Intermediate Python Programming

Description: Context Managers and Thread Pool Futures

Instructions

- Please follow all instructions closely and read this in its entirety before writing ANY code. There is no penalty for going “above and beyond” the assignment requirements. However, failure to meet all required parts can affect your grade. Please consult the rubric for each section.
- For this assignment you will create a single “raw” Python file with the name “Lab5.py”.
- We will be using SQLite in this lab, so make sure you don’t skip over the setup section.
    - While database access is not a topic we are focusing on in this course, it is a good skill to have.
    - You will be given all required commands, so if your SQL is rusty, do not worry.

Setup

- The first thing we must do is verify that SQLite is installed and accessible from Python. This lab requires at least version 2.6 of SQLite.
    - If you have any problems with your specific installation, please check: http://www.sqlitetutorial.net/download-install-sqlite/
- Create a folder to contain your files, called “Lab5”
- To verify SQLite is accessible from Python, create a new file in that folder called TestSQLite.py and add the following code:

```
import sqlite3
from sqlite3 import Error

def create_connection():
    try:
        conn = sqlite3.connect(':memory:')
        print(f"You are running SQLite version {sqlite3.version}")
    except Error as e:
        print(e)
    finally:
        conn.close()

if __name__ == '__main__':
    create_connection()
```

- Next, save the file and execute it from the terminal/shell:

```
$ python TestSQLite.py
```

- The output should show something like:

```
You are running SQLite version 2.6.0
```

- If the version is not at least 2.6 or there is an error, consult the installation documentation above. It may be that you require an update using pip:

  ```
  $ pip install pysqlite
  ```

- You do not need to turn in TestSQLite.py, it is just to verify you have the necessary components installed.

Part I

In the first part of the lab you are going to create a SQLite database and populate it with people that have random names.

- To create random names you will need access to first & last name lists. Download the FirstNames.txt and LastNames.txt files and place them in the Lab5 folder.
- Next, create a new file called Lab5.py in the same folder as the text files.

generate_people():

- Inside Lab5.py, create function with the following signature:

```
def generate_people(count):
```

- This function will create a list of "count" tuples (ID, First Name, Last Name), each of which has a random first and last name.
  - *Note: The ID should simply be the counter you use in your loop/comprehension.*
  - Create two lists, last_names and first_names and initialize them to [].
  - Using a file context manager, open the LastNames.txt file for read-only access:

    ```
    with open('LastNames.txt', 'r') as filehandle:
    ```

  - Inside the "with" context block, loop through the file using the filehandle's readline() function and place each name read into last_names.
    - A better approach would be to use a comprehension using filehandles's readlines() function.
    - If you are noticing weird characters at the end of each name (carriage returns or line feeds), don't forget about String's rstrip() function.
  - Once you have the last_names list populated, perform a similar process for the first_names list.
  - Now that both name lists are loaded you must create the tuple list of random names.
    - Create a for loop (or better yet, a comprehension) that creates "count" tuples in a list called "names", where:
      - The first item of the tuple is the ID of the person – use the counter variable of the for loop
      - The second item is the first name. Use random.randint() to determine a name to pull out of first_names. Don't forget to use the len() function on first_names when getting a random index from randint.
      - The third item is the last name. Pretty much the same process as the first name item above.

- Don’t forget you create a tuple with the following syntax:

```
my_tuple = (item1, item2, item3)
```

- Now, create a “main” block at the bottom pf Lab5.py and test the generate_people() function:

```
if __name__ == "__main__":
    people = generate_people(5)
    print(people)
```

- Sample output:

```
[(0, 'JONATHAN', 'VILLARREAL'), (1, 'HARRIETTE', 'MAY'), (2, 'GALE', 'SAVAGE'), (3,
'MALISSA', 'BRANDT'), (4, 'MERCY', 'RAY')]
```

- Note: If the output is missing the parentheses around each person, then you did not create tuples. If there is an extra set of square brackets around each person then that is a list of lists, not a list of tuples.

Extra Credit:

- Use ThreadPoolExecutor to spawn two threads, each of which loads one of the name text files and returns the name list.
- This will require a worker/future function that accepts a filename as a parameter and returns the processed list of names.

Scoring:

Criteria	**Points**
Lab setup (imports, variables, etc.)	5
Correct use of file’s context manager	5
Text files correctly processed	5
Correct random name generation and return	5
Tester implementation/functionality	5
TOTAL	**25**
EXTRA CREDIT	
Use of ThreadPoolExecutor to load text files in 2 threads	10

Part II

Now you are going to create a SQLite database and insert a set of random names into it.

- First, add the following import statements at the top of the Lab5.py file in addition to any existing statements:

```
import sqlite3
from sqlite3 import Error
```

- Next create the following variables just under the import statements, outside of any method:

```
people_db_file = "sqlite.db"  # The name of the database file to use
max_people = 500  # Number  of records to create
```

- Now, create the create_people_database() function:

```
def create_people_database(db_file, count):
```

Where:
  - The db_file parameter represents a filename.
  - The count parameter represents the number of records to generate.

  - The method will first create a connection to SQLite and then utilize a context manager to handle its lifetime:

```
conn = sqlite3.connect(db_file)
with conn:
    ### The rest of the code will go here ###
```

  - Next, create a SQL command string to create the "people" table, open a cursor and then execute the command (the triple quotes are important for line continuations):

```
sql_create_people_table = """ CREATE TABLE IF NOT EXISTS people (
    id integer PRIMARY KEY,
    first_name text NOT NULL,
    last_name text NOT NULL); """
cursor = conn.cursor()
cursor.execute(sql_create_people_table)
```

  - The next step is to truncate any existing data from the table with the following commands:

```
sql_truncate_people = "DELETE FROM people;"
cursor.execute(sql_truncate_people)
```

  - Next, generate a list of person tuples by calling the generate_people() function created in the previous step:

```
people = generate_people(count)
```

- Finally, create the query to add the people records and use a for loop to execute the statement:

```
sql_insert_person = "INSERT INTO people(id,first_name,last_name) VALUES(?,?,?);"

for person in people:
    #print(person) # uncomment if you want to see the person object
    cursor.execute(sql_insert_person, person)
    #print(cursor.lastrowid) # uncomment if you want to see the row id
cursor.close()
```

- Now, back in the "__main__" section, execute the create_people_database() function:

```
create_people_database(people_db_file, max_people)
```

There will be no direct output from this method, but you can check the results by using a third party database UI, like DBeaver (https://dbeaver.io/).

Scoring:

Criteria	**Points**
create_people_database() definition and functionality	20
TOTAL	**20**

Part III

The next part of the lab is to create a context manager class called PersonDB that will provide read access to the database created in part II:

```
class PersonDB():
```

- __init__() signature:
  `def __init__(self, db_file=''):`
  - For this method, all that needs to be done is to store the db_file parameter value into a self.db_file variable. You will use it in the __enter__() method.
- __enter__() signature:
  `def __enter__(self):`
  - __enter__ represents the method that is called when a class object of type PersonDB is used within a "with" context.
  - This method will initiate the connection to a SQLite database indicated by self.db_file.

  - Recall from Part II how to call the connect() method of sqlite3. Store the resulting connection object in a variable called self.conn.
  - Return self

- __exit__() signature:
  `def __exit__(self, exc_type, exc_value, exc_traceback):`
  - __exit__ represents the method that is called after the last line of a with block is executed.
  - This method will close the database connection by calling self.conn.close().

- load_person() signature:
  `def load_person(self, id):`
  - This method will attempt to load a "people" record with the id of "id".
  - Create a sql string to select a specific record from the database:

```
sql = "SELECT * FROM people WHERE id=?"
```

  - Next, open a cursor and use that cursor to execute the sql command:

```
cursor = self.conn.cursor()
cursor.execute(sql, (id,))
records = cursor.fetchall()
```

    - Note: the execute command accepts the sql command string and a tuple of parameters to substitute for the question marks in the command string. Notice the comma with nothing after it? That is to ensure `(id,)` results in a tuple, not a single evaluated value.

  - Next create a variable called result and assign to it a tuple that represents an empty record if nothing is found in the database:

```
result = (-1,'','')  # id = -1, first_name = '', last_name = ''
```

  - Finally, check to see if the "records" list contains any data and if it does get the first value out of the list. If you simply return records, you will get a list with one item in it instead of just the one item.

```
if records is not None and len(records) > 0:
    result = records[0]
cursor.close()
return result
```

- Now, create a new method *outside of the class* called test_PersonDB() to test load_person() using a context manager:

  ```
  def test_PersonDB():
  ```

  - Using a "with" block with a PersonDB object, attempt to load and print three person records:

    ```
    with PersonDB(people_db_file) as db:
        print(db.load_person(10000))  # Should print the default
        print(db.load_person(122))
        print(db.load_person(300))
    ```

- Next, go to the "__main__" section and call test_PersonDB() to test the method you just created.

Sample output (your results WILL vary due to the random nature of person creation):

```
(-1, '', '')
(122, 'ELISA', 'SINGLETON')
(300, 'GENNY', 'WEBER')
```

Scoring:

Criteria	**Points**
Implement PersonDB class	
__init__() implementation/functionality	5
__enter__() implementation/functionality	5
__exit__() implementation/functionality	5
load_person() implementation/functionality	5
Use test_PersonDB() to test PersonDB	5
TOTAL	**25**

Part IV

In the final part of the lab you are going to utilize a ThreadPoolExecutor to load all of the people records in the database. You are going to implement most of this section on your own, but here are a few requirements that should assist your thinking process:

First, create a helper method (outside the PersonDB class) that will act as an intermediary to PersonDB's load_person method:

```
def load_person(id, db_file):
    with PersonDB(db_file) as db:
        return db.load_person(id)
```

You can use this to simplify the future target for the threads.

- You are to use ***exactly*** 10 worker threads (the argument to ThreadPoolExecutor)
- You must create the ThreadPoolExecutor with a context manager "with" block
- The future target is the load_person() function defined above
- You can assume that the record ids to load are 0 through max_people – 1
    - Yes, that means you will spawn max_people futures…
- Try to launch all of the future workers using a comprehension that returns a list of future objects so you can use wait() or as_completed() to sync back up on the main thread
- Put all the resulting records in a list and print it out (will be a big list)

Extra Credit:

- Once all records are loaded in a list [], sort the list by last name and first name (multiple sort keys).
- Print that list.
- Hint: Review the lambdas demo

Scoring:

Criteria	**Points**
Proper creation and use of ThreadPoolExecutor	5
Spawning of futures	15
Assembly of results of futures into a single list	10
TOTAL	**30**
EXTRA CREDIT	
Sort the records by last name then first name	5