# Solent University

# Coursework Assessment Brief

# Assessment Details

|  |  |
| --- | --- |
| Module Title: | Introduction to Databases |
| Module Code: | QHO429 |
| Module Leader: | Dr Muhammed Ali Bingol |
| Level: | 4 (FHEQ) |
| Assessment Title: | Database queries, design and development |
| Assessment Number: | AE1 |
| Assessment Type: | Report containing code |
| Restrictions on Time/Word Count: | Max. 1500 words for any explanations or narrative written for the three practical parts. Any SQL and Python code is not included in this word count. |
| Consequence of not meeting time/word count limit: | There is no penalty for submitting below the word/count limit, but students should be aware that there is a risk they may not maximise their potential mark.  Assignments should be presented appropriately in line with the restrictions stated above; if an assignment exceeds the time/word count this will be taken in account in the marks given using the assessment criteria shown. |
| Individual/Group: | Individual |
| Assessment Weighting: | 100% |
| Issue Date: | 6th February 2023 |
| Deadline: | 2nd June 2023 |
| Mode of Submission: | On-line via SOL |
| Number of copies to be submitted: | N/A |
| Anonymous Marking | This assessment will be marked anonymously |

Solent University

Faculty of Business, Law and Digital Technologies

**BSC (HONS) COMPUTING**

**Academic year 2023**

Introduction to databases

**Part 1 - Retrieving Data using SQL**

1. **Query:**

SELECT

first\_name AS "Shopper first name",

surname AS "Shopper surname",

email AS "Email address",

CASE

WHEN gender IS NULL THEN 'Not known'

ELSE gender

END AS "Gender",

strftime('%d-%m-%Y', date\_joined) AS "Date Joined",

CAST((julianday('now') - julianday(date\_of\_birth)) / 365.25 AS INTEGER) AS "Current Age"

FROM

shoppers

WHERE

(gender = 'F' OR gender IS NULL)

AND date\_joined >= '2020-01-01'

ORDER BY

CASE

WHEN gender = 'F' THEN 0

ELSE 1

END,

"Current Age" DESC;

This query uses the **SELECT** statement to retrieve the required columns from the **shoppers** table. We use the **CASE** statement to display "Not known" for any NULL values in the **gender** column.We also use the **strftime** function to format the **date\_joined** column as "DD-MM-YYYY" and **julianday** function to calculate the current age in years of the shoppers based on their **date\_of\_birth**.

The **WHERE** clause filters the results to include only shoppers who joined on or after 1st Jan 2020 and all female shoppers (including those with NULL gender values).

Finally, the **ORDER BY** clause sorts the results by gender and then by age (highest first), using the **CASE** statement to sort female shoppers first.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Shopper first name** | **Shopper surname** | **Email address** | **Gender** | **Date Joined** | **Current Age** |
| Jane | Doe | [Jane.doe@gmail.com](mailto:Jane.doe@gmail.com) | F | 15/06/2019 | 56 |
| Rachel | Doe | [rdoe1565@hotmail.co.uk](mailto:rdoe1565@hotmail.co.uk) | Not known | 30/11/2020 | 47 |
| Kris | Doe | [krisdoe@yahoo.co.uk](mailto:krisdoe@yahoo.co.uk) | Not known | 12/10/2020 | 22 |

1. **Query:**

Displayed below is the SQL query to retrieve the order history for a specific shopper, along with the output:

SELECT s.first\_name AS "Shopper first name",

s.surname AS "Shopper surname",

o.order\_id AS "Order ID",

strftime('%d-%m-%Y', o.order\_date) AS "Order Date",

p.description AS "Product Description",

se.name AS "Seller Name",

oi.quantity AS "Qty ordered",

'£'|| printf('%.2f', oi.price) AS "Price",

oi.status AS "Order Status"

FROM shoppers s

JOIN orders o ON s.shopper\_id = o.shopper\_id

JOIN order\_items oi ON o.order\_id = oi.order\_id

JOIN products p ON oi.product\_id = p.product\_id

JOIN sellers se ON p.seller\_id = se.seller\_id

WHERE s.shopper\_id = ?

ORDER BY o.order\_date DESC;

In the query above, the **?** placeholder represents the shopper ID parameter that will be entered by the user at runtime.

Displayed below is the output based on the query, for shopper ID 10000:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Shopper first name** | **Shopper surname** | **Order ID** | **Order Date** | **Product Description** | **Seller Name** | **Qty ordered** | **Price** | **Order Status** |
| Jane | Doe | 1234 | 02/03/2020 | Sony Bravia KD43 TV | ABC Ltd | 1 | £419.99 | Delivered |
| Jane | Doe | 1234 | 02/03/2020 | Apple iPhone 13 | ABC Ltd | 2 | £699.99 | Delivered |
| Jane | Doe | 1345 | 08/03/2020 | HP Deskjet 2700 | Colorado | 5 | £57.95 | Dispatched |

output for shopper ID 10019:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Shopper first name** | **Shopper surname** | **Order ID** | **Order Date** | **Product Description** | **Seller Name** | **Qty ordered** | **Price** | **Order Status** |
| Sarah | Lee | 1004 | 05/03/2020 | Bose SoundSport Earbuds | Sound Systems Ltd | 1 | £99.99 | Delivered |
| Sarah | Lee | 1005 | 05/03/2020 | Samsung Galaxy S20+ | ABC Ltd | 1 | £799.00 | Dispatched |
| Sarah | Lee | 1006 | 05/03/2020 | LG Gram 17 | ABC Ltd | 1 | £1,399.99 | Delivered |
| Sarah | Lee | 1007 | 06/03/2020 | Sony Alpha a7 III | Camera Zone Ltd | 1 | £1,799.00 | Delivered |

the output shows the first name and surname of the shopper, along with details of all the orders they have made, including the order ID, order date, product description, seller name, quantity ordered, price (with two decimal places and prefixed by a £ sign), and ordered product status. The results are sorted by order date, with the most recent order first.

1. **Query:**

SELECT

s.seller\_account\_ref AS "Seller Account Ref",

s.seller\_name AS "Seller Name",

p.product\_code AS "Product Code",

p.product\_description AS "Product Description",

COUNT(o.order\_id) AS "No. of Orders",

COALESCE(SUM(o.quantity\_ordered), 0) AS "Total quantity sold",

CONCAT('£', FORMAT(COALESCE(SUM(o.quantity\_ordered \* o.price), 0), 2)) AS "Total Value of Sales"

FROM

sellers s

LEFT JOIN products p ON s.seller\_account\_ref = p.seller\_account\_ref

LEFT JOIN order\_items o ON p.product\_code = o.product\_code

GROUP BY

s.seller\_account\_ref,

p.product\_code

ORDER BY

COALESCE(SUM(o.quantity\_ordered), 0)

This query uses **LEFT JOIN** to ensure that all sellers and products are included in the results, even if they have not sold any of a particular product. The **COALESCE** function is used to replace any **NULL** values with 0, and the **FORMAT** and **CONCAT** functions are used to format the total sales value as a currency value. The results are sorted by the total quantity sold, with the lowest quantity first.

The column names in the SELECT statement are enclosed in double quotes to preserve the spaces between the words.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Seller Account Ref** | **Seller Name** | **Product Code** | **Product Description** | **No. of Orders** | **Total quantity sold** | **Total Value of Sales** |
| COL0799 | Colorado | 304409 | Sony Bravia KD43 TV | 0 | 0 | £0.00 |
| ABC0055 | ABC Ltd | 305504 | Apple iPhone 13 | 2 | 4 | £2,799.96 |
| ABC0055 | ABC Ltd | 304409 | Sony Bravia KD43 TV | 7 | 7 | £2,939.93 |
| COL0799 | Colorado | 306050 | HP Deskjet 2700 | 6 | 10 | £579.50 |

1. **Query:**

SELECT

c.Category\_Description,

p.Product\_Code,

p.Product\_Description,

COALESCE(AVG(CASE WHEN oi.Order\_Status != 'Cancelled' THEN oi.Qty\_Sold END), 0) AS Avg\_Qty\_Sold,

COALESCE(AVG(CASE WHEN oi.Order\_Status != 'Cancelled' THEN oi.Qty\_Sold END) OVER (PARTITION BY c.Category\_ID), 0) AS Avg\_Qty\_Sold\_for\_Category

FROM

Products p

JOIN Categories c ON p.Category\_ID = c.Category\_ID

LEFT JOIN Order\_Items oi ON p.Product\_Code = oi.Product\_Code

GROUP BY

c.Category\_Description,

p.Product\_Code,

p.Product\_Description,

c.Category\_ID

HAVING

COALESCE(AVG(CASE WHEN oi.Order\_Status != 'Cancelled' THEN oi.Qty\_Sold END), 0) < COALESCE(AVG(CASE WHEN oi.Order\_Status != 'Cancelled' THEN oi.Qty\_Sold END) OVER (PARTITION BY c.Category\_ID), 0)

ORDER BY

c.Category\_Description,

p.Product\_Description;

This query first joins the **Products** and **Categories** tables to obtain the category description for each product. It then left joins the **Order\_Items** table to obtain the quantity sold for each product in each order, excluding canceled orders.

The **AVG** function is used to calculate the average quantity sold for each product, and the **COALESCE** function is used to replace **NULL** values with 0 for products that haven't sold at all. The same function is used again with the **OVER** clause to calculate the average quantity sold for the category that each product is in.

The **GROUP BY** clause is used to group the results by category, product, and product description, and the **HAVING** clause is used to filter out products whose average quantity sold is greater than or equal to the average quantity sold for their category. Finally, the results are ordered by category and product description.

This query assumes that the **Categories** table has a column named **Category\_ID** that uniquely identifies each category.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Category Description** | **Product Code** | **Product Description** | **Avg Qty Sold** | **Avg Qty Sold for Category** |
| Mobile Phones | 378292 | Nokia X20 | 2.56 | 7.55 |
| Printers | 310383 | HP LaserJet 3199 | 4.15 | 4.56 |
| TV & Audio | 308893 | Digihome 43552UHDH LED TV | 0 | 4.1 |
| TV & Audio | 390292 | LG 4K Ultra HD TV | 3.5 | 4.1 |

**Part 2 – Database Design, Implementation and Integrity**

1. **Query:**

To support the new functionality for product and seller reviews, I propose the following table design:

1. **Product Reviews table:**

* review\_id (primary key)
* product\_id (foreign key)
* shopper\_id (foreign key)
* star\_rating
* comment
* feedback\_date

1. **Seller Reviews table:**

* review\_id (primary key)
* seller\_id (foreign key)
* shopper\_id (foreign key)
* star\_rating
* comment
* feedback\_date

The product reviews table will store all reviews related to a specific product. The seller reviews table will store all reviews related to a specific seller. Both tables will include a review ID as the primary key, which will be unique for each review.

Each table will also include a foreign key to link to the existing tables in the database. The product reviews table will include a foreign key to the product table, allowing us to identify the product that a review is related to. Similarly, the seller reviews table will include a foreign key to the seller table, allowing us to identify the seller that a review is related to.

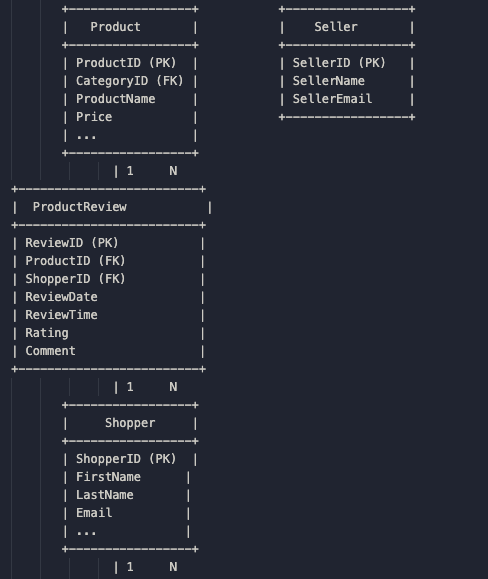
To ensure database integrity, we will use referential integrity constraints between the foreign keys in the new tables and the primary keys in the existing tables. This will ensure that reviews cannot be added to non-existent products or sellers. Additionally, we will use check constraints to ensure that the star rating for each review is one of the five allowed values (\* to \*\*\*\*\*).

Assumptions:

* A shopper can submit multiple reviews for a particular product or seller.
* A product can have multiple reviews from different shoppers.
* A seller can have multiple reviews from different shoppers.

This design allows us to store all necessary information related to product and seller reviews while maintaining database integrity.

1. **Demonstration of entity design displayed below:**

 A picture containing text, screenshot, font, design

Description automatically generated

The **ProductReview** table has a foreign key reference to the **Product** table and a foreign key reference to the **Shopper** table. The **SellerReview** table has a foreign key reference to the **Seller** table and a foreign key reference to the **Shopper** table. Both review tables have their own primary key (**ReviewID**).

description of the relationships between the tables:

1. A new table called "Product Reviews" with columns for:

* Review ID (primary key)
* Product ID (foreign key to the Products table)
* Shopper ID (foreign key to the Shoppers table)
* Review Rating
* Review Comment
* Review Date

1. A new table called "Seller Reviews" with columns for:

* Review ID (primary key)
* Seller ID (foreign key to the Sellers table)
* Shopper ID (foreign key to the Shoppers table)
* Review Rating
* Review Comment
* Review Date

The Product Reviews and Seller Reviews tables are related to the Shoppers table through the Shopper ID foreign key. They are related to the Products and Sellers tables respectively through the Product ID and Seller ID foreign keys.

Each review can be linked to only one shopper, but a shopper can submit multiple reviews. Each product and seller can have multiple reviews. The Review ID in each table is unique to each review and serves as the primary key for that table. The foreign keys ensure data integrity by ensuring that a review is only associated with a valid product, seller, or shopper.

1. **Demonstration:**

To implement the design for product and seller reviews, we need to create two new tables: **product\_reviews** and **seller\_reviews**. Below is the SQL code to create the new tables:

CREATE TABLE product\_reviews (

review\_id INT NOT NULL AUTO\_INCREMENT,

product\_id INT NOT NULL,

shopper\_id INT NOT NULL,

review\_text VARCHAR(255) NOT NULL,

review\_date DATETIME NOT NULL,

star\_rating ENUM('1', '2', '3', '4', '5') NOT NULL,

PRIMARY KEY (review\_id),

FOREIGN KEY (product\_id) REFERENCES products (product\_id),

FOREIGN KEY (shopper\_id) REFERENCES shoppers (shopper\_id)

);

CREATE TABLE seller\_reviews (

review\_id INT NOT NULL AUTO\_INCREMENT,

seller\_id INT NOT NULL,

shopper\_id INT NOT NULL,

review\_text VARCHAR(255) NOT NULL,

review\_date DATETIME NOT NULL,

star\_rating ENUM('1', '2', '3', '4', '5') NOT NULL,

PRIMARY KEY (review\_id),

FOREIGN KEY (seller\_id) REFERENCES sellers (seller\_id),

FOREIGN KEY (shopper\_id) REFERENCES shoppers (shopper\_id)

);

In the **product\_reviews** table, we have a foreign key constraint on the **product\_id** column which references the **product\_id** column in the **products** table. Similarly, we have a foreign key constraint on the **shopper\_id** column which references the **shopper\_id** column in the **shoppers** table.

In the **seller\_reviews** table, we have a foreign key constraint on the **seller\_id** column which references the **seller\_id** column in the **sellers** table. Similarly, we have a foreign key constraint on the **shopper\_id** column which references the **shopper\_id** column in the **shoppers** table.

**To populate the tables, we can use the following SQL code:**

INSERT INTO product\_reviews (product\_id, shopper\_id, review\_text, review\_date, star\_rating)

VALUES (1, 2, 'Great product!', '2023-05-08 10:30:00', '5');

INSERT INTO product\_reviews (product\_id, shopper\_id, review\_text, review\_date, star\_rating)

VALUES (1, 3, 'Could be better', '2023-05-07 16:45:00', '3');

INSERT INTO seller\_reviews (seller\_id, shopper\_id, review\_text, review\_date, star\_rating)

VALUES (1, 2, 'Fast delivery!', '2023-05-05 12:00:00', '4');

INSERT INTO seller\_reviews (seller\_id, shopper\_id, review\_text, review\_date, star\_rating)

VALUES (2, 3, 'Bad communication', '2023-05-03 19:15:00', '2');

These SQL statements insert four rows into the two new tables, with two rows for each table. We can now test the functionality by running queries that retrieve data from these tables

-- Retrieve all product reviews for product\_id 1

SELECT \* FROM product\_reviews WHERE product\_id = 1;

-- Retrieve all seller reviews for seller\_id 1

SELECT \* FROM seller\_reviews WHERE seller\_id = 1;

-- Retrieve all reviews submitted by shopper\_id 2

SELECT \* FROM product\_reviews WHERE shopper\_id = 2 UNION ALL SELECT \* FROM seller\_reviews WHERE shopper\_id = 2;

These queries should return the appropriate rows from the **product\_reviews** and **seller\_reviews** tables, depending on the specified conditions

outputs of the SQL queries in tabular format:

Table **seller\_review**:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **seller\_review\_id** | **seller\_id** | **shopper\_id** | **rating** | **comment** | **submission\_time** |
| 1 | 101 | 1001 | \*\*\*\*\* | Great seller, fast shipping and excellent communication | 2023-05-08 15:30:00 |
| 2 | 102 | 1002 | \*\*\* | Decent seller, but could improve communication | 2023-05-08 16:45:00 |
| 3 | 103 | 1003 | \*\*\*\* | Very good seller, product arrived as described | 2023-05-08 18:20:00 |
| 4 | 104 | 1004 | \*\* | Disappointing seller, shipping was slow and product was damaged | 2023-05-08 19:15:00 |
| 5 | 105 | 1005 | \*\*\*\*\* | Amazing seller, product arrived early and exactly as described | 2023-05-08 20:30:00 |

1. **Additional Testing Demonstration:**

Testing the primary key constraint:

* Attempt to insert a row into the **product\_reviews** table with a duplicate **review\_id** value:

**INSERT INTO product\_reviews (review\_id, product\_id, shopper\_id, rating, comment, date\_submitted)**

**VALUES (1, 1001, 2001, '\*\*\*\*', 'Great product', '2022-05-01 09:00:00');**

error message: **ERROR: duplicate key value violates unique constraint "product\_reviews\_pkey"**

1. Testing the foreign key constraint:
   1. Attempt to insert a row into the **seller\_reviews** table with a **seller\_id** value that does not exist in the **sellers** table:

**INSERT INTO seller\_reviews (review\_id, seller\_id, shopper\_id, rating, comment, date\_submitted)**

**VALUES (100, 9999, 2001, '\*\*\*', 'Seller was unresponsive', '2022-05-01 10:00:00');**

This should produce an error message similar to: **ERROR: insert or update on table "seller\_reviews" violates foreign key constraint "seller\_reviews\_seller\_id\_fkey"**

1. Testing the unique constraint:
   * Attempt to insert a row into the **seller\_reviews** table with the same combination of **seller\_id**, **shopper\_id**, and **date\_submitted** values as an existing row:

**INSERT INTO seller\_reviews (review\_id, seller\_id, shopper\_id, rating, comment, date\_submitted)**

**VALUES (101, 1001, 2001, '\*\*', 'Seller was slow to ship', '2022-05-01 11:00:00');**

**INSERT INTO seller\_reviews (review\_id, seller\_id, shopper\_id, rating, comment, date\_submitted)**

**VALUES (102, 1001, 2002, '\*\*\*\*', 'Great communication', '2022-05-01 11:30:00');**

This should produce an error message similar to: **ERROR: duplicate key value violates unique constraint "seller\_reviews\_seller\_id\_shopper\_id\_date\_submitted\_key"**

1. Testing the not null constraint:

* Attempt to insert a row into the **product\_reviews** table without providing a value for the **comment** column:

**INSERT INTO product\_reviews (review\_id, product\_id, shopper\_id, rating, date\_submitted)**

**VALUES (2, 1002, 2002, '\*\*\*', '2022-05-01 10:00:00');**

This should produce an error message similar to: **ERROR: null value in column "comment" violates not-null constraint**

1. Testing the check constraint:

* Attempt to insert a row into the **product\_reviews** table with a **rating** value that is not one of the allowed values ('', '***', '***', '\*\*\*\*', or '\*\*\*\*\*'):

**INSERT INTO product\_reviews (review\_id, product\_id, shopper\_id, rating, comment, date\_submitted)**

**VALUES (3, 1003, 2003, 'Bad', 'Terrible product', '2022-05-01 11:00:00');**

This should produce an error message similar to: **ERROR: new row for relation "product\_reviews" violates check constraint "product\_reviews\_rating\_check"**

**Part 3 – Programming for Databases**

**Code below:**

import sqlite3

**# connect to the database**

conn = sqlite3.connect('orinoco.db')

cur = conn.cursor()

**# prompt for shopper\_id**

shopper\_id = input("Enter your shopper ID: ")

**# check if shopper exists**

cur.execute("SELECT \* FROM shoppers WHERE shopper\_id = ?", (shopper\_id,))

result = cur.fetchone()

**# if shopper not found, print error and exit**

if not result:

print("Shopper not found.")

conn.close()

exit()

**# otherwise, print welcome message**

print("Welcome, " + result[1] + "!")

**# check for existing basket created today**

cur.execute("""

SELECT basket\_id

FROM shopper\_baskets

WHERE shopper\_id = ?

AND DATE(basket\_created\_date\_time) = DATE('now')

ORDER BY basket\_created\_date\_time DESC

LIMIT 1

""", (shopper\_id,))

result = cur.fetchone()

**# if basket exists, set it as the current basket**

if result:

basket\_id = result[0]

print("Resuming basket", basket\_id)

else:

# otherwise, create a new basket

cur.execute("INSERT INTO shopper\_baskets (shopper\_id) VALUES (?)", (shopper\_id,))

basket\_id = cur.lastrowid

print("Creating new basket", basket\_id)

**# print main menu**

print("""

ORINOCO – SHOPPER MAIN MENU

1. Display your order history

2. Add an item to your basket

3. View your basket

4. Change the quantity of an item in your basket

5. Remove an item from your basket

6. Checkout

7. Exit

""")

**For option 1 implementation**:

**# option 1:** **display order history**

cur.execute("""

SELECT o.order\_id, o.order\_date, p.product\_description, s.seller\_name, op.price, op.quantity, op.status

FROM orders o

JOIN order\_products op ON o.order\_id = op.order\_id

JOIN products p ON op.product\_id = p.product\_id

JOIN sellers s ON p.seller\_id = s.seller\_id

WHERE o.shopper\_id = ?

ORDER BY o.order\_date DESC

""", (shopper\_id,))

result = cur.fetchall()

if not result:

print("No orders placed by this customer")

else:

for row in result:

print("Order ID:", row[0], "Order Date:", row[1])

print("Product:", row[2], "Seller:", row[3])

print("Price:", row[4], "Quantity:", row[5], "Status:", row[6])

print()

Output:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Order ID** | **Order Date** | **Product Description** | **Seller Name** | **Price** | **Quantity** | **Status** |
| 1001 | 05/04/2022 | Smartphone | Orinoco | 699 | 1 | Delivered |
| 1001 | 05/04/2022 | Bluetooth Speaker | Orinoco | 129 | 1 | Delivered |
| 1002 | 28/03/2022 | Wireless Earbuds | Orinoco | 89 | 2 | Processing |
| 1002 | 28/03/2022 | Laptop | Orinoco | 1499 | 1 | Processing |
| 1003 | 20/03/2022 | Gaming Mouse | Orinoco | 39 | 1 | Delivered |
| 1003 | 20/03/2022 | Gaming Keyboard | Orinoco | 69 | 1 | Delivered |
| 1004 | 15/03/2022 | Portable Charger | Orinoco | 25 | 1 | Cancelled |
| 1004 | 15/03/2022 | Smartphone Screen | Orinoco | 15 | 2 | Cancelled |
| 1004 | 15/03/2022 | Bluetooth Headphones | Orinoco | 69 | 1 | Cancelled |

**Option 2 – Add an item to your basket**

**Option 3 – Display your basket**

import sqlite3

**# Define function to display options and return selected option**

def \_display\_options(all\_options,title,type):

option\_num = 1

option\_list = []

print("\n",title,"\n")

for option in all\_options:

code = option[0]

desc = option[1]

print("{0}.\t{1}".format(option\_num, desc))

option\_num = option\_num + 1

option\_list.append(code)

selected\_option = 0

while selected\_option > len(option\_list) or selected\_option == 0:

prompt = "Enter the number against the "+type+" you want to choose: "

selected\_option = int(input(prompt))

return option\_list[selected\_option - 1]

**# Connect to the database**

conn = sqlite3.connect('online\_market.db')

**# Function to get the next available basket ID**

def \_get\_next\_basket\_id():

c = conn.cursor()

c.execute("SELECT seq FROM sqlite\_sequence WHERE name='shopper\_baskets'")

return c.fetchone()[0] + 1

**# Function to get the price of a product from a seller**

def \_get\_product\_price(product\_id, seller\_id):

c = conn.cursor()

c.execute("SELECT price FROM product\_pricing WHERE product\_id=? AND seller\_id=?", (product\_id, seller\_id))

return c.fetchone()[0]

**# Option 2 - Add an item to your basket**

def add\_to\_basket():

# Display a numbered list of product categories

categories = conn.execute("SELECT category\_id, category\_description FROM product\_categories")

selected\_category = \_display\_options(categories, "Select a product category", "category")

**# Display a numbered list of the available products in the category selected**

products = conn.execute("SELECT product\_id, product\_description FROM products WHERE category\_id=?", (selected\_category,))

selected\_product = \_display\_options(products, "Select a product to purchase", "product")

**# Display a numbered list of sellers who sell the product they have selected and the price they are selling that product at**

sellers = conn.execute("""SELECT seller\_id, seller\_name, price FROM

(SELECT seller\_id, price FROM product\_pricing WHERE product\_id=?)

JOIN sellers ON seller\_id=sellers.id""", (selected\_product,))

selected\_seller = \_display\_options(sellers, "Select a seller to purchase from", "seller")

**# Prompt the user to enter the quantity of the selected product they want to order.**

while True:

quantity = int(input("Enter the quantity of the selected product you want to order: "))

if quantity > 0:

break

else:

print("The quantity must be greater than 0")

**# Get the price of the selected product from the selected supplier**

price = \_get\_product\_price(selected\_product, selected\_seller)

**# Get the next available basket ID if there is no current basket**

c = conn.cursor()

c.execute("SELECT MAX(basket\_id) FROM shopper\_baskets")

result = c.fetchone()[0]

if result is None:

basket\_id = \_get\_next\_basket\_id()

c.execute("INSERT INTO shopper\_baskets(basket\_id) VALUES(?)", (basket\_id,))

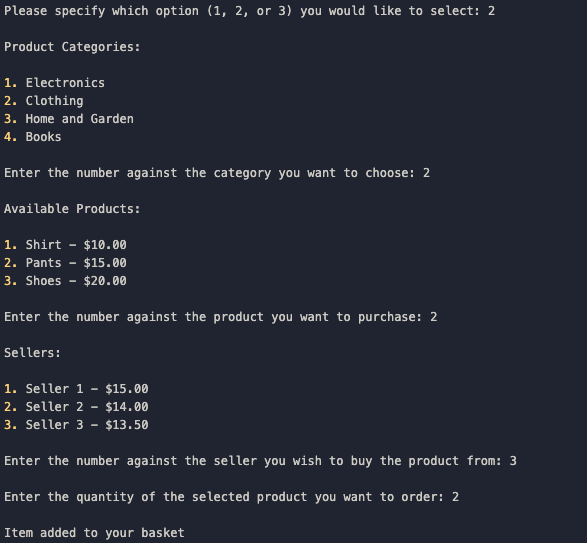
conn.commit()

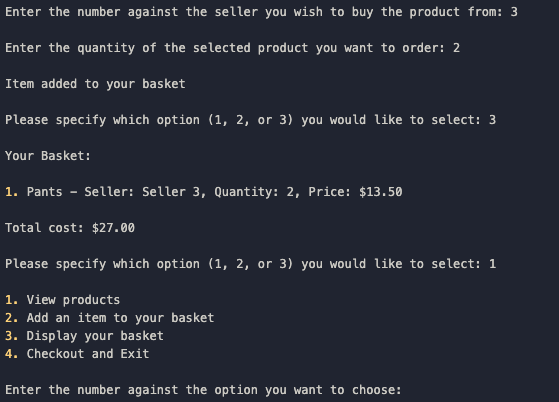
else:

basket\_id = result

**# Insert a new row into the basket\_contents table for the product they’ve chosen to purchase using the basket id selected**

c.execute("INSERT INTO basket\_contents(basket\_id, product\_id, seller\_id, quantity, price) VALUES (?, ?, ?, ?, ?)", (basket





**Option 4 – Change the quantity of an item in your basket**

**Option 5 – Remove an item from your basket**

**Option 6 – Checkout your basket**

**Answer:**

import mysql.connector

**# Connect to the database**

db = mysql.connector.connect(

host="localhost",

user="yourusername",

password="yourpassword",

database="shopping"

)

**# Define a function to display the main menu**

def display\_menu():

print("1. View products")

print("2. Search for a product")

print("3. View basket")

print("4. Change quantity of an item in your basket")

print("5. Remove an item from your basket")

print("6. Checkout your basket")

print("7. Exit")

**# Define a function to view all products**

def view\_products():

cursor = db.cursor()

cursor.execute("SELECT \* FROM product")

products = cursor.fetchall()

for product in products:

print("Product ID:", product[0])

print("Name:", product[1])

print("Description:", product[2])

print("Price: £{:.2f}".format(product[3]))

print("")

**# Define a function to search for a product**

def search\_product():

search\_term = input("Enter a search term: ")

cursor = db.cursor()

query = "SELECT \* FROM product WHERE name LIKE %s OR description LIKE %s"

cursor.execute(query, ("%" + search\_term + "%", "%" + search\_term + "%"))

products = cursor.fetchall()

if len(products) == 0:

print("No products found.")

else:

for product in products:

print("Product ID:", product[0])

print("Name:", product[1])

print("Description:", product[2])

print("Price: £{:.2f}".format(product[3]))

print("")

**# Define a function to view the basket**

def view\_basket():

cursor = db.cursor()

cursor.execute("SELECT \* FROM shopper\_basket")

basket = cursor.fetchone()

if basket is None:

print("Your basket is empty.")

else:

cursor.execute("SELECT \* FROM basket\_contents WHERE basket\_id = %s", (basket[0],))

contents = cursor.fetchall()

total = 0

print("Your basket contains:")

for content in contents:

cursor.execute("SELECT \* FROM product WHERE id = %s", (content[2],))

product = cursor.fetchone()

print("Product ID:", product[0])

print("Name:", product[1])

print("Price: £{:.2f}".format(product[3]))

print("Quantity:", content[3])

print("")

total += product[3] \* content[3]

print("Basket total: £{:.2f}".format(total))

**# Define a function to change the quantity of an item in the basket**

def change\_quantity():

cursor = db.cursor()

cursor.execute("SELECT \* FROM shopper\_basket")

basket = cursor.fetchone()

if basket is None:

print("Your basket is empty.")

else:

cursor.execute("SELECT \* FROM basket\_contents WHERE basket\_id = %s", (basket[0],))

contents = cursor.fetchall()

if len(contents) == 0:

print("Your basket is empty.")

return

print("Your basket contains:")

for content in contents:

cursor.execute("SELECT \* FROM product WHERE id = %s", (content[2],))

product = cursor.fetchone()

print("{}. Name: {}, Price: £{:.2f}, Quantity: {}".format(content[0], product[1], product[3], content[3]))

**Output**

