USER GUIDE for dataAccess.py

An API for a SQL Database
of Continuous Glucose Monitoring Data
in the GluGo© Ecosystem of Apps

By: Caleb, Adnan, Kamran, Sam, Saba, Alec

Date last modified: December 5, 2016

Contents:

Introduction:

DataAccess is a Python class providing an Application Programmer's Interface (API) to interact with a SQL database of information related to wearers of Continuous Glucose Monitoring sensors (CGM). The class offers a level of abstraction as the methods in the class perform the SQL commands while the user merely supplies the required arguments to a given function. This document will clear up questions the user might have about the database and how the user can use it to your advantage.

About using SQL

SQL ("sequel") is a structured English query language - a universal programming language to create and access databases. SQL is non-procedural, meaning it does not have conditional statements (if-then-else, or do while loops).

SQL creates relational databases (where the data has relationships). A relational database is essentially linking 2D tables with rows and columns. It is important to note that this is not necessarily how the data is stored - rather, it is how it is retrieved and inserted (it is how we interact with the database).

There are four basic commands in SQL:

(1) Insert
(2) Select
(3) Update
(4) Delete

SQLite is a library that implements a SQL database. To program the database, sqlite3 was utilized but limits the code and will be pointed out as needed.. SQLite is a very powerful library and is widely used around the world. It is different from many other SQL databases as it is capable of writing to disk instead of just to a server.

The DataAccess API facilitates communicating with the database for these SQL commands.

1) **`INSERT INTO TABLE_NAME [(column1, column2, column3,...,columnN)] VALUES (value1, value2, value3,...,valueN);`**

2) **`SELECT column1, column2, column3 FROM table_name`**
    a) **`SELECT * FROM table_name`**
        i) **`This allows you to select all of the column names`**

3) **`UPDATE table_name SET column1 = value1, column2 = value2 WHERE [condition]`**

    a) **`Not including WHERE will result in the updates happening in every row in the specified column.`**

4) **`DELETE CAN NOT BE IMPLEMENTED IN SQLITE3 and thus no API call is available.`**

Our database:

Thus far, three out of the four basic SQL commands have been programmed, while the delete function has not been written as SQLite does not allow the deletion of columns. Be wary when adding to the database, it is recommended to create a copy of the database so that if an error is committed, it is still possible to go back to the previous version.

The way things are organized in the database is through a primary key - "**`at`**". The table below explains what else is in the database. For right now, it is important to know that "at" refers to the timestamps in the csv files that we are dealing with and is the unique identifier for each row/record in a database. The database is indexed based on the "at" column.

Each patient has their own table, in a common database. Thus, a table of patient IDs (PID) have been linked so that they each point to a table with further information specific to each patient. This is not a connection you can directly modify after creating an instance of the dataAccess class.

For each patient object, we have the following default table:

at	Timestamp (integer not NULL)	4 bytes
id	row_id (integer)	4 bytes
gluc	Glucose value (integer)	4 bytes
bolis	Bolis value (integer)	4 bytes
carb	Carbohydrate (integer in grams)	4 bytes
name	Name of the device (eg: Dexcom, text)	Length + 1
model	Model (G4/ G5, text)	Length + 1
evnt	Type of recording (Regular or calibration, text)	Length + 1

Features that are not implemented (yet):

- Normalized database for the device names: which is a separate table for device names "normalized" to reduce redundancy. Normalize in this context means to remove similar device names and put them into a separate, connected table.
- Being able to use a patient's name instead of their PID integer to refer to their specific table in a common database
- updateCol() function: partially due to time and the ability to present this database as soon as possible, and partly because there is a function (updateRow()) that allows the user to update all columns (for whichever rows the user has specified).

Function: `insertData(path)`

Description: This is the primary function used to create and populate a local database given a comma-separated value (.csv) file from a patient's CGM file. For example, a file downloaded from Dexcom's Clarity website for a patient wearing a Dexcom G5 CGM sensor. This function checks for duplicate events, thus a new row is inserted only if the event has a unique date/time stamp.

Argument:

1. **`path`**: a filepath to a csv file of CGM data

A sample of the data (header and a few sample lines) is shown below:

Index	timestamp	glucose value	transmitter time
27	6/13/2016 0:04	108	738857
28	6/13/2016 0:09	105	739157

Returns: None

Side effect(s): If the function is successful then new data will be added to the existing database and committed for storage.

Limitations: `insertData()` does not update rows of data in the database, rather, this method should only be used to add multiple rows of new data at one time (and create a database file if a database doesn't exist).

Use the function call `insertData(`*`path to a csv file`*`)`

```
from dataAccess import dataAccess as DB
import dateTime as DT

# path to filled out Database that has already been created with
patient information from the provided CSVs
dbpath="lab.db"

# Connect to the Database by creating an instance of dataAccess
kateDB = DB("Kate", dbpath)
```

```
#call the function on the Kate table with new data
kateDB.insertData("C:\...\Desktop\DBlab\Kate_G4_June1_Aug16_2016.c
sv")
```

Function: getData(start = None, stop = None, desc = False, datetimes = True)

Description: getData takes in a start and stop time and returns a 2D array of all the column data for the rows you have specified. The primary function is to get data from the database.

Arguments:

2. Start: this is the date that the user would like to start getting data from (the OLDEST date). If the user does not pass anything, the function will get data from the oldest timestamp in the datafile.
3. Stop: this is the date that the user would like to end the data collection at. If the user does not pass anything, the function will get data up until the newest timestamp in the datafile.
4. Desc: this stands for descending and refers to the order in which you would like the matrix to be organizing. If desc=True, the matrix will be returned in the order: newest → oldest. If desc=False, the matrix will be returned in the order: oldest → newest.
5. Datetimes: if this is true, then the user will get back datetime objects, otherwise the user will get back unixtimes.

Return value: getData returns a 2D array based on whatever timestamps you have given the function. Can be used in the same way the user would usually use a 2D array.

Limitations: getData doesn't work if the uses gives the function a stop time that comes before the start time (Eg. June 10, 2016 (start) and May 5, 2016 (stop)) does NOT work.

Use the function call below, where db is a path to the database:

```
from dataAccess import dataAccess as DB
import dateTime as DT

#path to filled out Database that has already been created with
patient information from the provided CSVs

dbpath = “lab.db”

#connect to the Database by creating an instance of dataAccess
kateDB = DB(“Kate”, dbpath)
```

```
#call the function on the Kate table which will return all of
the data in the database for Kate
print(kateDB.getData())
```

Function: addColumn(colName, vals)

Description: addColumn takes in a column name and a dictionary of values which are in {datetime : value} format, appends the column on the end of the database, and populates said column with the specified values. Datetime objects are the "key" in the dictionary. This function is used to add a new column to the table to help track patterns in the data.

Arguments:

(1) colName: column name

(2) vals: dictionary of {datetime : value, datetime : value}

Output: new column is appended to end of the database and populated with values.

Limitations: This function only appends a new column under the user specified name.

Deletion of columns not possible:
http://stackoverflow.com/questions/8442147/how-to-delete-or-add-column-in-sqlite
http://stackoverflow.com/questions/5938048/delete-column-from-sqlite-table

Use the function call below:

```
from dataAccess import dataAccess as DB
import dateTime as DT

#path to filled out Database that has already been created with
patient information from the provided CSVs

dbpath = "lab.db"

#connect to the Database by creating an instance of dataAccess
kateDB = DB("Kate", dbpath)

#call the function on the Kate table to create a new column
kateDB.addcolumn('"slope"', values)

^values is a python dictionary that contains datetimes and the
value you want added in the column
Below is a sample

[{\"1465778389\":666},{\"1465777482\":666},{\"1465778681\":666}]
```

Function: updateRow(datetime, valueList)

Description: UpdateRow allows the user to change the specified column names in a single table row where ‘at’ = datetime.

Arguments:

(1) datetime: datetime object that indicates which row is being updated
(2) valueList: dictionary of values to update the indicated row where the key of each value is the column name to be updated.

Output: None.

Limitations: The user needs to know the exact date and time, and the datetime must be an object. The list of values the user wishes to enter must be in an existing column in the database. Otherwise, the user will need to use addColumn() and then updateRow()

How to use this function with Python: Create a variable, set it to a specific date as a dateTime object, and create a dictionary of values to enter/change.

What to use this function for: Adding new or changing existing values to individual rows in each table in the database.

```
from dataAccess import dataAccess as DB
import dateTime as DT

#path to filled out Database that has already been created with
patient information from the provided CSVs

dbpath = “lab.db”

#connect to the Database by creating an instance of dataAccess
kateDB = DB(“Kate”, dbpath)

#call the function on the Kate table with new data for the
specific row
kateDB.updateRow(datetime, valueList)

^valueList would look like this
{"at": 2413, "carb": 204}
```

Function: getColumnNames()

Description: getColumnNames returns the list of column names from the database to the user. This function can be helpful as it will tell the user what columns have already been made and will aid in creating new columns for data. In SQLite 3, this function finds the table in the database and then returning a list of all of the column names.

Arguments: none

Return Value: A list of all of the column names within the database. If the database is empty, the function will return an empty list.

Limitations: This function only returns the column names. No numerical data is returned to the user.

Use the function call below:

```
from dataAccess import dataAccess as DB
import dateTime as DT

#path to filled out Database that has already been created with
patient information from the provided CSVs

dbpath = “lab.db”

#connect to the Database by creating an instance of dataAccess
kateDB = DB(“Kate”, dbpath)

#call the function on the Kate table and get a list of the
column names
print(kateDB.getColumnNames())
```

Practice working with the database:

(1) Download dataAccess.zip from #database channel on Slack
(2) Unzip the file and rename emptyDB.sql to your desired database filename, all in your project folder
(3) Open testDB and go through it - there are comments explaining how to query the database
(4) There is also a lab to find glucose slopes. Remember that if you are modifying the database, make a backup copy first.