University of Colorado, Colorado Springs

CS 3030-001 Topics Computer Science

Python pyOBD

Armand Moussaouyi

Derek Cotton

11 May 2019

-------------------------------------------------------------------------------------------------------------------------------

For this class project the team with pyOBD to read, document, analyze, and display the trace for a given vehicle. Originally, we proposed OpenXC as our subject matter, but due to the limitations outside of the Ford brand and requirements for proprietary gear, we favored working with pyOBD instead. This package is not as developed, but more flexible than OpenXC.

The open source pyOBD library consists of several protocol files, 9 test files and 11 package files that bring serial input to an API. The protocol files identify the type of serial input from a vehicle via ELM327. ELM327 is the apparatus that supports the OBD II connection, that is mandated for support on U.S. vehicles since 2006. The test files (pytest) allow for the testing of connections and the data being pushed onto the port by the vehicle. The package files are the main of the program and operate as follows:

**obd.py 🡪**

-Primary API, where the commands originate from in order to extract decoded data from the stream.

**OBDCommand.py 🡪**

Creates the command object that is used to encode commands.py and create the request/response objects to be filled as they are returned. As they are returned, they are decoded by decoders.py/codes.py (and scaled if necessary via UnitsAndScaling.py).

**elm327.py 🡪**

This file pulls protocols and verifies the serial output as valid for the request being submitted.

**pyserial 🡪**

This module is imported to work with the raw serial data.

**Vehicle Input**

(Data such as steering wheel angle (-600-600 degrees), torque at transmission (-500-1500 Nm), engine speed (0-16382 RPM), vehicle speed (0-655 km/h) [remains positive even when moving backward, we can use gear status to compute direction], and accelerator pedal position (0-100%)...etc. )

**Workflow**

In Phase I ([optional] Phase II was not attempted) of our project we sought to read and format vehicle data. (NOTE: our JSON data parser intention was abandoned with OpenXC. Our second goal is to design a dashboard application (python GUI) to display our data in an intuitive manner.)

Reading the vehicle data proved to be an arduous and failed exercise that created the limitation in the work accomplished. After trying several different available drivers for the ELM327 cable that was purchased, none were effective in communicating with the vehicle. To remedy this without the ability to obtain a replacement, an ELM327 emulator was implemented at recommendation of the pyOBD testing README. It took the combination of com0com (port emulator) and OBDSim to be able obtain a response from the API. Prior to this a Flask server (serve\_em) and Raspberry Pi platform was built with the purpose of creating a public display output and to control the input on the vehicle. This remained underdeveloped, due to failure of the ELM327 hardware.

The run.py initiates the GUI program and implements the Dashboard.py format and Car.py class. Dashboard formats the car information into the GUI and starts trace via the pyODB library. The Dashboard implements the tkinter library for GUI construction. In hind-sight, there may have been an opportunity in using Kivy for touch support, PyGObject for base-class flexibility and support for GIMP (image), or PyQt for multi-platformed support. We could also further implement combinations of libraries as there is a need to expand the capabilities for our future work.

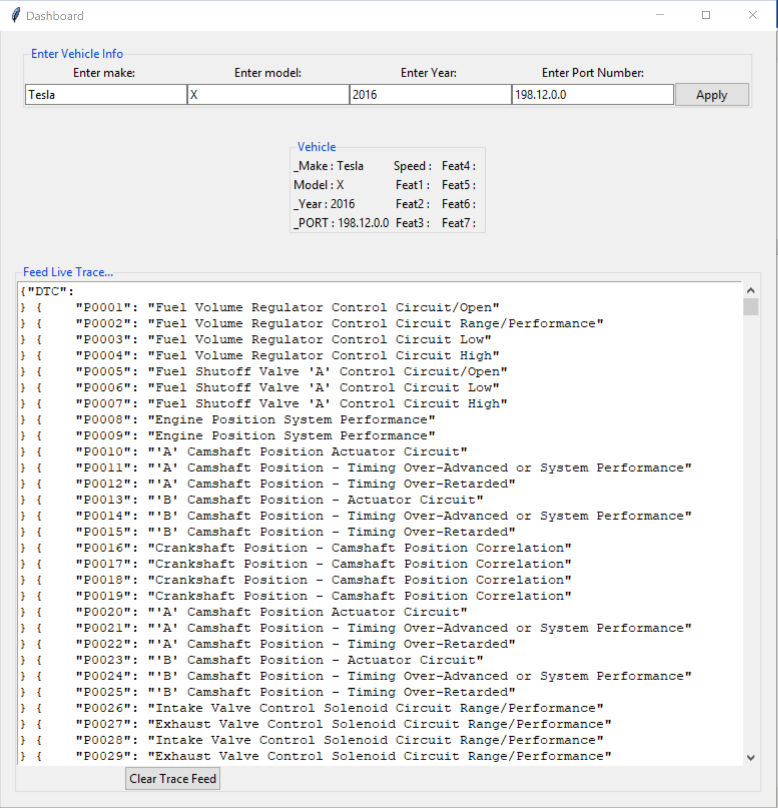
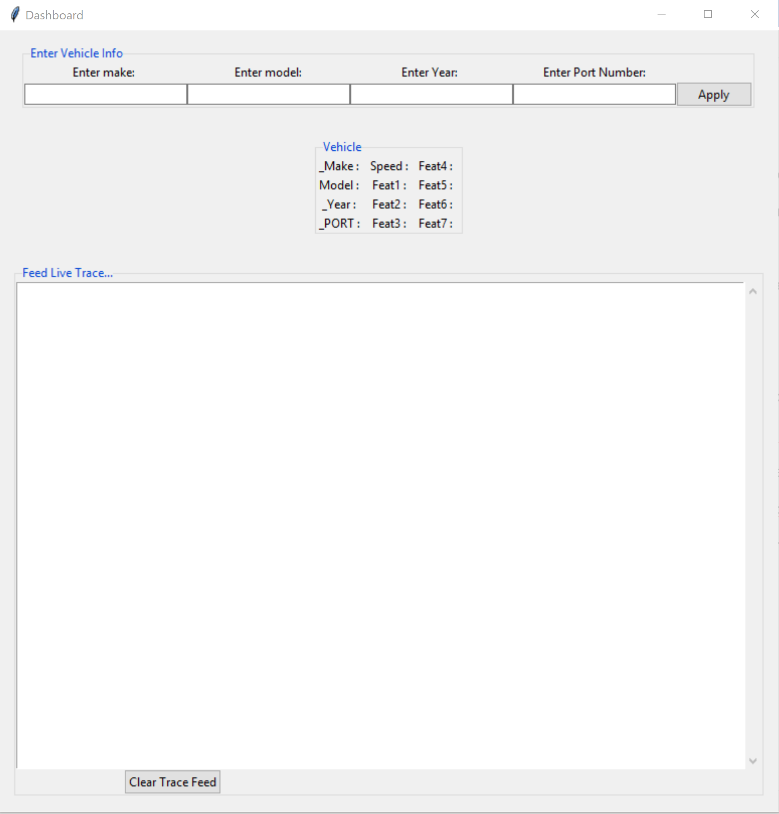
There are several reasons why we may implement several things, the number one driver would be how to create native vehicle control, in a vehicle mounted unit. A comprehensive control unit for auto computer systems offers the most potential and should be highest priority. For now, the goal of creating a solution for this problem in auto computing requires much more development than anticipated.

The safety of a self-contained projects contrasted with a comprehensive programming goal, shows how much more difficult and team-oriented software manufacturing needs to be for success. Failures can teach us much, like how to leverage the many different software types available and how to choose and research a deeper problem. This was our case. In the future, planning should have some level of exploratory programming, and most importantly, dependencies of the project may be the most difficult issues. They also may be what you are most ill-equipped to deal with. Resolving dependencies first will allow for more flexibility and must be prioritized. Overall, this exercise was difficult and a lot of squeeze for little juice, but is still viable for future engagement.

Armand worked on the GUI to format the returned vehicle data, while Derek worked on reading the vehicle data, server and interfacing the GUI. It can be found at:

<https://github.com/armandmous/OpenXC>

The car class parses and process the car data by providing an abstract data type that can be called from and manipulated through an interface. The dashboard class creates a GUI that display the car data. The GUI is design using the python tk library components. By entering the car make, model, year, and port number we can extract the car data once the apply button is pressed. The port number is the car server ip address which contains the car data. All source code is executed by running the run.py file which creates a dashboard object.

Below is sample display of the GUI interface:

As of now we have the system reading data from sample files instead of the server. If had more time we would design a working server to hold our data and the car class will read data directly from the socket. If we had extra time, we would refine our gui interface as well.

References:

http://openxcplatform.com/

<http://openxcplatform.com/about/data-set.html>

<https://creativecommons.org/licenses/by/4.0/deed.en_US>

http://www.obdtester.com/pyobd