April 26, 2020

GitHib readme for this commit.

Mrphysh

John Brigham

Asheville North Carolina

**C++/Arduino:** The Arduino is now processing 6 inputs. Four are from pendulums and two are from rotating pendulums. The rotating pendulums catch the time with every other beat. This is done with a little trick; an extra Boolean switch. The C++ sends out the values as a flat file. I collect one number out of 19. Tis results in about 7000 numbers a day. The rotating pendulums are much slower and much less data is recorded. The microcontroller sends the numbers out as a flat file; just one after another, about 7 a minute.

The Python catches the numbers, adds a datetime stamp and puts them in a sqlite3 database. The database can be defined with a DATE field. But then I realized that the Python pulls out the value as a string anyway. The initial Python script sorted the numbers by size, into their groups but then that fell away. Since then I have just left them in a flat file and sorted them later. This seems better anyway.

The theme of this GitHub repository starts here. I have millions of rows of data with only two columns; a datetime stamp that the python sees as a string and the corresponding 7-digit integer.

I have made progress and if this can benefit anyone, I am cool with that. And for a beginning/intermediate Python programmer, there is a lot of understandable code.

**Day by day:**

**Good\_date\_increment.py**

I want to look at the data, one day at a time. The idea is to search the database for one day and give a summary in one row. Then search the next day and put a summary in one row. So, a database with 400,00 rows can be summarized in 60 rows, one for each day.

The Python search command looks like this

***c.execute ("SELECT \* FROM first\_table WHERE column\_one > '%s'" %date\_var)***

I am exactly sure about the '%s'" % variable treatment. And the Python uses a > to compare two strings. Oookay.

The trick is to modify the date\_var. this seems to work:

***date = datetime.strptime(date\_var, "%Y-%m-%d %H:%M:%S") # this sets up the variable***

***date = str(date + timedelta(days = 1))***

***date\_var = date***

So, go to the date and then pick the window, not with a [< and >] but using the fetchmany

***for row in c.fetchmany(search\_number): #use this format for a number of rows.***

***# In the final rendition, this might be 6000***

***# there are about 7000 records per day.***

***var\_one = row[0] #row[0 is the datetime***

I have been using a 20-digit field for the datetime. This gives the possibility of looking at morning versus evening.

**Sorting the data.**

**Python\_3col\_simple.py**

The numbers come in pairs; data with a datetime stamp.

This is pretty easy; by row, sort the number with ‘if’ statements.

There are lots of

*if holder!= None:*

**Averages:**

**Python\_rolling\_averages.py**

Within these [if/else] categories, the total is summed, the number of numbers is incremented and when it is over, the average is calculated. I have been storing the number of numbers for QC purposes.

**Rolling average**

I put variables in the ’if’ statements and then adjust them with every day. So that if the average is moving around the script should be able to follow them. This seems okay, but has not been carefully checked.

**Insert into new table.**

This pretty straightforward. I have them set up as functions. In my simple application, the Python and database need to be in the same folder.

*def dynamic\_data\_entry\_averages():*

*c.execute("INSERT INTO summary\_rolling\_windows(column\_one,column\_two,column\_three,column\_four) VALUES (?,?,?,?)",( date\_var, col\_two\_avg , col\_thr\_avg , col\_for\_avg ) )*

*conn.commit()*

Putting the table in a different database is the simple matter of making two connection objects and two cursor objects. It requires some very careful attention to detail, but no real trick. The **idea being, of course, to summarize a year’s data into 365 rows.**

**Good\_two\_databases.py**