**Project 1: File Format Conversions**

**Description**

Converting between 3 plain text data representations: CSV, JSON, and XML. You need to write Python 3 code in the starter code provided that can read in a given file, save them into some common internal representation, and output them in the desired format. You may only use built in modules of Python 3. And, we strongly encourage you to make use of the "csv", "json", and "xml.etree" modules.

**What needs to be done**

You need to submit a file named "project.py". There are 6 functions that need to be implemented.

To run your code, I've included some code in the starter file that runs one of the testcases. You are welcome to modify/delete this code as you see fit. What each of the test cases do is provide a formatted string to one of the read\_\*\_string functions, saving the returned object to a variable, then passing this object to one of the write\_\*\_string functions and checking that the string returned is formatted correctly. All possible combinations of read and write functions are tested.

**Intermediate Object**

What should the object returned by the read\_\*\_string functions (and loaded into the write\_\*\_string function)? It is up to you, think about what information needs to be stored to determine the correct output. You can use lists, dictionaries, tuples, custom objects, or anything else. My test cases don't care about the form of the object.

**Details**

**Order of Columns/Attributes**

Although the order of the columns/attributes doesn't matter in practice, for ease of testing, order the columns/attributes of file in lexicographical order.

**CSV Specific Instruction**

You need to have a header line denoting the columns.

**JSON Specific Instructions**

The tabular JSON format that you will be using for input and output involves a file with a single array. For each record (e.g. line in a csv file), it becomes one object in the JSON array. A record object has a property (i.e. key) for each column, and a value which is the value of that record.

**XML Specific Instructions**

XML has a lot of freedom with how you structure your data. For this project, in each XML file, there should be a single "data" node, with as many "record" nodes within it as needed. In each record, there will be an column node corresponding to each column. In each column node, the text content should be the value for that record. There should be no attributes for any element.

Example:

<data>

     <record>

          <id>1</id>

          <name>Josh Nahum</name>

     </record>

     <record>

          <id>2</id>

          <name>Tyler Derr</name>

     </record>

</data>

Note: This output has been prettified, the correct output is all on one line (no extraneous whitespace).

**What are the example datasets?**

There are two example data files provided for you (from <https://support.spatialkey.com/spatialkey-sample-csv-data/> with some minor modifications):

* "\*crime\*" contains 7,584 crime records, as made available by the Sacramento Police Department.
* "\*realestate.\*" is a list of 985 real estate transactions in the Sacramento area reported over a five-day period, as reported by the Sacramento Bee.

**Project 2: CREATE, INSERT, SELECT**

**Description**

You will be creating a python module that emulates the behaviour of the built-in module, "sqlite3". Your module will be able to execute SQL statements corresponding to: creating tables, inserting rows, and selecting rows.

**What you need to do**

All of the code you write for this project must be in "project.py".

Each test case imports "project.py" and expects to find a "connect" function with one parameter. It passes a filename (specifically "test.db"), which isn't used in this project. The "connect" function should return an object that has two methods "execute" and "close". For each SQL statement in a test case, the test case will pass that SQL statement as a string to the "execute" method. The method should return an empty list, unless a select statement was executed that yielded rows. In that case, a list of tuples with each denoting a row, should be returned. The "close" method doesn't do anything, yet.

There is no need to write any database output to any files. Persistence will be covered in a later project. Each test will be done on a clean, empty database.

Your program is allowed to use any built in modules of python, with the exception of sqlite3. sqlite3 is the reference implementation for this project, your output will be compared against it for correctness. Importing the sqlite3 module in project.py will be considered academic dishonesty for these projects.

You may (and should) write additional functions and classes in "project.py" to perform the needed actions.

**SQL statements**

All SQL keywords will be in ALL\_CAPS. For this project, the SQL keywords you need to handle are as follows:

CREATE TABLE

INSERT INTO VALUES

SELECT FROM ORDER BY

SQL data types (for this project) and their python equivalents are:

|  |  |
| --- | --- |
| **SQL** | **Python 3** |
| NULL | None |
| INTEGER | int |
| REAL | float |
| TEXT | str |

**SQL Syntax**

**CREATE TABLE**

Example: CREATE TABLE stocks (symbol TEXT, price REAL, qty INTEGER);

Creates a table named "stocks", with three columns: column "symbol" composed of text strings, column "price" composed of real valued numbers (floating point), and column "qty" composed of integers.

**INSERT INTO**

Example: INSERT INTO stocks VALUES ('GOOG', 40.4, 7);

Inserts the row ('GOOG', 40.4, 7), into the table named "stocks".

**SELECT \***

Example: SELECT \* FROM stocks ORDER BY price;

Returns all the rows in the table named "stocks". The rows must be ordered by the column "price" ascending.

Given the above examples, the result should be a list of tuples: [('GOOG', 40.4, 7)]

**SELECT columns**

Example: SELECT qty, symbol FROM stocks ORDER BY price, qty;

Returns all rows, but only showing the qty and symbol values in that order. The rows are sorted by price (and where price is the same, by qty).

**Test Guarantees**

* All tests will be legal SQL (no syntax errors, inserting into nonexistent tables or data type violations).
* All select statements will have "FROM" and "ORDER BY" clauses.
* In "CREATE TABLE" statements every attribute will have a data type and no constraints.
* All table and column names will start with a letter (or underscore) and be followed by 0 or more letters or numbers or underscores.
* No TEXT will contain single quotes.

**NULL Values**

Some tests will have NULL values. You need to represent this fact with the python object, None. When returning NULL values from a select statement, you should yield a python None object.

**Example**

Example Test File of SQL statements ("006: Different Types"):

import project

conn = project.connect("test.db")

conn.execute("CREATE TABLE students (col1 INTEGER, col2 TEXT, col3 REAL);")

conn.execute("INSERT INTO students VALUES (3, 'hi', 4.5);")

conn.execute("INSERT INTO students VALUES (7842, 'string with spaces', 3.0);")

conn.execute("INSERT INTO students VALUES (7, 'look a null', NULL);")

result = conn.execute("SELECT col1, col2, col3 FROM students ORDER BY col1;")

result\_list = list(result)

expected = [(3, 'hi', 4.5), (7, 'look a null', None), (7842, 'string with spaces', 3.0)]

print("expected:", expected)

print("student: ", result\_list)

assert expected == result\_list

Correct output from "006: Different Types":

expected: [(3, 'hi', 4.5), (7, 'look a null', None), (7842, 'string with spaces', 3.0)]

student: [(3, 'hi', 4.5), (7, 'look a null', None), (7842, 'string with spaces', 3.0)]

**Project 3: UPDATE, WHERE, JOIN**

**Description**

This project will be extending the "project" module, which solves Project 2, and emulates the behaviour of the built in module, "sqlite3". Your module will be able to execute SQL statements corresponding to: (from Project 2) creating tables, inserting rows, and selecting rows; (and adding) various syntax improvements, updating and deleting rows, joins and the distinct keyword.

**project.py**

You should probably copy the project.py file you submitted to Project 2, or build off the instructor's Project 2 solution. Note: you are responsible for understanding every line of code in your submissions, even if that code was written by the instructor.

**What you need to do**

All of the code you write for this project must be in "project.py".

There is no need to write any database output to any files. Persistence will be covered in a later project. Each test will be done on a clean, empty database.

Your program is allowed to use any built in modules of python, with the exception of sqlite3. sqlite3 is the reference implementation for this project, your output will be compared against it for correctness. Importing the sqlite3 module in project.py will be considered academic dishonesty for these projects.

You may (and should) write additional functions and classes in "project.py" to perform the needed actions.

**SQL Keywords**

All SQL keywords will be in ALL\_CAPS. For this project, the SQL keywords you need to handle are as follows (red are new for this project):

CREATE TABLE

INSERT INTO VALUES

SELECT FROM ORDER BY

UPDATE SET DELETE

WHERE DISTINCT

LEFT OUTER JOIN ON

**New SQL Syntax**

**Extensions over Project 2**

* Single quoted strings may have single quotes in them (escaped with a preceding single quote), you need to store and return the python string after removing this escaping. Example: 'My dog''s name' -> "My dog's name".
* Any time a column name is used it may be qualified with a table name ("id" is the same thing as "students.id" if the table being SELECT'ed is "students").
* The "\*" column name may be qualified or included as one of multiple columns (SELECT \*, name ..., SELECT student.\*, name ...).

**INSERT INTO**

The INSERT INTO statement can specify the columns and order of the VALUES being inserted: INSERT INTO students (id, name) VALUES ... If not all the columns of a table are specified, the absent columns will have NULL inserted. Multiple rows can be inserted with a single INSERT INTO statement:INSERT INTO students (name, grade) VALUES ('Josh', 3.7), ('Tyler', 2.5), ('Hangchen', 3.9);

**WHERE clause**

SELECT (as well as, UPDATE and DELETE) statements can have a WHERE clause that specifies what rows should be processed. SELECT \* WHERE id > 4; To make things easier, all WHERE clauses will be in this form: WHERE column\_name operator value The column\_name may be qualified. The operator will be one of: >, <, =, !=, IS NOT,IS. The value will be a constant (not a different column or expression). There won't be any parentheses, ANDs or ORs in the projects.

**DELETE statement**

DELETE works much like UPDATE, but instead removes all rows from a table, (unless a WHERE clause is added, in that case only removes the rows which pass the predicate).DELETE FROM students; DELETE FROM students WHERE id > 4;

**UPDATE statement**

You will need to add the UPDATE statement: UPDATE table\_name SET col1 = value1, col2 = value2; An UPDATE statement changes the associated columns to the value. For simplicity, the value will always be a constant. If a WHERE clause is added, only those rows will be updated (see above). UPDATE student SET grades=4.0 WHERE name = 'Josh';

**DISTINCT keyword**

The DISTINCT keyword specifies you only want the unique values (no duplicates). For simplicity, we the tests will only use the DISTINCT keyword with a single output column. Example: SELECT DISTINCT column\_name FROM ... ORDER BY column\_name;

**LEFT OUTER JOIN clause**

The LEFT OUTER JOIN is the only join you need to implement for this project. If will be of the form:SELECT "columns" FROM "table\_a" LEFT OUTER JOIN "table\_b" ON "column from table\_a" = "column from table\_b" ORDER BY "columns"; As shown above, the ON clause will always match on equality with a column from the left table then the right table. For simplicity, all column names will be qualified in queries involving joins.

**Project 4: Data Definition Language**

**Description**

This project will involve adding more Data Definition Language (DDL) constructs, mostly corresponding to dropping tables and transactions.

**Files in Project4/**

* project.py: You need to supply this file. You should probably copy the project.py file from Project3 or from Project3 solution released by the instructor.

**What you need to do**

All of the code you write for this project must be in "project.py".

There is no need to write any database output to any files. Persistence will be covered in a later project. Each test will be done on a clean, empty database.

Your program is allowed to use any built in modules of python, with the exception of sqlite3. sqlite3 is the reference implementation for this project, your output will be compared against it for correctness. Importing the sqlite3 module in project.py will be considered academic dishonesty for these projects.

You may (and should) write additional functions and classes in "project.py" to perform the needed actions.

**Changes to the connect function**

In order to ensure that your solution matches the behaviour of python's SQLite3 module, the connect function should accept a total of three parameters:

1. database: the filename of the database
2. timeout: your code should ignore this parameter
3. isolation\_level: your code should ignore this parameter

These two new parameters are used to instruct sqlite3 to not do some of the fancy optimizations that your code won't be sophiticated enough to perform. Example call to connect:

conn = connect("test.db", timeout=0.1, isolation\_level=None)

**Raising Exceptions**

There are times when your code should raise an exception (see <https://docs.python.org/3/tutorial/errors.html>). The type and message of the exception don't matter (and will never matter in any project). But, not raising an exception when expected will fail the test case.

**Test Categories**

**Regression**

These tests should pass if you completed Project 3. They should pass automatically at the start of the project. If these fail, you've made a regression (you have broken previously functional code).

**Connections**

These tests check that your code accepts multiple connections to the same database (only one database per test in this project).

**Create\_Drop\_Table**

You need to implement the CREATE and DROP table commands. The CREATE TABLE command you have already implemented, but there's one minor addition. Your code should raise an exception if a CREATE TABLE statement attempts to make a table that already exists.

If the CREATE TABLE command includes "IF NOT EXISTS" (example: CREATE TABLE IF NOT EXISTS students ...), then nothing should happen if the table already exists. You also need to implement the DROP TABLE statement (which has an optional "IF EXISTS" clause to not raise an error if the table doesn't exist). Examples:

DROP TABLE students;

DROP TABLE IF EXISTS students;

**Transactions**

* Transactions begin with a BEGIN TRANSACTION statement. Raise an exception if a BEGIN TRANSACTION occurs within an open transaction.
* Transactions end with COMMIT TRANSACTION (ROLLBACK tests are in a different category). Raise an exception, if a COMMIT TRANSACTION is attempted without a prior BEGIN TRANSACTION.
* The only statements the tests will happen within transactions are SELECT, INSERT, UPDATE and DELETE (the Data Manipulation Language statements).
* If a statement is before a BEGIN statement or after a COMMIT statement, that statement is in "autocommit mode" meaning that any locks needed are acquired before the statement runs, and released when the statement is complete. This behavior is disabled after a BEGIN statement (manual transaction mode).
* A shared lock is needed to read (SELECT). Shared locks block exclusive locks.
* A reserved lock is needed to write (INSERT, UPDATE, and DELETE). Reserved locks block exclusive locks and other reserved locks.
* An exclusive lock is needed to commit a write. Specifically, if the transaction has a reserved lock, it must be promoted to an exclusive lock upon commit. Exclusive locks block all other locks.
* Locks are released upon commit.
* If a lock can't be granted when requested, raise an exception.
* Writes are only visible to other connections upon commit (before then the transactions need to keep private copies of their writes).

Remember: sqlite only locks the entire database, not individual tables or rows.

**Recommended Implementation**

The way we recommend you handle transactions is make a copy of the database's tables when the transactions starts and do your reads and writes to the copy. Upon commit, copy the modified tables to the "real" database. If you implement the locking correctly, no other transaction should generate conflicts.

It may seem inefficient to copy the entire database for each transaction, and it is. The way sqlite actually does it is keeping private copies of all the pages of memory it writes to (and only writing to disk on commit), but that is too difficult to do from Python. So do the inefficient, but easy thing for this project, copy all the tables for each transaction.

**Transaction\_Modes**

Transactions can occur in one of three modes (BEGIN "mode" TRANSACTION:

* DEFERRED = locks are acquired when needed by a statement in the transaction
* IMMEDIATE = a reserved lock is acquired at start of the transaction and an exclusive lock is acquired when needed
* EXCLUSIVE = an exclusive lock is acquired at start of transaction

Transactions that don't specify a mode default to DEFERRED.

**Rollback**

The second way a transaction can end is with a rollback. All the changes made by the transaction needs to be reversed, and all locks released.

If you implement full isolation between transactions by only modifying a local copy of the database in each transaction, rollback is very simple, just don't copy your modified tables to the "real" database, just get rid of them.

You can only rollback within a manual transaction, if a connection tries to rollback outside of a transaction, raise an error.

**Project 5: Other Database Constructs**

**Description**

This project will involve adding more features of the SQL language to your DBMS. As this project will not use any features developed in Project 4, you may build off of either your Project 3 or Project 4 solution. There will not be a Project 4 Instructor Solution released.

**Files in Project 5**

* project.py: You need to supply this file. You should probably copy the project.py file from Project3 (or Project 4) or from Project3 solution released by the instructor.

**What you need to do**

All of the code you write for this project must be in "project.py".

There is no need to write any database output to any files. Each test will be done on a clean, empty database.

Your program is allowed to use any built in modules of python, with the exception of sqlite3. sqlite3 is the reference implementation for this project, your output will be compared against it for correctness. Importing the sqlite3 module in project.py will be considered academic dishonesty for these projects.

You may (and should) write additional functions and classes in "project.py" to perform the needed actions.

**Test Categories**

**Regression**

These tests should pass if you completed Project 3. They should pass automatically at the start of the project. If these fail, you've made a regression (you have broken previously functional code).

**DESC (Descending Ordering)**

The DESC keyword indicates that the collation should be reversed, so that values are ordered in a descending manner instead of the default (ascending).

SELECT \* FROM students ORDER BY name DESC, grade DESC;

**Default Values**

It is often useful to supply non-NULL default values for specific columns. For instance, if no grade is supplied for an insert statement, I may want to assume a grade of 0.0. The way you specify the default value for a column is in the CREATE TABLE statement:

CREATE TABLE students (name TEXT, grade REAL DEFAULT 0.0, id TEXT);

Additionally, if you want to insert a row with entirely default values, you can do so like:

INSERT INTO students DEFAULT VALUES;

**View**

Views are read-only named SELECT statements. They act like a table, but if any of the tables the underlying SELECT statements it draws from changes, the results returned are changed (on subsequent queries). Example:

CREATE TABLE students (name TEXT, grade REAL);

CREATE VIEW stu\_view AS SELECT \* FROM students WHERE grade > 3.0 ORDER BY name;

SELECT name FROM stu\_view ORDER BY grade;

For simplicity, none of the columns of the view's underlying tables will share names.

**Parameterized Queries**

Parameterized queries make it easy to reuse queries with wildcard (?) values pulled from the variables in the programming language interface. You need to implement a executemany method on your Connection class that accepts two arguments: a SQL statement with wildcard placeholders, and list of tuples with the values that should be slotted in. Example:

conn.execute("CREATE TABLE students (name TEXT, grade REAL, class INTEGER);")

conn.executemany("INSERT INTO students VALUES (?, ?, 480);", [('Josh', 3.5), ('Tyler', 2.5), ('Grant', 3.0)])

**Custom Collations**

Custom collation functions allow the user to supply a function that specifies the way a column should be ordered. Defines a function that takes two arguments (left and right): and the function returns -1 if left is less than right, 0 if they are even and 1 otherwise. Then the create\_collation method of the connection is called with two arguments (the name of the collation within SQL and the function itself). You need to write the create\_collation and update your SELECT-related code to accept such collations.

Example test case:

conn.execute("CREATE TABLE students (name TEXT, grade REAL, class INTEGER);")

conn.executemany("INSERT INTO students VALUES (?, ?, 480);", [('Josh', 3.5, 480), ('Tyler', 2.5, 480), ('Tosh', 4.5, 450), ('Losh', 3.2, 450), ('Grant', 3.3, 480), ('Emily', 2.25, 450), ('James', 2.25, 450)])

conn.execute("SELECT \* FROM students ORDER BY class, name;")

def collate\_ignore\_first\_letter(string1, string2):

string1 = string1[1:]

string2 = string2[1:]

if string1 == string2:

return 0

if string1 < string2:

return -1

else:

return 1

conn.create\_collation("skip", collate\_ignore\_first\_letter)

conn.execute("SELECT \* FROM students ORDER BY name COLLATE skip, grade;")

Be aware that DESC can be used with custom collations like so:

SELECT \* FROM students ORDER BY name COLLATE skip DESC, grade DESC;

**Aggregate Functions**

You need to support the aggregate functions: min and max. Because we aren't implementing the GROUP BY clause, any SELECT statement using an aggregate function will only return a single row.

Example:

SELECT max(grade) FROM students ORDER BY grade;

**Project 6: NoSQL**

**Description**

NoSQL-style databases are not characterized by any particular property, except that they have different goals than traditional SQL databases. For this project, you will be implementing a Document Store, a type of database with no schema, but can be useful none the less.

**Files in Project6**

* project.py: This is the file you must implement the project. You are welcome to import other modules (both builtin and other custom files) for use in this file. Within, you must create two classes, "Collection" and "Database". It should contain your name, MSU email, feedback, and any sources beyond this course you used.

**What you need to do**

All of the code for this project could be placed in the "project.py" file.

* Collection.\_\_init\_\_(self): This method initializes a collection.
* Collection.insert(self, document): This method takes a document (a python dictionary) and adds it to the collection. The collection needs to store its documents in insertion order.
* Collection.find\_all(self): This method returns a list of all the documents stored in the collection in insertion order.
* Collection.delete\_all(self): This method removes all the documents stored in the collection.
* Collection.find\_one(self, where\_dict):
  + This method returns the first document (in insertion order), that matches the where\_dict. The where\_dict is a python dictionary that contains key-value entries. If no match is found, return None.
  + If a document doesn't have each key in the where\_dict, it doesn't match. If a document has the correct keys, but doesn't have the same associated values, it doesn't match.
  + If a where\_dict is empty ({}), it matches all the documents.
  + Example: documents = {'age':27, 'name':'Josh', 'major':'CSE'}, where\_dict = {'age':27, 'major':'CSE'}, the document matches because it has all the entries in the where clause. If the document was: {'age':27, 'name':'Tyler', 'major':'Street'}, it wouldn't match. This document also doesn't match:{'name':'Grant', 'major':'CSE'}.
* Collection.find(self, where\_dict):
  + This method returns the all the documents which match the where\_dict (in insertion order). If no match is found, return an empty list.
* Collection.count(self, where\_dict):
  + This method returns the number of documents that match the where\_dict.
* Collection.delete(self, where\_dict):
  + This method removes the documents that match the where\_dict.
* Collection.update(self, where\_dict, changes\_dict):
  + This method adds/updates the documents that match the where\_dict with the changes\_dict.
* Collection.map\_reduce(self, map\_function, reduce\_function):
  + This method takes two arguments which are both functions ("map\_function" and "reduce\_function"). It applies the map function to each document, saving the each's result to a list. This list is passed to the reduce function. The result of the reduce function is returned.
  + The map function will be provided by the test. Example:
  + def find\_age(doc):
  + if 'age' in doc:
  + return doc['age']
  + return 0
  + Example reduce function: sum (the builtin sum function returns the sum of all the values in a list).
  + Example documents [{'age':4}, {'name':'Jim', 'age': 2}, {'happy':'go lucky'}]
  + The result of calling map\_reduce with the provided data is 6.
* Database.\_\_init\_\_(self, filename): The init method for the Database class. It takes a filename that is where the database will store its information.
* Database.get\_collection(self, name): Returns a Collection instance associated with the given name. If no such Collection exists, create an empty one and return it. Otherwise return the Collection associated with the name. Note: the returned Collection shouldn't be a copy. Changes made to the Collection should be reflected in the database.
* Database.get\_names\_of\_collections(self): Returns a list of (sorted) names of collections in the Database.
* Database.drop\_collection(self, name): Removes the collection associated with the given name from the Database.
* Database.close(self):
  + Saves the information in the Database to the file designated in the init method.
  + You can use whatever data format you want (JSON, XML, custom).
  + Then closes the database. This method will be called after working with the database.
  + After the close method is called, the Database will not be used again.
  + If a new Database is created using the same filename as a previously now closed Database, it should have the import the data and act like the original Database.
  + There can exist multiple, concurrent Database instances, but they will always have different filenames.

**Tips**

The tests for this project import and run your code. If your code outputs (prints) additional material, it will fail the test. I recommend a "debug\_mode" global variable that you can use to test if you want to print additional messages for debugging purposes.

You should consider trying to solve the tests by hand before implementing the project.