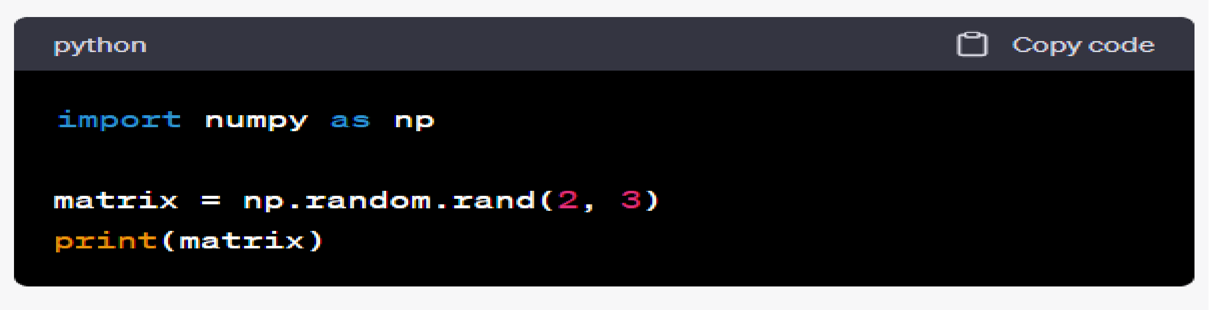
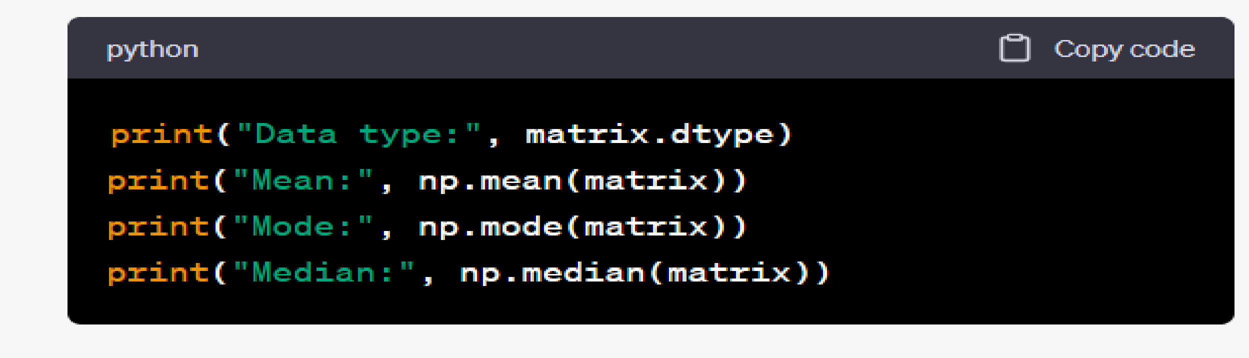
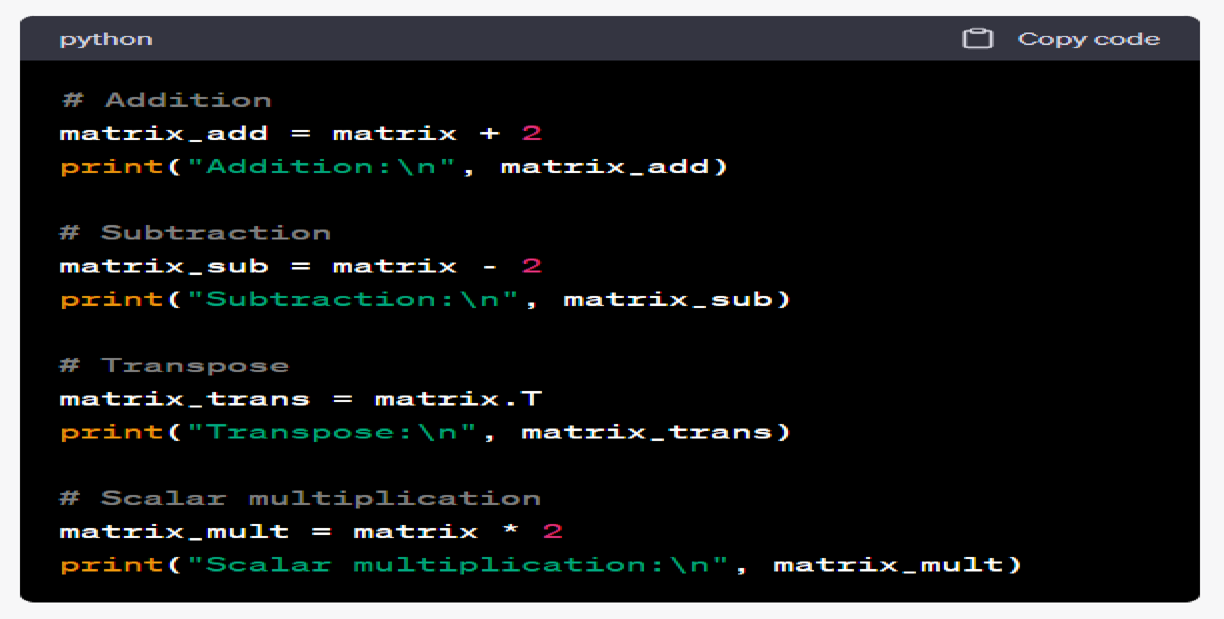
1. **<u>Data analysis and matrix operations</u>**



```python
import numpy as np

matrix = np.random.rand(2, 3)
print(matrix)
```

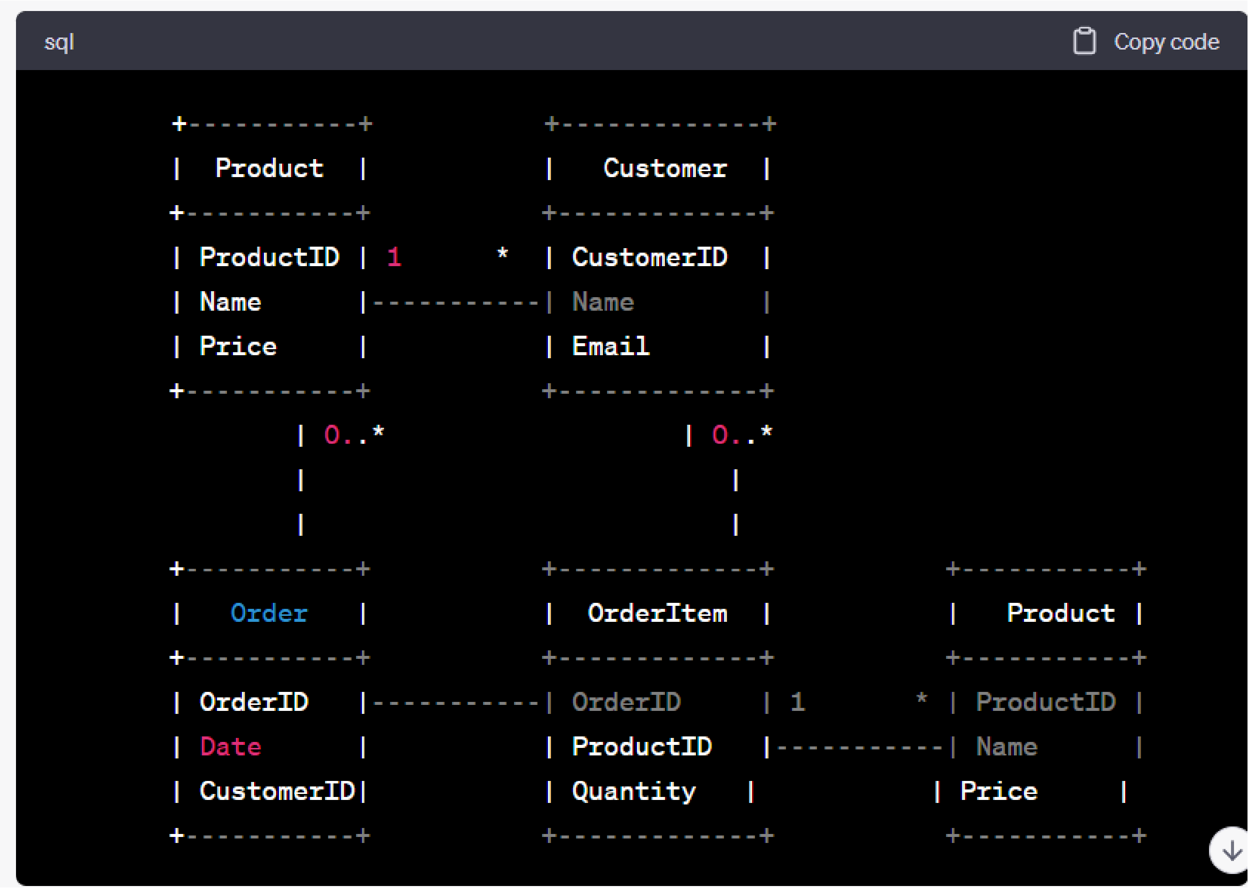


```python
 print("Data type:", matrix.dtype)
print("Mean:", np.mean(matrix))
print("Mode:", np.mode(matrix))
print("Median:", np.median(matrix))
```



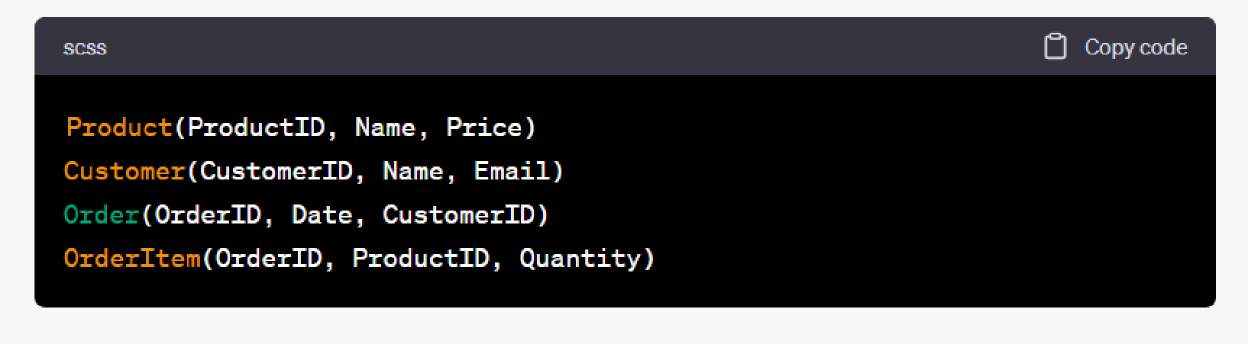
```python
# Addition
matrix_add = matrix + 2
print("Addition:\n", matrix_add)

# Subtraction
matrix_sub = matrix - 2
print("Subtraction:\n", matrix_sub)

# Transpose
matrix_trans = matrix.T
print("Transpose:\n", matrix_trans)

# Scalar multiplication
matrix_mult = matrix * 2
print("Scalar multiplication:\n", matrix_mult)
```

2. Checkpoint conceptual model



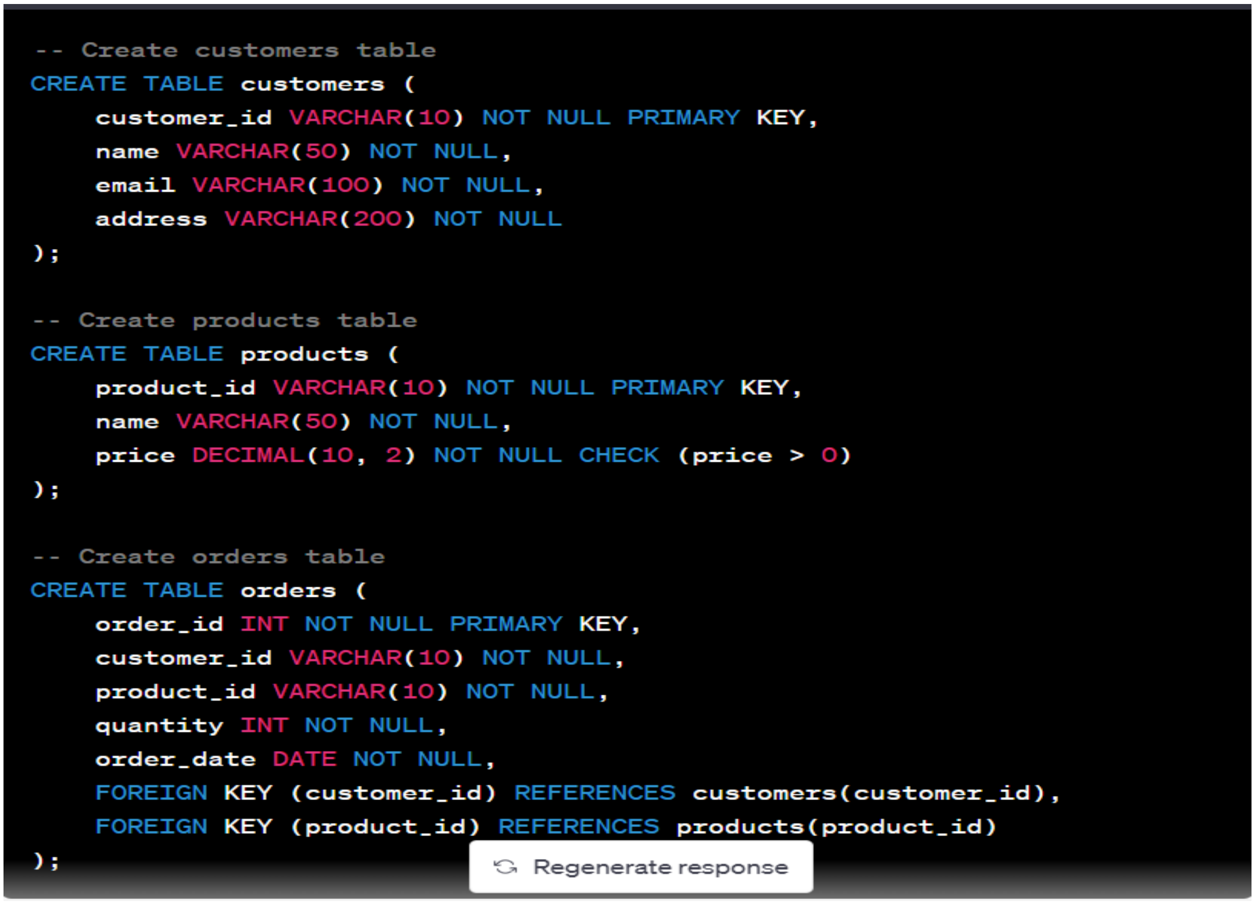
```
        +-----------+           +-------------+
        |  Product  |           |   Customer  |
        +-----------+           +-------------+
        | ProductID | 1      *  | CustomerID  |
        | Name      |-----------| Name        |
        | Price     |           | Email       |
        +-----------+           +-------------+
                | 0..*                    | 0..*
                |                            |
                |                            |
        +-----------+           +-------------+           +-----------+
        |   Order   |           |  OrderItem  |           |   Product |
        +-----------+           +-------------+           +-----------+
        | OrderID   |-----------| OrderID     | 1       * | ProductID |
        | Date      |           | ProductID   |-----------| Name      |
        | CustomerID|           | Quantity   |           | Price     |
        +-----------+           +-------------+           +-----------+
```

3. Checkpoint relational model



```
Product(ProductID, Name, Price)
Customer(CustomerID, Name, Email)
Order(OrderID, Date, CustomerID)
OrderItem(OrderID, ProductID, Quantity)
```

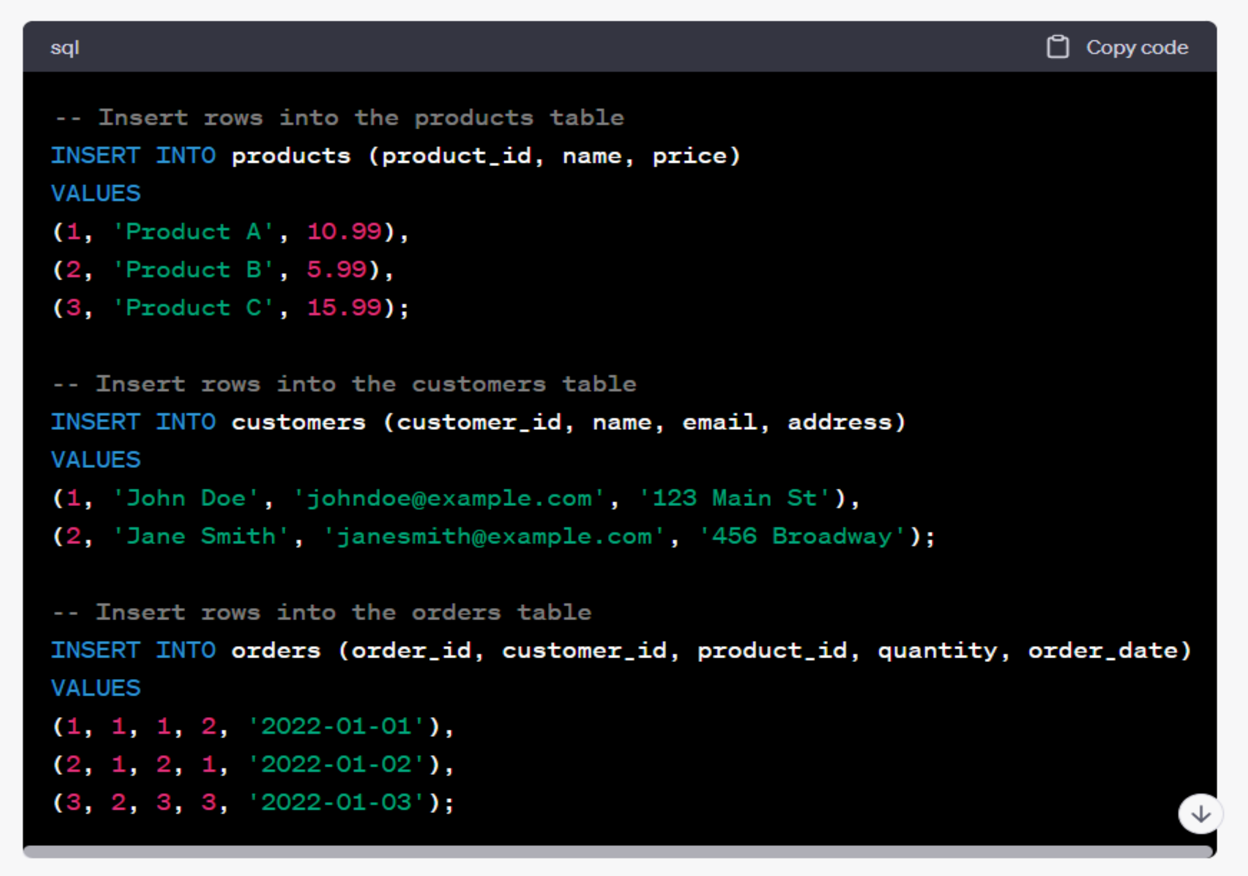
4. Data definition language:



```sql
-- Create customers table
CREATE TABLE customers (
    customer_id VARCHAR(10) NOT NULL PRIMARY KEY,
    name VARCHAR(50) NOT NULL,
    email VARCHAR(100) NOT NULL,
    address VARCHAR(200) NOT NULL
);

-- Create products table
CREATE TABLE products (
    product_id VARCHAR(10) NOT NULL PRIMARY KEY,
    name VARCHAR(50) NOT NULL,
    price DECIMAL(10, 2) NOT NULL CHECK (price > 0)
);

-- Create orders table
CREATE TABLE orders (
    order_id INT NOT NULL PRIMARY KEY,
    customer_id VARCHAR(10) NOT NULL,
    product_id VARCHAR(10) NOT NULL,
    quantity INT NOT NULL,
    order_date DATE NOT NULL,
    FOREIGN KEY (customer_id) REFERENCES customers(customer_id),
    FOREIGN KEY (product_id) REFERENCES products(product_id)
);
```

```
CREATE TABLE customers ( customer_id INT NOT NULL PRIMARY KEY,  name VARCHAR(50) NOT
NULL,   email VARCHAR(100) NOT NULL, address VARCHAR(200) NOT NULL );

CREATE TABLE products (
   product_id INT NOT NULL PRIMARY KEY,
   name VARCHAR(50) NOT NULL,
   price DECIMAL(10,2) NOT NULL CHECK (price > 0)
);

CREATE TABLE orders (
   order_id INT NOT NULL PRIMARY KEY,
   customer_id INT NOT NULL,
   product_id INT NOT NULL,
   quantity INT NOT NULL,
   order_date DATE NOT NULL,
   FOREIGN KEY (customer_id) REFERENCES customers(customer_id),
   FOREIGN KEY (product_id) REFERENCES products(product_id)
);
```

5. **Data manipulation language:**



```sql
-- Insert rows into the products table
INSERT INTO products (product_id, name, price)
VALUES
(1, 'Product A', 10.99),
(2, 'Product B', 5.99),
(3, 'Product C', 15.99);

-- Insert rows into the customers table
INSERT INTO customers (customer_id, name, email, address)
VALUES
(1, 'John Doe', 'johndoe@example.com', '123 Main St'),
(2, 'Jane Smith', 'janesmith@example.com', '456 Broadway');

-- Insert rows into the orders table
INSERT INTO orders (order_id, customer_id, product_id, quantity, order_date)
VALUES
(1, 1, 1, 2, '2022-01-01'),
(2, 1, 2, 1, '2022-01-02'),
(3, 2, 3, 3, '2022-01-03');
```

```
-- Insert rows into the products table
INSERT INTO products (product_id, name, price)
VALUES
(1, 'Product A', 10.99),
(2, 'Product B', 5.99),
(3, 'Product C', 15.99);

-- Insert rows into the customers table
INSERT INTO customers (customer_id, name, email, address)
VALUES
(1, 'John Doe', 'johndoe@example.com', '123 Main St'),
(2, 'Jane Smith', 'janesmith@example.com', '456 Broadway');

-- Insert rows into the orders table
INSERT INTO orders (order_id, customer_id, product_id, quantity, order_date)
VALUES
(1, 1, 1, 2, '2022-01-01'),
(2, 1, 2, 1, '2022-01-02'),
(3, 2, 3, 3, '2022-01-03');
```

6. Data manipulation language:

Customer (customer_id, customer_name, customer_tel)
Product (product_id, product_name, category, price)
orders(#customer_id, #product_id,orderDate, quantity,total_amount)

1. **Write a SQL query to retrieve the names of the customers who have placed an order for at least one widget and at least one gadget, along with the total cost of the widgets and gadgets ordered by each customer. The cost of each item should be calculated by multiplying the quantity by the price of the product.**

SELECT c.customer_name, SUM(o.quantity * p.price) as total_cost

FROM Customer c

JOIN Orders o ON c.customer_id = o.customer_id

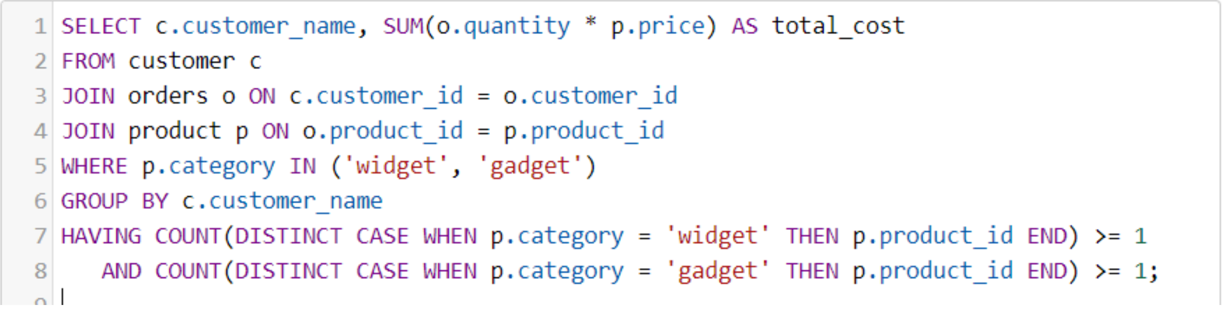
JOIN Product p ON o.product_id = p.product_id

WHERE p.category IN ('widget', 'gadget')

GROUP BY c.customer_id

HAVING COUNT(DISTINCT CASE WHEN p.category = 'widget' THEN p.product_id END) > 0

AND COUNT(DISTINCT CASE WHEN p.category = 'gadget' THEN p.product_id END) > 0



```
SELECT c.customer_name, SUM(o.quantity * p.price) AS total_cost
FROM customer c
JOIN orders o ON c.customer_id = o.customer_id
JOIN product p ON o.product_id = p.product_id
WHERE p.category IN ('widget', 'gadget')
GROUP BY c.customer_name
HAVING COUNT(DISTINCT CASE WHEN p.category = 'widget' THEN p.product_id END) >= 1
   AND COUNT(DISTINCT CASE WHEN p.category = 'gadget' THEN p.product_id END) >= 1;
```

- The query selects the customer name and the total cost of the widgets and gadgets ordered by each customer.
- The join clause joins the customer, orders, and product tables based on their corresponding IDs.
- The WHERE clause filters the results to only include products with the category of 'widget' or 'gadget'.
- The GROUP BY clause groups the results by the customer name.
- The HAVING clause filters the results to only include customers who have ordered at least one widget and at least one gadget.

2. **Write a query to retrieve thAe names of the customers who have placed an order for at least one widget, along with the total cost of the widgets ordered by each customer.**

SELECT c.customer_name, SUM(o.quantity * p.price) AS total_cost

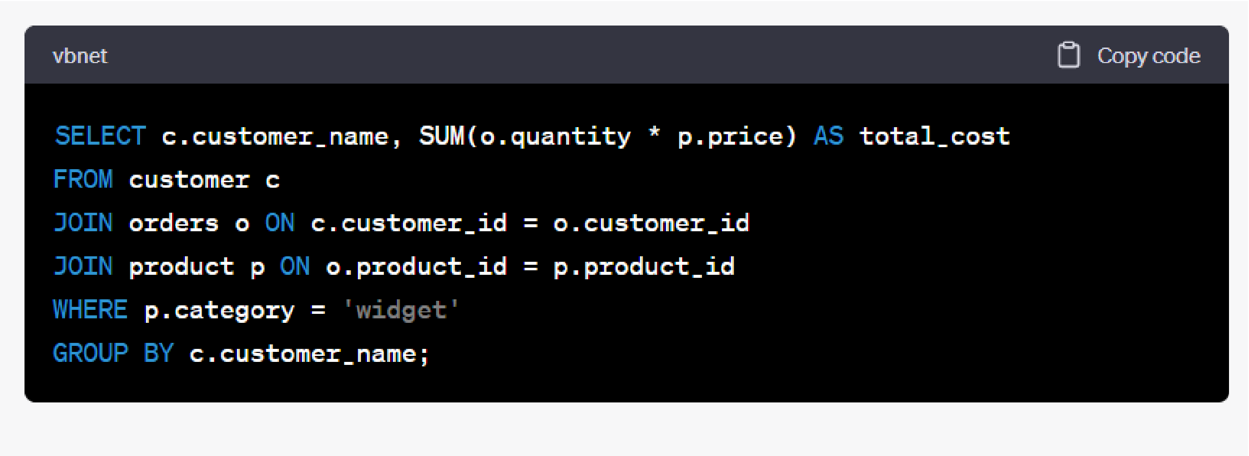
FROM customer c

JOIN orders o ON c.customer_id = o.customer_id

JOIN product p ON o.product_id = p.product_id

WHERE p.category = 'widget'

GROUP BY c.customer_name;



```
SELECT c.customer_name, SUM(o.quantity * p.price) AS total_cost
FROM customer c
JOIN orders o ON c.customer_id = o.customer_id
JOIN product p ON o.product_id = p.product_id
WHERE p.category = 'widget'
GROUP BY c.customer_name;
```

- The query selects the customer name and the total cost of the widgets ordered by each customer.
- The join clause joins the customer, orders, and product tables based on their corresponding IDs.
- The WHERE clause filters the results to only include products with the category of 'widget'.
- The GROUP BY clause groups the results by the customer name.

3. **Write a query to retrieve the names of the customers who have placed an order for at least one gadget, along with the total cost of the gadgets ordered by each customer.**

   SELECT c.customer_name, SUM(o.quantity * p.price) AS total_cost

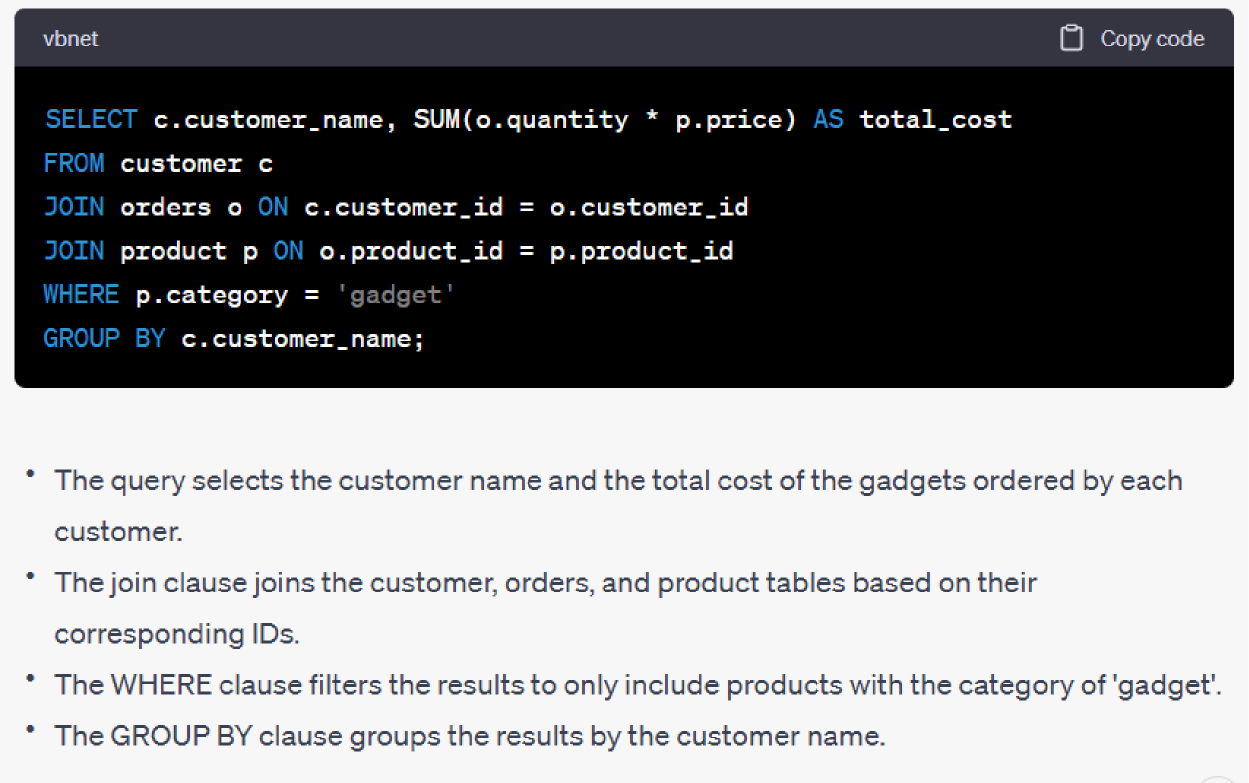
   FROM customer c

   JOIN orders o ON c.customer_id = o.customer_id

   JOIN product p ON o.product_id = p.product_id

   WHERE p.category = 'gadget'

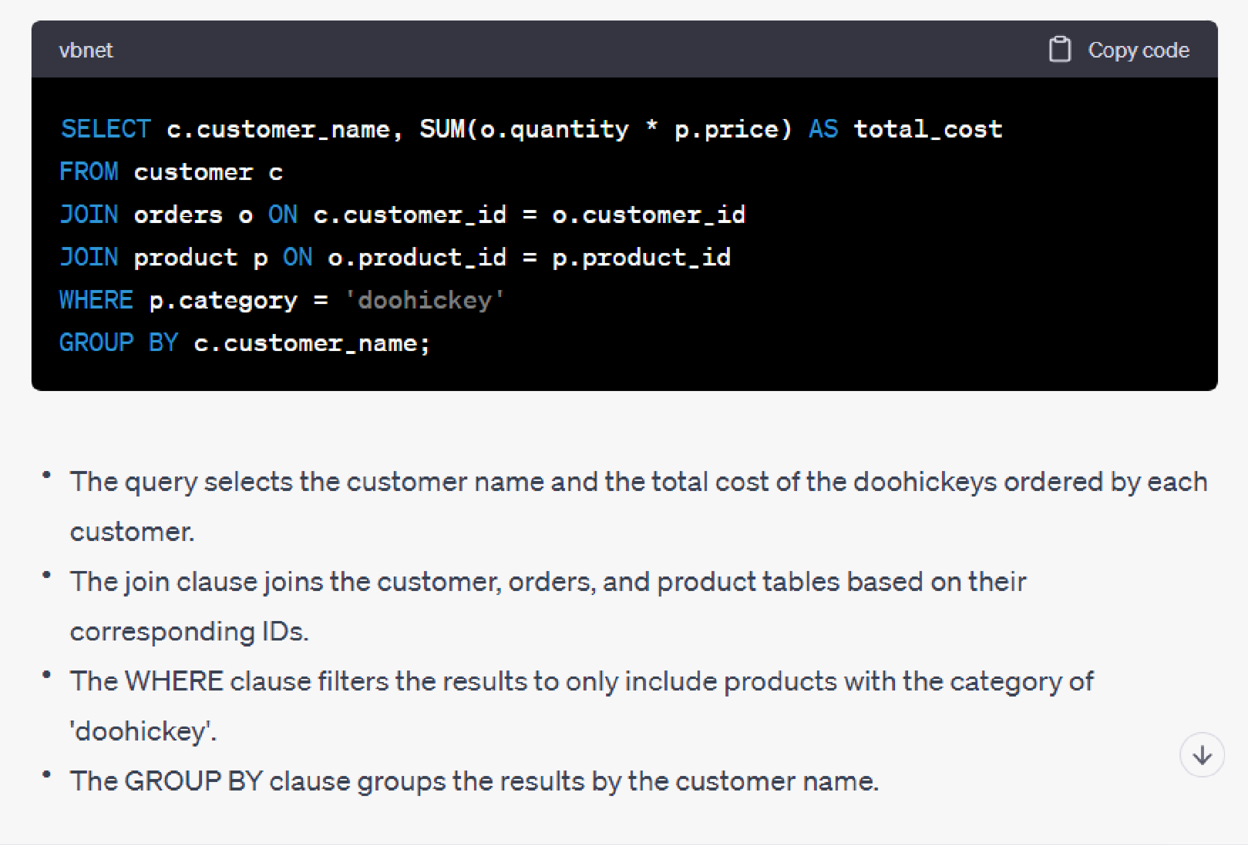
   GROUP BY c.customer_name;



```
SELECT c.customer_name, SUM(o.quantity * p.price) AS total_cost
FROM customer c
JOIN orders o ON c.customer_id = o.customer_id
JOIN product p ON o.product_id = p.product_id
WHERE p.category = 'gadget'
GROUP BY c.customer_name;
```

- The query selects the customer name and the total cost of the gadgets ordered by each customer.
- The join clause joins the customer, orders, and product tables based on their corresponding IDs.
- The WHERE clause filters the results to only include products with the category of 'gadget'.
- The GROUP BY clause groups the results by the customer name.

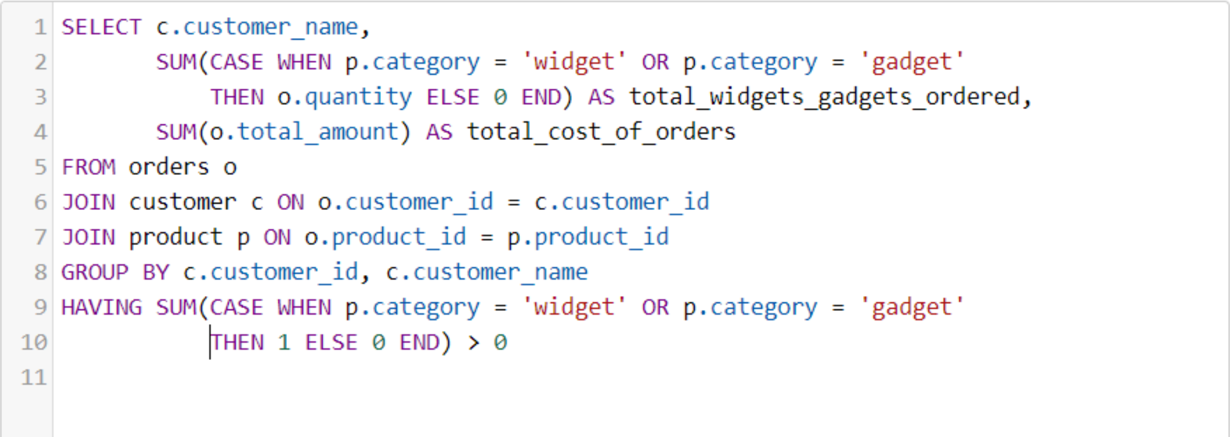
4. Write a query to retrieve the names of the customers who have placed an order for at least one doohickey, along with the total cost of the doohickeys ordered by each customer.



```
SELECT c.customer_name, SUM(o.quantity * p.price) AS total_cost
FROM customer c
JOIN orders o ON c.customer_id = o.customer_id
JOIN product p ON o.product_id = p.product_id
WHERE p.category = 'doohickey'
GROUP BY c.customer_name;
```

- The query selects the customer name and the total cost of the doohickeys ordered by each customer.
- The join clause joins the customer, orders, and product tables based on their corresponding IDs.
- The WHERE clause filters the results to only include products with the category of 'doohickey'.
- The GROUP BY clause groups the results by the customer name.

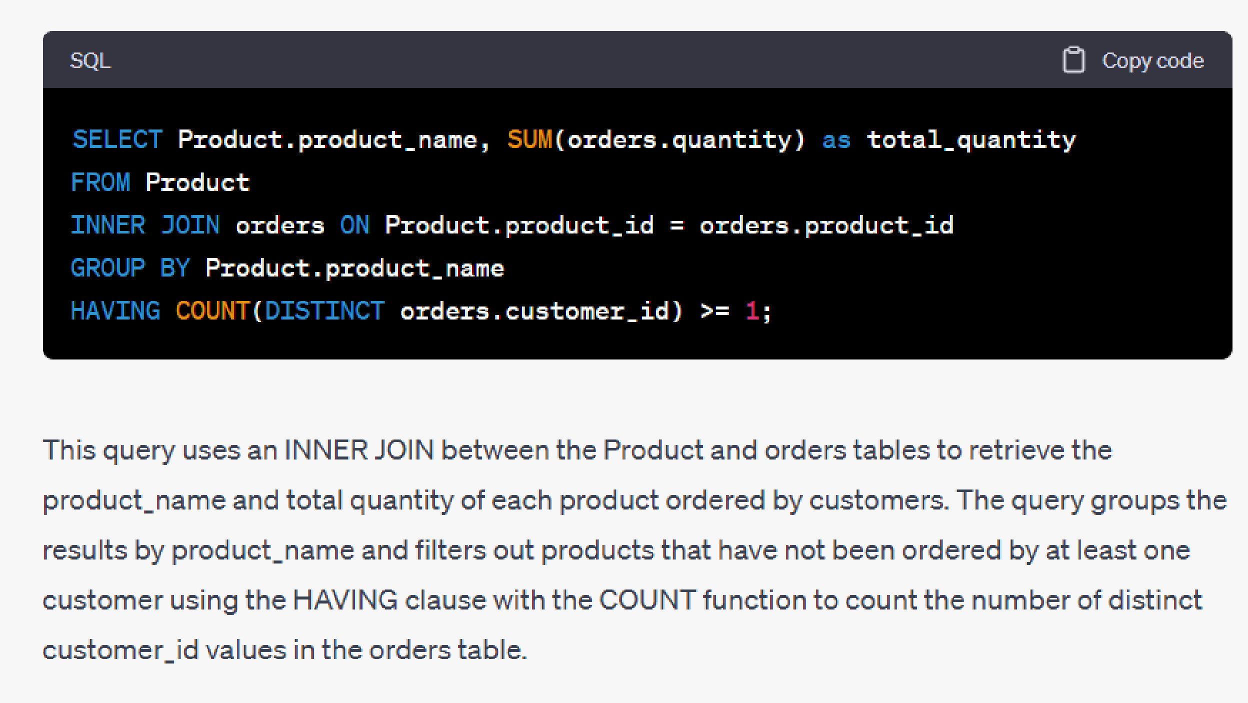
5. **Write a query to retrieve the total number of widgets and gadgets ordered by each customer, along with the total cost of the orders.**



```
SELECT c.customer_name,
       SUM(CASE WHEN p.category = 'widget' OR p.category = 'gadget'
           THEN o.quantity ELSE 0 END) AS total_widgets_gadgets_ordered,
       SUM(o.total_amount) AS total_cost_of_orders
FROM orders o
JOIN customer c ON o.customer_id = c.customer_id
JOIN product p ON o.product_id = p.product_id
GROUP BY c.customer_id, c.customer_name
HAVING SUM(CASE WHEN p.category = 'widget' OR p.category = 'gadget'
           THEN 1 ELSE 0 END) > 0

```

This query joins the orders, customer, and product tables and calculates the total number of widgets and gadgets ordered by each customer, along with the total cost of the orders. The query filters out customers who have not ordered any widgets or gadgets using the HAVING clause.

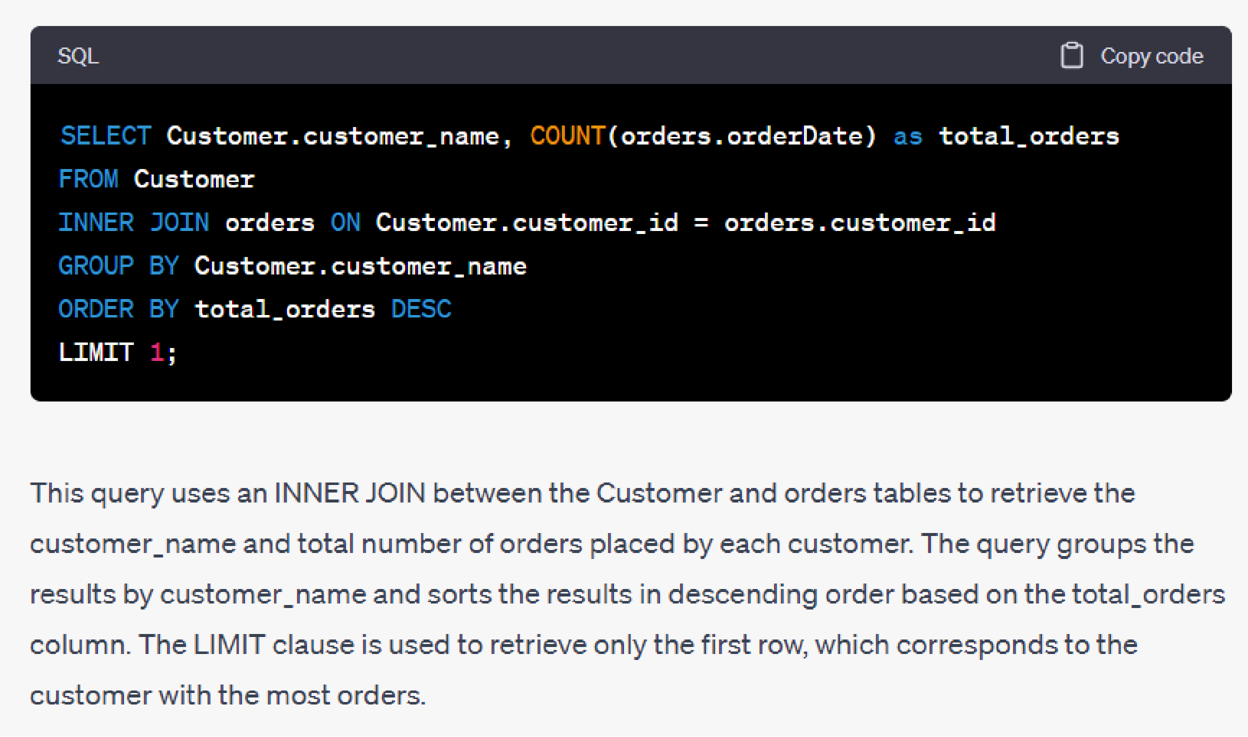
6. **Write a query to retrieve the names of the products that have been ordered by at least one customer, along with the total quantity of each product ordered.**



```sql
SELECT Product.product_name, SUM(orders.quantity) as total_quantity
FROM Product
INNER JOIN orders ON Product.product_id = orders.product_id
GROUP BY Product.product_name
HAVING COUNT(DISTINCT orders.customer_id) >= 1;
```

This query uses an INNER JOIN between the Product and orders tables to retrieve the product_name and total quantity of each product ordered by customers. The query groups the results by product_name and filters out products that have not been ordered by at least one customer using the HAVING clause with the COUNT function to count the number of distinct customer_id values in the orders table.

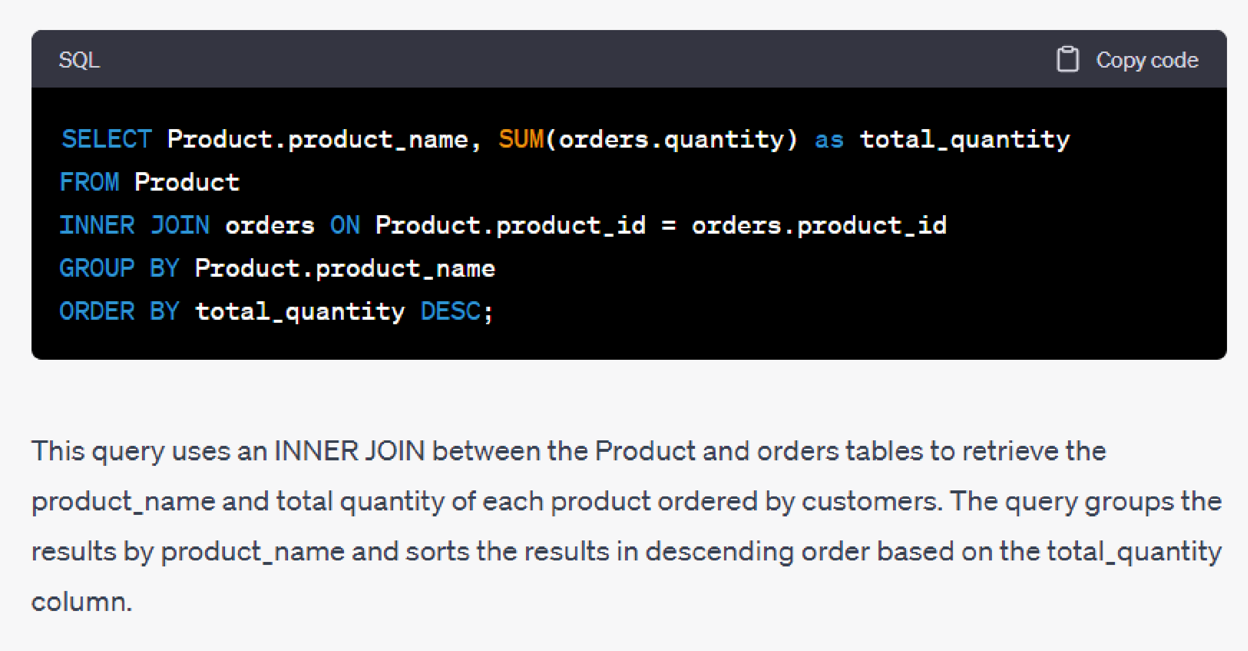
7. **Write a query to retrieve the names of the customers who have placed the most orders, along with the total number of orders placed by each customer.**



```sql
SELECT Customer.customer_name, COUNT(orders.orderDate) as total_orders
FROM Customer
INNER JOIN orders ON Customer.customer_id = orders.customer_id
GROUP BY Customer.customer_name
ORDER BY total_orders DESC
LIMIT 1;
```

This query uses an INNER JOIN between the Customer and orders tables to retrieve the customer_name and total number of orders placed by each customer. The query groups the results by customer_name and sorts the results in descending order based on the total_orders column. The LIMIT clause is used to retrieve only the first row, which corresponds to the customer with the most orders.

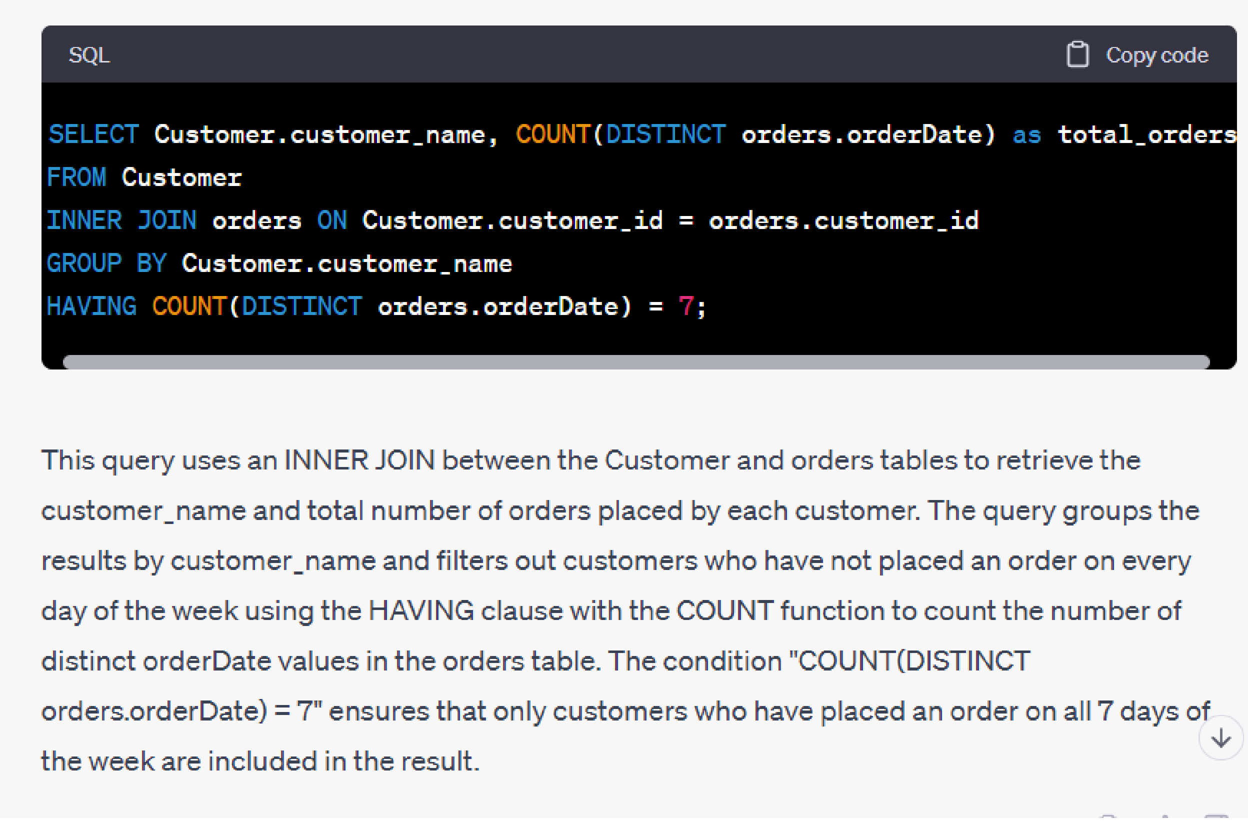
8. **Write a query to retrieve the names of the products that have been ordered the most, along with the total quantity of each product ordered.**



```sql
SELECT Product.product_name, SUM(orders.quantity) as total_quantity
FROM Product
INNER JOIN orders ON Product.product_id = orders.product_id
GROUP BY Product.product_name
ORDER BY total_quantity DESC;
```

This query uses an INNER JOIN between the Product and orders tables to retrieve the product_name and total quantity of each product ordered by customers. The query groups the results by product_name and sorts the results in descending order based on the total_quantity column.

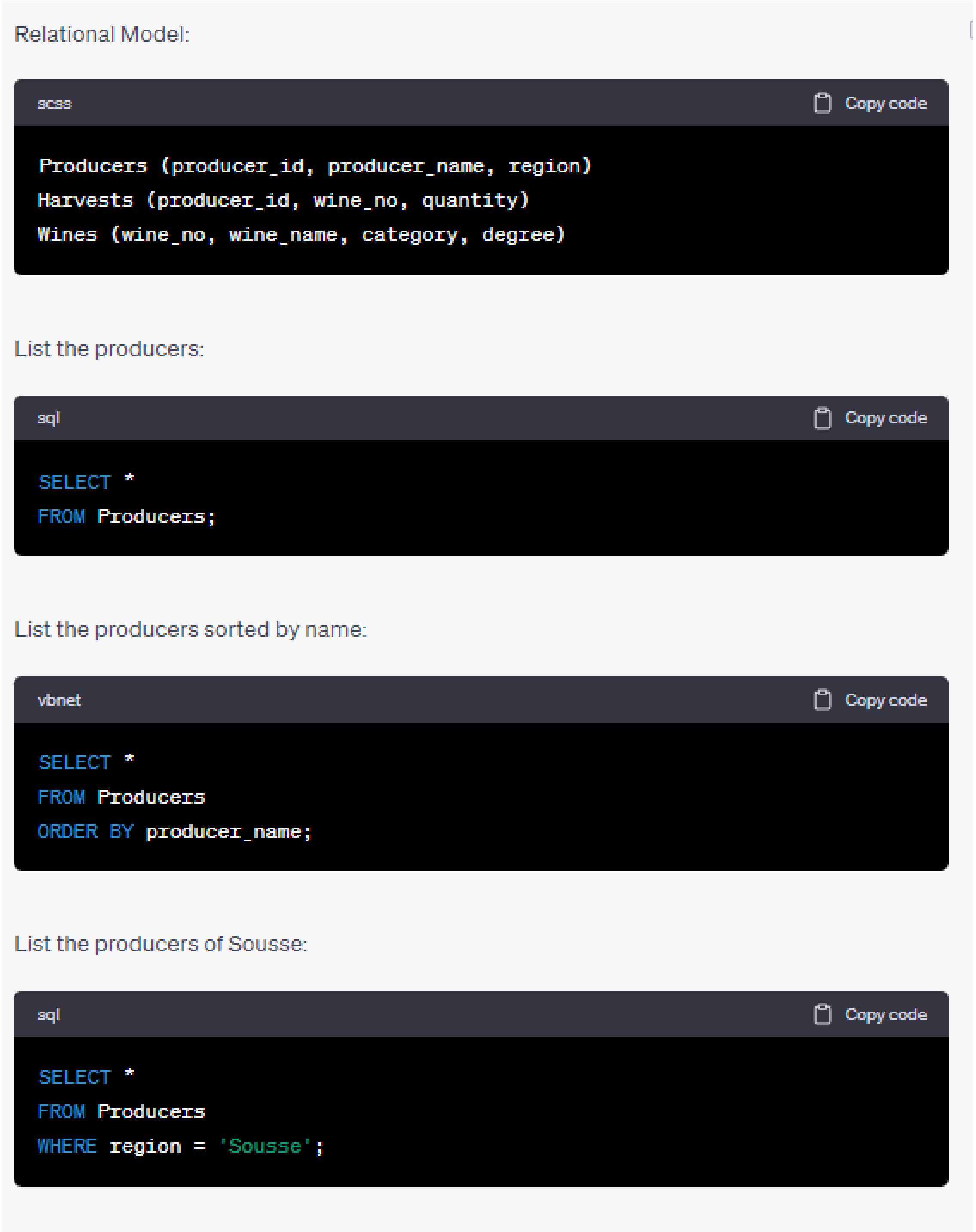
9. **Write a query to retrieve the names of the customers who have placed an order on every day of the week, along with the total number of orders placed by each customer.**



```sql
SELECT Customer.customer_name, COUNT(DISTINCT orders.orderDate) as total_orders
FROM Customer
INNER JOIN orders ON Customer.customer_id = orders.customer_id
GROUP BY Customer.customer_name
HAVING COUNT(DISTINCT orders.orderDate) = 7;
```

This query uses an INNER JOIN between the Customer and orders tables to retrieve the customer_name and total number of orders placed by each customer. The query groups the results by customer_name and filters out customers who have not placed an order on every day of the week using the HAVING clause with the COUNT function to count the number of distinct orderDate values in the orders table. The condition "COUNT(DISTINCT orders.orderDate) = 7" ensures that only customers who have placed an order on all 7 days of the week are included in the result.

7. Relational database management



Relational Model:

```scss
Producers (producer_id, producer_name, region)
Harvests (producer_id, wine_no, quantity)
Wines (wine_no, wine_name, category, degree)
```

List the producers:

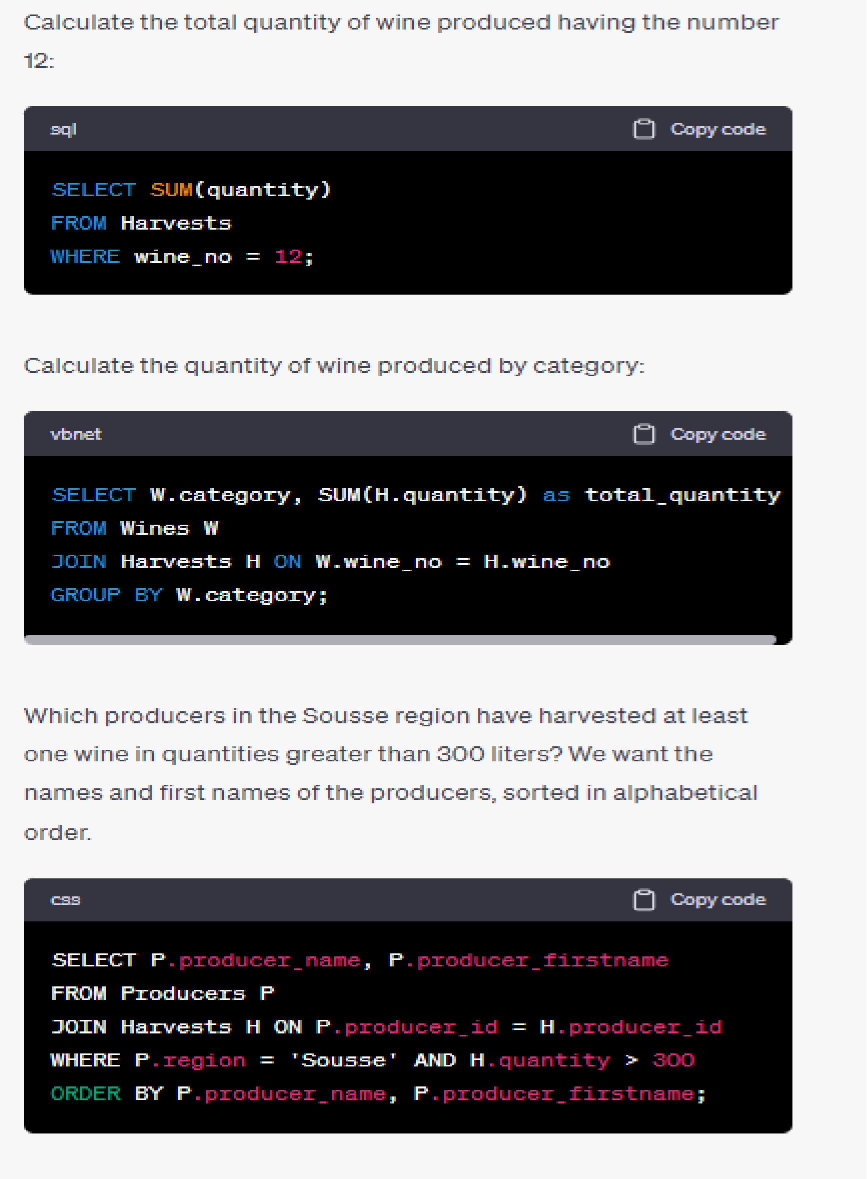
```sql
SELECT *
FROM Producers;
```

List the producers sorted by name:

```vbnet
SELECT *
FROM Producers
ORDER BY producer_name;
```

List the producers of Sousse:

```sql
SELECT *
FROM Producers
WHERE region = 'Sousse';
```



Calculate the total quantity of wine produced having the number 12:

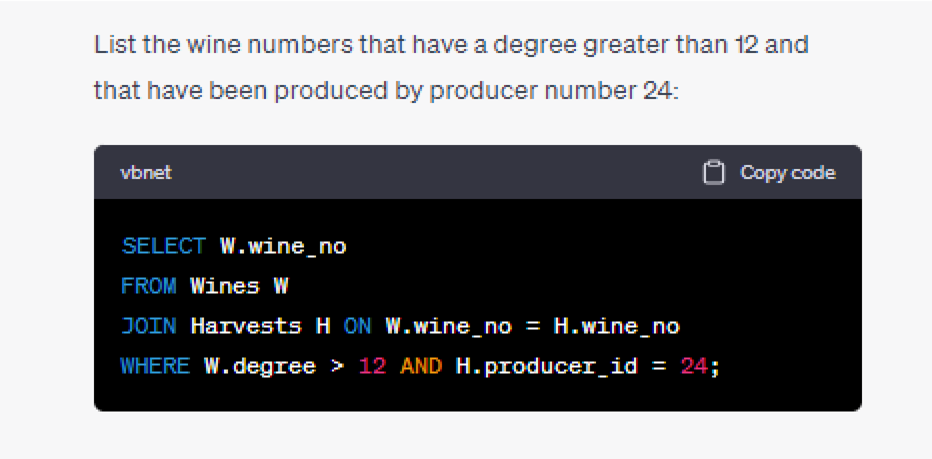
```sql
SELECT SUM(quantity)
FROM Harvests
WHERE wine_no = 12;
```

Calculate the quantity of wine produced by category:

```vbnet
SELECT W.category, SUM(H.quantity) as total_quantity
FROM Wines W
JOIN Harvests H ON W.wine_no = H.wine_no
GROUP BY W.category;
```

Which producers in the Sousse region have harvested at least one wine in quantities greater than 300 liters? We want the names and first names of the producers, sorted in alphabetical order.

```css
SELECT P.producer_name, P.producer_firstname
FROM Producers P
JOIN Harvests H ON P.producer_id = H.producer_id
WHERE P.region = 'Sousse' AND H.quantity > 300
ORDER BY P.producer_name, P.producer_firstname;
```



List the wine numbers that have a degree greater than 12 and that have been produced by producer number 24:

```vbnet
SELECT W.wine_no
FROM Wines W
JOIN Harvests H ON W.wine_no = H.wine_no
WHERE W.degree > 12 AND H.producer_id = 24;
```

https://colab.research.google.com/drive/1ONGyb2ybutqlIiQTjllh_fqsm0smAMNj#scrollTo=iQ4_sALt0HDc

8. **Python syntax checkpoint**

```
print("Welcome to Python Pizza Deliveries!")
size = input("What size pizza do you want? S, M, or L ")
add_pepperoni = input("Do you want pepperoni? Y or N ")
extra_cheese = input("Do you want extra cheese? Y or N ")

# calculate base price based on size
if size == "S":
    base_price = 15
elif size == "M":
    base_price = 20
elif size == "L":
    base_price = 25
else:
    print("Invalid input for size. Please enter S, M, or L.")
    exit()

# add pepperoni cost to base price
if add_pepperoni == "Y":
    if size == "S":
        base_price += 2
    else:
        base_price += 3

# add extra cheese cost to base price
if extra_cheese == "Y":
    base_price += 1

print (f"Your final bill is: ${base_price}.")
```

9. Python data structure

```
# create an empty shopping list
shopping_list = []

# define the menu options
menu = """
Menu:
1. Add item
2. Remove item
3. View list
4. Quit
"""

# loop until the user chooses to quit
while True:
    # display the menu
    print(menu)

    # get the user's selection
    choice = input("Enter your choice (1-4): ")

    # handle the user's selection
    if choice == '1':
        # check if the list is full
        if len(shopping_list) == 5:
            print("Sorry, the list is full.")
        else:
            # prompt the user to enter an item and add it to the
list
            item = input("Enter an item to add: ")
            shopping_list.append(item)
            print(item, "has been added to the list.")
    elif choice == '2':
        # check if the list is empty
        if len(shopping_list) == 0:
            print("Sorry, the list is empty.")
        else:
            # prompt the user to enter an item and remove it from
the list
```

```
        item = input("Enter an item to remove: ")
        if item in shopping_list:
            shopping_list.remove(item)
            print(item, "has been removed from the list.")
        else:
            print(item, "is not in the list.")
elif choice == '3':
    # display the list of items
    print("Current shopping list:")
    for item in shopping_list:
        print("- ", item)
elif choice == '4':
    # exit the program
    print("Goodbye!")
    break
else:
    # handle invalid input
    print("Invalid choice. Please enter a number from 1 to 4.")
```

10. Python functions

```python
#python function
# define basic mathematical functions
def add(num1, num2):
    return num1 + num2

def subtract(num1, num2):
    return num1 - num2

def multiply(num1, num2):
    return num1 * num2

def divide(num1, num2):
    return num1 / num2

# create dictionary to assign functions to operation symbols
operations = {
    "+": add,
    "-": subtract,
    "*": multiply,
    "/": divide
}

# define calculator function
def calculator():
    num1 = float(input("What is the first number? "))
    should_continue = True

    while should_continue:
        for symbol in operations:
            print(symbol)

        operation_symbol = input("Pick an operation symbol from the line above: ")
        num2 = float(input("What is the next number? "))

        calculation_function = operations[operation_symbol]
        answer = calculation_function(num1, num2)

        print(f"{num1} {operation_symbol} {num2} = {answer}")
        choice = input(f"Type 'y' to continue calculating with
{answer}, or 'n' to start a new calculation: ")
        if choice == "y":
            num1 = answer
        else:
            should_continue = False
            calculator()

calculator()
```

11. Object oriented programming: creating a bank account

```
class Account:
    def __init__(self, account_number, account_balance,
account_holder):
        self.account_number = account_number
        self.account_balance = account_balance
        self.account_holder = account_holder

    def deposit(self, amount):
        self.account_balance += amount

    def withdraw(self, amount):
        if self.account_balance >= amount:
            self.account_balance -= amount
        else:
            print("Error: Insufficient funds.")

    def check_balance(self):
        return self.account_balance
#to test the class, we can create an instance of it and ccall its
methods
my_account = Account("123456789", 1000.0, "John Doe")

print("Initial balance:", my_account.check_balance())

my_account.deposit(500.0)
print("Balance after deposit:", my_account.check_balance())

my_account.withdraw(200.0)
print("Balance after withdrawal:", my_account.check_balance())

my_account.withdraw(2000.0)  # Should print an error message
print("Balance after withdrawal:", my_account.check_balance())
```

12. Python numpy checkpoint

```
#python numpy checkpoint
import numpy as np
# Create the grades array
grades = np.array([85, 90, 88, 92, 95, 80, 75, 98, 89, 83])
mean = np.mean(grades) #mean, median, and standard deviation
median = np.median(grades)
std_dev = np.std(grades)
max_grade = np.max(grades) # Find the maximum and minimum grades
min_grade = np.min(grades)
# Sort the grades in ascending order
sorted_grades = np.sort(grades)
# Find the index of the highest grade
highest_index = np.argmax(grades)
# Count the number of students who scored above 90
num_above_90 = np.sum(grades > 90)
# Calculate the percentage of students who scored above 90
percent_above_90 = np.mean(grades > 90) * 100
# Calculate the percentage of students who scored below 75
percent_below_75 = np.mean(grades < 75) * 100
# Extract all the grades above 90 and put them in a new array called
"high_performers"
high_performers = grades[grades > 90]
# Create a new array called "passing_grades" that contains all the
grades above 75
passing_grades = grades[grades > 75]
# Print the results
print("Grades:", grades)
print("Mean:", mean)
print("Median:", median)
print("Standard Deviation:", std_dev)
print("Max Grade:", max_grade)
print("Min Grade:", min_grade)
print("Sorted Grades:", sorted_grades)
print("Index of Highest Grade:", highest_index)
print("Number of Grades Above 90:", num_above_90)
print("Percentage of Grades Above 90:", percent_above_90)
print("Percentage of Grades Below 75:", percent_below_75)
print("High Performers:", high_performers)
print("Passing Grades:", passing_grades)
```

13. File handling checkpoint

```
#file handling checkpoint
import numpy as np

# Open the CSV file
with open('/content/Lending-Company-Saving.csv', 'r') as f:
    # Use NumPy's genfromtxt function to read the file into a numpy
array
    loan_data = np.genfromtxt(f, delimiter=',', skip_header=1)

# Find the mean, median, and standard deviation of the total loan
amounts
mean_loan = np.mean(loan_data[:, 6])
median_loan = np.median(loan_data[:, 6])
std_loan = np.std(loan_data[:, 6])

# Print the results
print("Mean loan amount: ", mean_loan)
print("Median loan amount: ", median_loan)
print("Standard deviation of loan amount: ", std_loan)
```