to as many things as possible. While this did fix almost all of my issues in respect to updating the GUI as a result of communication from the vehicle, there were still variables that I could not access in all the necessary places. I played around with quite a few ideas and eventually settled on creating global variables for the vehicle connection and trip times. This handled the biggest issue of using the two libraries together. However, I was still running into difficulties when trying to properly display information received from the vehicle. After a lot of trial and error with different type casting and processes of extracting data from Python-OBD objects, I was able to successfully convert data into a format where it could be displayed by PyQt5.

I recognize that global variables are generally not the best way to resolve scope issues and that there are almost certainly other ways to achieve the same result. I am comfortable with my use of global variables here as my project as a whole is relatively small, there is no possibility of concurrency issues, and certain variables, such as the OBD connection, are truly used throughout the whole program.

**What I Learned**

Aside from working with my vehicle and making a tool that I can use in my everyday life, the most enjoyable part of this project was easily the knowledge I gained along the way. I had watched some YouTube videos to learn the basics of Python about a year before starting this project, but besides that, all of the technologies I used were completely new to me.

After brainstorming project ideas with a friend, he started talking about things he had done with his raspberry pi. I was immediately interested in trying to use it in conjunction with my car in some way. Now that I have successfully implemented the pi in my project and had a few months to play around with it, I am extremely excited to use it for future projects.

My only in depth experience with developing a GUI prior to this application was for web apps. I was initially concerned this knowledge would not translate very well to a Python GUI library. While there are differences, I was pleased to find that PyQt5 was relatively easy to pick up and use. Developing a GUI in Python really helped me solidify and grow my knowledge and understanding of the common tools involved in building a user interface as well as my knowledge of the Python language.

Through the course of developing this application, I learned a lot about what OBD actually is and what can be done with it. Since I was not directly working with any of the vehicle source code I was unable to gain an in depth, hands dirty, knowledge on the subject. However, I did go down some very interesting rabbit holes in the research process of this project that have reinforced my desire to continue learning.

Perhaps the biggest takeaways is that I now feel confident in my ability to digest documentation in order to implement a library, use new technologies to complete a project, and debug issues that are more complex than a simple stackoverflow post.

**Do Differently**

I imagine it is a normal process to get to the end of a large project and think about how things could have gone differently. While this certainly rings true, I do not have any regrets about the way everything turned out. If I had more time and we were not in the middle of a pandemic, I would have enjoyed testing OBD commands that my vehicle does not support. This would allow me to see if I could add anything useful to the application even though the feature would not be available for some vehicle manufactures. Another big thing that feels like it could use some work is my organization of the whole project. Since I built the GUI code and the OBD code separately, the end product feels a bit smashed together. I think the project would benefit from some refactoring, which I may do as a learning process in the future, but in the end I had no desire to blow up my completed and working project a few weeks before it was due.

**Grade**

There are 90 points available for my project consisting of five main categories. I was able to implement all of my proposed features except for the emissions test function. I was initially thinking that I would attempt this feature when I gained a good understanding of Python-OBD but working with it only made me more cautious. After all, I had to manually edit source files just to get the connection to work. In the online documentation it lists a mode called mode 6, which runs various tests on different vehicle sensors. A large disclaimer states this mode has passed software tests but has not been tested on an actual vehicle. After completing all other aspects, I decided that these points were not worth potentially damaging my car.

I successfully implemented a graphical user interface, made the application accessible on a raspberry pi/display, provided trip logging, and established communication with the vehicle to query trouble code information as well as monitor desired variables. According to my rubric, I completed 80 out of 90 points and should receive an A-.

**Conclusion**

I am very pleased with the process and results of my project. I was able to incorporate a topic that has always been very interesting to me while simultaneously stepping outside of my comfort zone and learning a variety of new skills. By combining the data logged from this application with my dual dash cameras, I essentially equipped my vehicle with a (destructible) black box. In the future, I plan to keep playing around with the different abilities of Python-OBD and hopefully add some fun features to my application.