- Create Table

Steps to Create a Table in SQLite using Python

1. **Import the SQLite3 Module:** Use import sqlite3 to access SQLite functionality in Python.
2. **Establish Connection:** Use the connect() method to establish a connection to your SQLite database.
3. **Create a Cursor Object:** The cursor() method creates a cursor object that allows you to execute SQL commands.

```
import sqlite3

# Create database and connection to it (con); & create cursor to interact with database.
con = sqlite3.connect('TestDatabase.db')
cursor = con.cursor()
print('DB initialized\n')
```

The SQL query to be executed can be written in form of a string, and then executed using the **cursor.execute()** method.

The results can be fetched from the server by using the **fetchall()** method.

Syntax:

```
query = 'SQL query;'
cursor.execute(query)
result = cursor.fetchall()
```

```
# Check version of SQLite & execute/retrieve query
query = 'SELECT sqlite_version();'
cursor.execute(query)
result = cursor.fetchall()
print('The SQLite version is ' + str(result))
```

Syntax

```
CREATE TABLE table_name (
    column1 datatype PRIMARY KEY,
    column2 datatype,
    column3 datatype,
    ...
    columnN datatype
);
```

```
# Write query to create table and store to value (creator_query)
creator_query = """
    CREATE TABLE IF NOT EXISTS Ingredient_Table (
        Name STR PRIMARY KEY,
        Measure STR NOT NULL,
        Count INT NOT NULL,
        Cost STR NOT NULL
    );
"""
cursor.execute(creator_query)
print('\nThe Ingredients Table has been created.\n')
```

Storage Class in SQLite

The SQLite DataTypes and their corresponding Python types are as follows
A storage class can be called a collection of similar DataTypes. SQLite provides the following storage classes:

Storage Class	Value Stored	Storage Class	Python Datatype
NULL	NULL	NULL	None
INTEGER	Signed Integer (1, 2, 3, 4, 5, or 8 bytes depending on magnitude)	INTEGER	int
REAL	Floating point value (8 byte IEEE floating-point numbers)	REAL	float
TEXT	TEXT string (encoded in UTF-8, UTF-16BE or UTF-16LE	TEXT	str
BLOB (Binary Large Object)	Data stored exactly the way it was input, generally in binary format	BLOB	bytes

SQLite column constraints

SQLite column constraints are rules applied to individual columns within a table to maintain data integrity and enforce specific conditions on the values stored in those columns.

- **NOT NULL:** Ensures that a column cannot contain `NULL` values. If an attempt is made to insert or update a row with a `NULL` value in a `NOT NULL` column, SQLite will raise an error.

- **UNIQUE:** Guarantees that all values in a column are distinct. No two rows can have the same value in a `UNIQUE` column. While `NULL` values are generally allowed in `UNIQUE` columns, each `NULL` is considered unique.

- **PRIMARY KEY:** Uniquely identifies each row in a table. A `PRIMARY KEY` column automatically implies `NOT NULL` and `UNIQUE` constraints. A table can only have one `PRIMARY KEY`.

- **DEFAULT:** Specifies a default value for a column if no value is explicitly provided during an `INSERT` operation.

```
# Write query to create table and store to value (creator_query)
creator_query = """
    CREATE TABLE IF NOT EXISTS Ingredient_Table (
        Name STR UNIQUE PRIMARY KEY,
        Measure STR NOT NULL,
        Count INT DEFAULT 0,
        Cost STR DEFAULT 'waiting'
    );
"""
cursor.execute(creator_query)
print('\nThe Ingredients Table has been created.\n')
```

Insert Data Using Column Names and Values

Syntax:

INSERT INTO table_name (column1, column2, column3, ...) VALUES (value1, value2, value3, ...);

```
# Define a static 'test' entry to the table with specified values
# Execute actually 'performs' the change we define
cursor.execute('INSERT OR IGNORE INTO Ingredient_Table (Name, Measure, Count, Cost) VALUES ("Potato", "Lbs.", 20, 1.02)')
# Commit 'solidifies' the change(s) we define'
con.commit()
```

Insert Data Using Functions & User Input

```
# Create a function which takes input from the user to populate ingredient entries
def entry_function(curse, connect):
    name = input('Ingredient Name: ')
    measure = input('What unit measure? (Lb, Oz, Qt, etc.): ')
    count = input('Currently in stock: ')
    cost = input('Dollar cost per unit: ')

    # The action of storing user values 'exists as' variable (enter)
    enter = """ INSERT INTO Ingredient_Table (Name, Measure, Count, Cost)
    VALUES (?, ?, ?, ?);
    """
    # Execute actually 'performs' the change we define
    curse.execute(enter, (name, measure, count, cost))
    # Commit 'solidifies' the change(s) we define'
    connect.commit()
    print('Ingredient added!')
```

Remember:

The SQL query to be executed can be written in form of a string, and then executed using the **cursor.execute()** method.

The results can be fetched from the server by using the **fetchall()** method.

Syntax:

```
query = 'SQL query;'
cursor.execute(query)
result = cursor.fetchall()
```