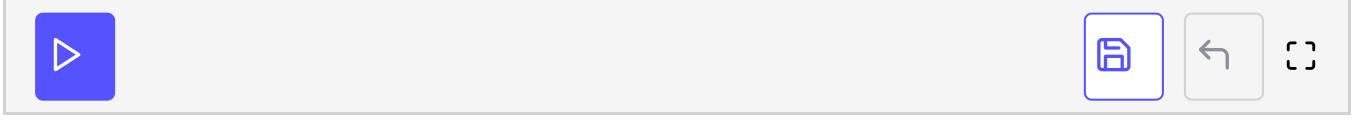
Creating a More Complex Test

Most code is a lot more complex than our **mymath.py** example. So let's create a piece of code that depends on a database being in existence. We will create a simple script that can create the database with some initial data if it doesn't exist along with a few functions that will allow us to query it, delete and update rows. We will name this script **simple_db.py**. This is a fairly long example, so bear with me:

```
import sqlite3

def create_database():
    conn = sqlite3.connect("mydatabase.db")
    cursor = conn.cursor()

    # create a table
    cursor.execute("""CREATE TABLE albums
                          (title text, artist text, release_date text,
                           publisher text, media_type text)
                       """)
    # insert some data
    cursor.execute("INSERT INTO albums VALUES "
                   "('Glow', 'Andy Hunter', '7/24/2012',"
                   "'Xplore Records', 'MP3')")

    # save data to database
    conn.commit()

    # insert multiple records using the more secure "?" method
    albums = [('Exodus', 'Andy Hunter', '7/9/2002',
               'Sparrow Records', 'CD'),
              ('Until We Have Faces', 'Red', '2/1/2011',
               'Essential Records', 'CD'),
              ('The End is Where We Begin', 'Thousand Foot Krutch',
               '4/17/2012', 'TFKmusic', 'CD'),
              ('The Good Life', 'Trip Lee', '4/10/2012',
               'Reach Records', 'CD')]
    cursor.executemany("INSERT INTO albums VALUES (?,?,?,?,?)",
                       albums)
    conn.commit()

def delete_artist(artist):
    """
    Delete an artist from the database
    """
    conn = sqlite3.connect("mydatabase.db")
```

```
    cursor = conn.cursor()

    sql = """
    DELETE FROM albums
    WHERE artist = ?
    """
    cursor.execute(sql, [(artist)])
    conn.commit()
    cursor.close()
    conn.close()


def update_artist(artist, new_name):
    """
    Update the artist name
    """
    conn = sqlite3.connect("mydatabase.db")
    cursor = conn.cursor()

    sql = """
    UPDATE albums
    SET artist = ?
    WHERE artist = ?
    """
    cursor.execute(sql, (new_name, artist))
    conn.commit()
    cursor.close()
    conn.close()


def select_all_albums(artist):
    """
    Query the database for all the albums by a particular artist
    """
    conn = sqlite3.connect("mydatabase.db")
    cursor = conn.cursor()

    sql = "SELECT * FROM albums WHERE artist=?"
    cursor.execute(sql, [(artist)])
    result = cursor.fetchall()
    cursor.close()
    conn.close()
    return result


if __name__ == '__main__':
    import os
    if not os.path.exists("mydatabase.db"):
        create_database()

    delete_artist('Andy Hunter')
    update_artist('Red', 'Redder')
    print(select_all_albums('Thousand Foot Krutch'))
```



You can play around with this code a bit to see how it works. Once you're comfortable with it, then we can move on to testing it.

Now some might argue that creating a database and destroying it for each test is pretty big overhead. And they might have a good point. However, to test certain functionality, you sometimes need to do this sort of thing. Besides, you don't usually need to create the entire production database just for sanity checks.

Anyway, this is once again for illustrative purposes. The unittest module allows us to override **setUp** and **tearDown** methods for these types of things. So we will create a setUp method that will create the database and a tearDown method that will delete it when the test is over. Note that the setup and tear down will occur for each test. This prevents the tests from altering the database in such a way that a subsequent test will fail.

Let's take a look at the first part of the test case class. Save the following code in a file named *test_db.py*:

```
import os
# import simple_db
import sqlite3
import unittest

class TestMusicDatabase(unittest.TestCase):
    """
    Test the music database
    """

    def setUp(self):
        """
        Setup a temporary database
        """
        conn = sqlite3.connect("mydatabase.db")
        cursor = conn.cursor()

        # create a table
        cursor.execute("""CREATE TABLE albums
                          (title text, artist text, release_date text,
                           publisher text, media_type text)
                       """)
        # insert some data
        cursor.execute("INSERT INTO albums VALUES "
                       "('Glow', 'Andy Hunter', '7/24/2012',"
                       "'Xplore Records', 'MP3')")

        # save data to database
        conn.commit()

        # insert multiple records using the more secure "?" method
        albums = [('Exodus', 'Andy Hunter', '7/9/2002',
                   'Sparrow Records', 'CD'),
```

```
                  ('Until We Have Faces', 'Red', '2/1/2011',
                   'Essential Records', 'CD'),
                  ('The End is Where We Begin', 'Thousand Foot Krutch',
                   '4/17/2012', 'TFKmusic', 'CD'),
                  ('The Good Life', 'Trip Lee', '4/10/2012',
                   'Reach Records', 'CD')]
        cursor.executemany("INSERT INTO albums VALUES (?,?,?,?,?)",
                           albums)
        conn.commit()

    def tearDown(self):
        """
        Delete the database
        """
        os.remove("mydatabase.db")
```

The **setUp** method will create our database and then populate it with some data. The **tearDown** method will delete our database file. If you were using something like MySQL or Microsoft SQL Server for your database, then you would probably just drop the table, but with sqlite, we can just delete the whole thing.

Now let's add a couple of actual tests to our code. You can just add these to the end of the test class above:

```
def test_updating_artist(self):
    """
    Tests that we can successfully update an artist's name
    """
    simple_db.update_artist('Red', 'Redder')
    actual = simple_db.select_all_albums('Redder')
    expected = [('Until We Have Faces', 'Redder',
                 '2/1/2011', 'Essential Records', 'CD')]
    self.assertListEqual(expected, actual)

def test_artist_does_not_exist(self):
    """
    Test that an artist does not exist
    """
    result = simple_db.select_all_albums('Redder')
    self.assertFalse(result)
```

The first test will update the name of one of the artists to the string **Redder**. Then we do a query to make sure that the new artist name exists. The next test will also check to see if the artist known as "Redder" exists. This time it shouldn't as the database was deleted and recreated between tests. Let's try running it to see what happens:

```
python -m unittest test_db.py
```

The command above should result in the output below, although your runtime will probably differ:

```
..
----------------------------------------------------------------------
Ran 2 tests in 0.032s

OK
```

Pretty cool, eh? Now we can move on to learn about test suites!