DATABASES AND PYTHON

There are many different variations of SQL, and some are suited to certain purposes better than others. The simplest, most lightweight version of SQL is SQLite, which runs directly on your machine and comes bundled with Python automatically.

SQLite is usually used within applications for small internal storage tasks, but it can also be useful for testing SQL code before setting an application up to use a larger database.

In order to communicate with SQLite, we need to import the SQLite module and connect to a database:

```
import sqlite3

connection = sqlite3.connect("test_database.db")
```

If the database you are connecting to doesn't yet exist, it will be created when the connect() function is executed. Here we've created a new database named test_database.db, but connecting to an existing database works exactly the same way.

Now we need a way to communicate across the connection:

```
c = connection.cursor()
```

This line instantiates a Cursor object. A cursor is a control structure that enables operations on a database. Our Cursor object 'c' will let us execute commands on our SQL database 'test_database' and return the results.

Now we can easily execute regular SQL statements on the database through the cursor like so:

```
c.execute("CREATE TABLE People(FirstName TEXT, LastName TEXT, Age INT)")
```

This line creates a new table named People and inserts three new columns into the table: text to store each person's FirstName, another text field to store the LastName, and an integer to store Age.

We can insert data into this new table like this:

```
c.execute("INSERT INTO People VALUES('Ron', 'Obvious', 42)")

connection.commit()
```

Here we've inserted a new row, with a Firstname of Ron, a LastName of Obvious, and an Age equal to 42. In the next line, we had to commit the change we made to the table to say that we really meant to change the table's contents - otherwise our change wouldn't actually be saved.

NOTE: We used double quotation marks in the string above, with single quotes denoting strings inside of the SQL statement. Although Python doesn't differentiate between using single and double quotes, some versions of SQL (including SQLite) only allow strings to be enclosed in single quotation marks, so it's important not to switch these around.

At this point, you could close and restart IDLE completely, and if you then reconnect to test_database.db, your People table will still exists there, storing Ron Obvious and his Age; this is why SQLite can be useful for internal storage for those times when it makes sense to structure your data as a database of tables rather than writing output to individual files. The most common example of this is to store information about users of an application.

NOTE: If you just want to create a one-time-use database while you're testing code or playing around with table structures, you can use the special name :memory: to create the database in temporary RAM like so:

```
connection = sqlite3.connect(':memory:')
```

If we want to delete the People table, it's as easy as executing a DROP TABLE statement:

```
c.execute("DROP TABLE IF EXISTS People")
```

(Here we also checked if the table exists before trying to drop it, which is usually a good idea; it helps to avoid errors if we happened to try deleting a table that's already been deleted or never actually existed in the first place.)

Once we're done with a database connection, we should close() the connection; just like closing files, this pushes any changes out to the database and frees up any resources currently in memory that are no longer needed. You close the database connection in the same way as with files:

```
connection.close()
```

When working with a database connection, it's also a good idea to use the with keyword to simplify your code (and your life), similar to how we have used with to open files:

```
with sqlite3.connect("test_database.db") as connection:

# perform any SQL operations using connection here
```

Besides making your code more compact, this will benefit you in a few important ways.

Firstly, you no longer need to commit() changes you make; they're automatically saved. Using with also helps with handling potential errors and freeing up resources that are no longer needed, much like how we can open (and automatically close) files using the with keyword.

Keep in mind, however, that you will still need to commit() a change if you want to see the result of that change immediately (before closing the connection); we'll see an example of this later in the section.

If you want to run more than one line of SQL code at a time, there are a couple possible options. One simple option is to use the executescript() method and give it a string that represents a full script; although lines of SQL code will be separated by semicolons, it's common to pass a multi-line string for readability. Our full code might look like so (type in and execute this code):

```
importsqlite3

with sqlite3.connect('test_database.db') as connection:

c = connection.cursor()

c.executescript("""

DROP TABLE IF EXISTS People;

CREATE TABLE People(FirstName TEXT, LastName TEXT, Age INT);

INSERT INTO People VALUES('Ron', 'Obvious', '42');

""")
```

We can also execute many similar statements by using the executemany() method and supplying a tuple of tuples, where each inner tuple supplies the information for a single command. For instance, if we have a lot of people's information to insert into our People table, we could save this information in the following tuple of tuples:

```
peopleValues = (

('Ron', 'Obvious', 42),

('Luigi', 'Vercotti', 43),

('Arthur', 'Belling', 28)
```

We could then insert all of these people at once (after preparing our connection and our People table) by using the single line:

```
c.executemany("INSERT INTO People VALUES(?, ?, ?)", peopleValues)
```

Here, the question marks act as place-holders for the tuples in peopleValues; this is called a parameterized statement. The difference between parameterized and non-parameterized code is very similar to how we can write out strings by concatenating many parts together versus using the string format() method to insert specific pieces into a string after creating it.

For security reasons, especially when you need to interact with a SQL table based on user-supplied input, you should always use parameterized SQL statements. This is because the user could potentially supply a value that looks like SQL code and causes your SQL statement to behave in unexpected ways. This is called a “SQL injection” attack and, even if you aren’t dealing with a malicious user, it can happen completely by accident.

For instance, suppose we want to insert a person into our People table based on user-supplied information.

We might initially try something like the following, assuming we already have our People table set up (type in and execute this code):

```
import sqlite3

# get person data from user

firstName = raw_input("Enter your first name: ")

lastName = raw_input("Enter your last name: ")

age = int(raw_input("Enter your age: "))

# execute insert statement for supplied person data

with sqlite3.connect('test_database.db') as connection:

c = connection.cursor()

line ="INSERT INTO People Values('"+ firstName +"','"+ lastName +

"',"+str(age) +")"

c.execute(line)
```

Notice how we had to change age into an integer to make sure that it was a valid age, but then we had to change it back into a string in order to concatenate it with the rest of the line; this is because we created the line by adding a bunch of strings together, including using single quotation marks to denote strings within our string.

If you're still not clear how this works, try inserting a person into the table and then print (line) to see how the full line of SQL code looks.

But what if the user's name included an apostrophe? Try adding Flannery O'Connor to the table, and you'll see that she breaks the code; this is because the apostrophe gets mixed up with the single quotes in the line, making it appear that the SQL code ends earlier than expected.

In this case, our code only causes an error (which is bad) instead of corrupting the entire table (which would be very bad), but there are many other hard-to-predict cases that can break SQL tables when not parameterizing your statements. To avoid this, we should have used placeholders in our SQL code and inserted the person data as a tuple (type in and execute this code):

```
import sqlite3

# get person data from user and insert into a tuple

firstName = raw_input("Enter your first name: ")

lastName = raw_input("Enter your last name: ")

age = int(raw_input("Enter your age: "))

personData = (firstName, lastName, age)

# execute insert statement for supplied person data

with sqlite3.connect('test_database.db') as connection:

c = connection.cursor()

c.execute("INSERT INTO People VALUES(?, ?, ?)", personData)
```

We can also update the content of a row by using a SQL UPDATE statement. For instance, if we wanted to change the Age associated with someone already in our People table, we could say the following (for a cursor within a connection):

```
c.execute("UPDATE People SET Age=? WHERE FirstName=? AND LastName=?",
(45, 'Luigi', 'Vercotti'))
```

Of course, inserting and updating information in a database isn't all that helpful if we can't fetch that information back out. Just like with readline() and readlines() when reading files, there are two available options; we can either retrieve all the results of a SQL query, using fetchall(), or retrieve a single result at a time, using fetchone().

First, let's insert some people into a table and then ask SQL to retrieve information from some of them (type in and execute this code):

```
import sqlite3

peopleValues = (

('Ron', 'Obvious', 42),

('Luigi', 'Vercotti', 43),

('Arthur', 'Belling', 28)

)

with sqlite3.connect('test_database.db') as connection:

c = connection.cursor()

c.execute("DROP TABLE IF EXISTS People")

c.execute("CREATE TABLE People(FirstName TEXT, LastName TEXT, Age INT)")

c.executemany("INSERT INTO People VALUES(?, ?, ?)",
 peopleValues)

# select all first and last names from people over age 30

c.execute("SELECT FirstName, LastName FROM People WHERE Age > 30")

for row in c.fetchall():

print row
```

We executed a SELECT statement that returned the first and last names of all people over the age of 30, then called fetchall() on our cursor to retrieve the results of this query, which are stored as a list of tuples. Looping over the rows in this list to view the individual tuples, we see:

```
>>>

(u'Ron', u'Obvious')

(u'Luigi', u'Vercotti')

>>>
```

The “u” before each string stands for unicode (as opposed to ASCII) and basically means that the string might contain complicated characters that can’t be represented by the usual basic set of characters we normally see in English text.

NOTE: Python 3 note: You won’t see the “u” before the strings in Python 3, because strings in Python 3 are always stored as unicode by default. In order to get this same functionality in Python 2, you can include the following line at the beginning of your script: from __future__ import unicode_literals

If we wanted to loop over our result rows one at a time instead of fetching them all at once, we would usually use a loop such as the following (modify your code to use this method of processing the results of the SQL query):

```
c.execute("SELECT FirstName, LastName FROM People WHERE Age > 30")

while True:

row = c.fetchone()

if row is None:

break

print row
```

This checks each time whether our fetchone() returned another row from the cursor, displaying the row if so and breaking out of the loop once we run out of results.

The None keyword is the way that Python represents the absence of any value for an object.

When we wanted to compare a string to a missing value, we used empty quotes to check that the string object had no information inside: stringName == ""

When we want to compare other objects to missing values to see if those objects hold any information, we compare them to None, like so: objectName is None This Booloean comparison will return True if objectName exists but is empty and False if objectName holds any value.

DRILL

1. Create a database table in RAM named Roster that includes the fields ‘Name’, ‘Species’ and ‘IQ’

2. Populate your new table with the following values:

1 Jean-Baptiste Zorg, Human, 122
2 Korben Dallas, Meat Popsicle, 100
3 Ak'not, Mangalore, -5

3. Update the Species of Korben Dallas to be Human

4. Display the names and IQs of everyone in the table who is classified as Human