makersportal.com

Python Datalogger - Using pySerial to Read Serial Data Output from Arduino

Joshua Hrisko

10-12 minutes

After I became proficient with Arduino I found myself trapped in its development environment (IDE). I needed to escape from the simplicity of the serial port and transform the platform into a usable engineering tool. I tried saving to SD cards, but decided adding more hardware was superfluous; I tried Bluetooth and WiFi, but again, barring specific Internet of Things applications, I found those to be roundabout ways of achieving what I wanted: a simple datalogging system. I ultimately arrived at Python's pySerial library, which reads directly form the serial port and was a complete solution to my predicament.

Using pySerial is much easier than one might expect, as the most fundamental implementation consists of only three lines of code:

```
import serial
ser = serial.Serial('/dev/ttyACM0')
ser_bytes = ser.readline()
```

These three simple lines read a single row of data from the serial port. In the case of Raspberry Pi, the serial port (on my Arduino) is located at '/dev/ttyACM0'. You may also find yours there, or at an integer increment (ttyACM1, ttyACM2, etc.), or perhaps a different address completely. Check your Arduino IDE serial port for the exact location. After successfully reading a line from your Arduino, verify that it is in the desired format. I chose to print and read a single line, but you may prefer comma separated or similar formats. The code above isn't particularly interesting, but it verifies that pySerial is working and that you are parsing data correctly from your serial port. Once the method

above is understood, we can advance onto loops and recording data in real-time.

NOTE: I will be using a DHT11 temperature sensor to produce data on the Arduino end. Since this is a tutorial on reading data from the serial port using Python, not Arduino, I recommend visiting a DHT11 tutorial to learn how to print temperature data from the sensor to the serial port (see here, or here).

I will be using a while loop and keyboard interrupt (CTL-C) to loop and halt the datalogging. The data from the serial port also needs to be converted from unicode to float (or another datatype) so that the data can be processed in Python. In my case, I am using "utf-8" to decode, which is Arduino's default encoding and the most commonly used character encoding format. You may also notice the 'ser.flushInput()' command - this tells the serial port to clear the queue so that data doesn't overlap and create erroneous data points. Sometimes the conversion via float() can create errors, but this is due to overprinting from the Arduino's end. If you receive such an error, restart the python script and try again.

```
import serial
ser = serial.Serial('/dev/ttyACM0')
ser.flushInput()

while True:
    try:
        ser_bytes = ser.readline()
        decoded_bytes =
float(ser_bytes[0:len(ser_bytes)-2].decode("utf-8"))
        print(decoded_bytes)
    except:
        print("Keyboard Interrupt")
        break
```

Now we have a working real-time data printer in Python. Decidedly, this isn't particularly interesting because we aren't saving or plotting the data, so we'll cover how to do both of those next.

Saving Serial Data to CSV File

In the code below I have implemented a way to save the serial data in real-time to a .csv file.

```
import serial
import time
import csv

ser = serial.Serial('/dev/ttyACM0')
ser.flushInput()

while True:
    try:
        ser_bytes = ser.readline()
        decoded_bytes =
float(ser_bytes[0:len(ser_bytes)-2].decode("utf-8"))
        print(decoded_bytes)
        with open("test_data.csv","a") as f:
            writer = csv.writer(f,delimiter=",")

writer.writerow([time.time(),decoded_bytes])
    except:
        print("Keyboard Interrupt")
        break
```

Now we have a working datalogger! This is as simple as it gets, and it's remarkably powerful. The three lines that start as: " with open("test_data.csv","a") as f: " look for a file called 'test_data.csv' and create it if it doesn't exist. The "a" in parentheses tells Python to append the serial port data and ensure that no data is erased in the existing file. This is a grand result because it not only takes care of saving the data to the .csv file, but it creates one for us and prevents overwriting and (most times) corruption. How considerate!

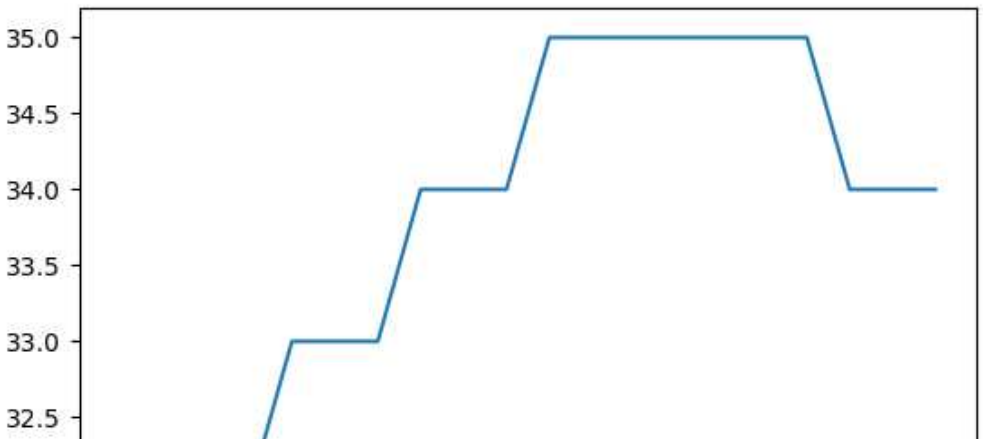
Live Plotting Serial Data Using matplotlib

To take things a bit further, I decided to aggrandize the content here and include a real-time plot. I find real-time plotting a useful tool when acquiring data of any kind. It is much easier to find faults in visualizations than it is to find them in streaming values.

NOTES: while I was using Raspberry Pi, I came across an issue between reading the serial port, saving to .csv, and updating the plots. I found that updating the plot occupied a lot of processing time, which resulted in slower reading of the serial port. Therefore, I advise anyone who is using the method below to assess whether you are reading all the bytes that are being outputted by the Arduino. I found that I was missing bytes or they were getting backed up in the queue in the buffer. Do some tests to verify the speed of your loop. This will prevent lost bytes and dropouts of data. I found that my loop took roughly half a second to complete, which means that my serial port should not be outputting more than 2 points per second. I actually used 0.8 seconds as the time between data records and it appeared to catch all data points. The slow loop is a result of the plotting, so once you comment out all of the plot code, you will get a much higher data rate and .csv save rate (just as above).

```
import serial
import time
import csv
import matplotlib
matplotlib.use("tkAgg")
import matplotlib.pyplot as plt
import numpy as np

ser = serial.Serial('/dev/ttyACM0')
ser.flushInput()

plot_window = 20
y_var = np.array(np.zeros([plot_window]))

plt.ion()
```

```
fig, ax = plt.subplots()
line, = ax.plot(y_var)

while True:
    try:
        ser_bytes = ser.readline()
        try:
            decoded_bytes =
float(ser_bytes[0:len(ser_bytes)-2].decode("utf-8"))
            print(decoded_bytes)
        except:
            continue
        with open("test_data.csv","a") as f:
            writer = csv.writer(f,delimiter=",")

writer.writerow([time.time(),decoded_bytes])
        y_var = np.append(y_var,decoded_bytes)
        y_var = y_var[1:plot_window+1]
        line.set_ydata(y_var)
        ax.relim()
        ax.autoscale_view()
        fig.canvas.draw()
        fig.canvas.flush_events()
    except:
        print("Keyboard Interrupt")
        break
```



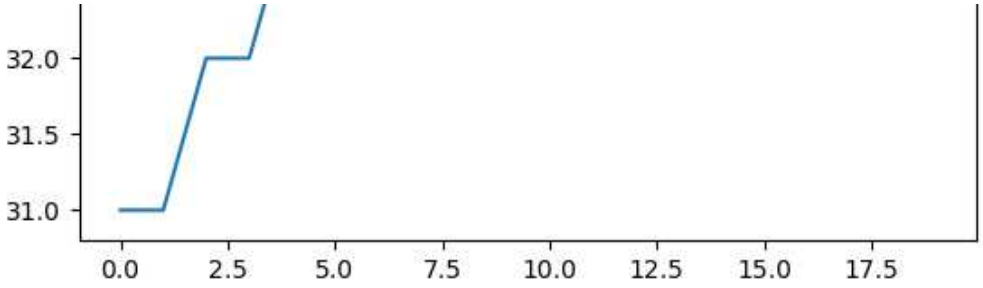


Figure 1: DHT11 Live Plot from Serial Port

Plot produced by matplotlib in Python showing temperature data read from the serial port. During this plot, the sensor was exposed to a heat source, which can be seen here as an increase from 31 to 35 degrees C.

This tutorial was created to demonstrate that the Arduino is capable of acting as an independent data logger, separate from wireless methods and SD cards. I found Python's pySerial method a while ago, and I wanted to share its capabilities with makers and engineers that may be having the same issues that I was encountering. Printing data to Arduino's serial port and then reading it through Python gives the user the freedom to investigate the data further, and take advantage of the advanced processing tools of a computer, rather than a micro controller. This method also allows the user to bridge the gap between live data and laboratory measurements. With real-time datalogging via the serial port, one can mimic the laboratory setup of acquisition, analysis, and live observation. Often, with Arduino the user is trapped in the serial port, or is relegated to communication via protocols, which can take time and energy. However, importing the data into Python frees the user of middle-men and allows the data to be processed in any way preferred. I use pySerial often, whether for recording temperature data using thermocouples, or high-frequency hall sensor measurements to monitor moving parts. This method can be used to take pressure measurements in the laboratory, or even record calibration data to improve your instrumentation accuracy; the possibilities are truly endless.

See More in Arduino and Python:

Arduino, Data Analysis, Programming, Python, Most PopularFebruary 25, 2018pySerial, Python, programming, serial, bytes, datalogger, data,

Arduino, Analyze, Data Analysis, Serial Port, DHT11, matplotlib, loop, while loop, csv, tkAgg, interrupt, figure, 2018 #1, Most Popular 201824 Comments

Previous

Heat Transfer of the Raspberry Pi Using Arduino, An Infrared Thermometer, and Type-K Thermocouple

Arduino, Data Analysis, Engineering, Editor's Picks, Raspberry Pi, Programming, PythonJoshua HriskoMarch 7, 2018Heat Transfer, Broadcom, heat, CPU, Arduino Uno, Arduino, Thermocouple, MLX90614, MAX6675, IR, Infrared, Wiring, Temperature, GitHub, Ceramic, Aluminum, Sparkfun, Adafruit, Python, Serial Port, Serial, pySerial, Loop, Stress, Convection, Conduction, Nusselt Number, Empirical, Heat Sink, Aero-Thermal, Raspberry Pi, Heat Transfer Arduino, Arduino Heat, Arduino Heat Engineer, Arduino Heat Transfer, Arduino Engineering Heat Transfer, Engineer, Engineering

Next

Academy Awards Data Mining, Statistical Analysis, and Visualization Using Wikipedia and Plotly

Data Analysis, MATLAB, Lifestyle, ProgrammingJoshua HriskoFebruary 25, 2018Oscars, Academy Awards, Data Analysis, Data Mining, Data, MATLAB, Plotting, Visualization, Plotly