Result dictionaries with SQLite - a note on Lecture 7

Last week we looked at creating and executing SQL queries on our SQLite databases from Python programs. At the very end of the lecture you were introduced to an example similar to snippet 1 below.

```python
import sqlite3

# Establish database connection and cursor
con = sqlite3.connect("test.db")
cur = con.cursor()

# Execute a query
result = cur.execute("SELECT postcode, streetname, house_number FROM address")

# Do something with the resulting row tuples
for row in result:
    # Output each address as "<house_number> <streetname>, <postcode>"
    print(row[2] + " " row[1] + ", " + row[0])

# Close the database connection
con.close()
```

Code Snippet 1: Executing a simple query on an SQLite database from Python

Here, we establish a connection to our `test.db` database using the `sqlite3` package[1] and get a cursor from the connection to allow us to make queries. We then use the cursor to execute an SQL query to fetch the house number, street name and postcode for each property in the `address` table. To output the addresses to the terminal, we then use a for loop over the `result` cursor to read the values for each returned `row tuple` in turn.

You will notice that the ordering of the fields in the query maps to the index of the values in each `row` tuple. For example, the postcode field appears first in our query and is accessed via `row[0]`.

[1] https://docs.python.org/2/library/sqlite3.html

But what if you want to access fields by their name? Snippet 2 shows how to alter an attribute of the connection such that cursors do not return rows as a `tuple`, but as a special built-in `sqlite.Row` type - which amongst a few other things, allows access to fields returned in a row by name in a dictionary-like fashion.

```python
import sqlite3

# Establish database connection
con = sqlite3.connect("test.db")

# Set the row_factory attribute of the database connection to sqlite3.Row
# This causes result rows to be returned as sqlite.Row rather than tuples
con.row_factory = sqlite3.Row

# Now get the cursor
#   If we had set the row_factory after the cursor, the new factory would
#   not have been applied to the cursor and we'd still get tuples back...
cur = con.cursor()

# Execute a query
result = cur.execute("SELECT postcode, streetname, house_number FROM address")

# Do something with the resulting rows
for row in result:
    # We can now access field values by their name
    # Output each address as "<house_number> <streetname>, <postcode>"
    print(row["house_number"] + " " + row["streetname"] + ", " + row["postcode"])

# Close the database connection
con.close()
```

Code Snippet 2: Changing an attribute of `con` to return resulting rows as an `sqlite.Row` rather than a `tuple`

The `row_factory` attribute of the database connection is a function responsible for the format in which cursors return rows from a query. As we've seen in snippet 1, by default the `row_factory` manufactures a `tuple` for each row. However, as mentioned the `sqlite3` package includes an optimised `Row` type[2] which can be assigned to the connection's `row_factory` as demonstrated.

Note that this must be done before you request the cursor from the connection, else the cursor will have the default `row_factory` given to it by the connection. Finally, as you can see, we are now able to access fields with dictionary-like syntax.

Don't forget to close your connections!

[2]Hopefully you will remember from your experiments with importing and using functions from your weather package, that the `Row` type of the `sqlite3` package is accessed via `sqlite3.Row` - assuming `sqlite3` has been imported of course.