1. **Implement program for given integer array num move all 0's to the end of it while maintaing the relative order of the non zero.**

In python:

Mehod 1:

def move\_zeroes(nums):

# Pointer for the position to place the next non-zero element

position = 0

# Traverse through the array

for i in range(len(nums)):

if nums[i] != 0:

# Swap non-zero element with the element at the position pointer

nums[position], nums[i] = nums[i], nums[position]

position += 1

return nums

# Example usage

nums = [0, 1, 0, 3, 12]

move\_zeroes(nums)

print(nums) # Output: [1, 3, 12, 0, 0]

method 2:

def move\_zeroes(nums):

# Create an index to keep track of the position for non-zero elements

index = 0

# Iterate through the list

for num in nums:

if num != 0:

nums[index] = num

index += 1

# Fill the remaining positions with zeroes

while index < len(nums):

nums[index] = 0

index += 1

return nums

# Example usage

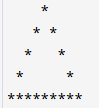
nums = [0, 1, 0, 3, 12]

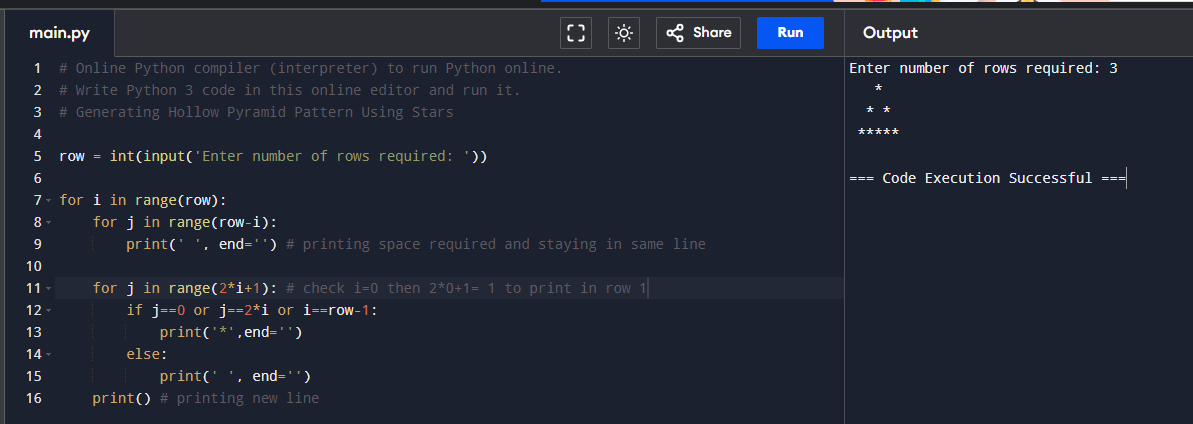
move\_zeroes(nums)

print(nums) # Output: [1, 3, 12, 0, 0]



**2.Program to Print Hollow Star Pyramid using any programming language?**





# Generating Hollow Pyramid Pattern Using Stars

row = int(input('Enter number of rows required: '))

for i in range(row):

for j in range(row-i):

print(' ', end='') # printing space required and staying in same line

for j in range(2\*i+1): # check i=0 then 2\*0+1= 1 to print in row 1

if j==0 or j==2\*i or i==row-1:

print('\*',end='')

else:

print(' ', end='')

print() # printing new line

**3. Implement a program to Count the Occurrences of Each Word in a String using any programming language.**

# occurrence of count

string=input("Enter string:")

word=input("Enter word:")

a=[]

count=0

a=string.split(" ")

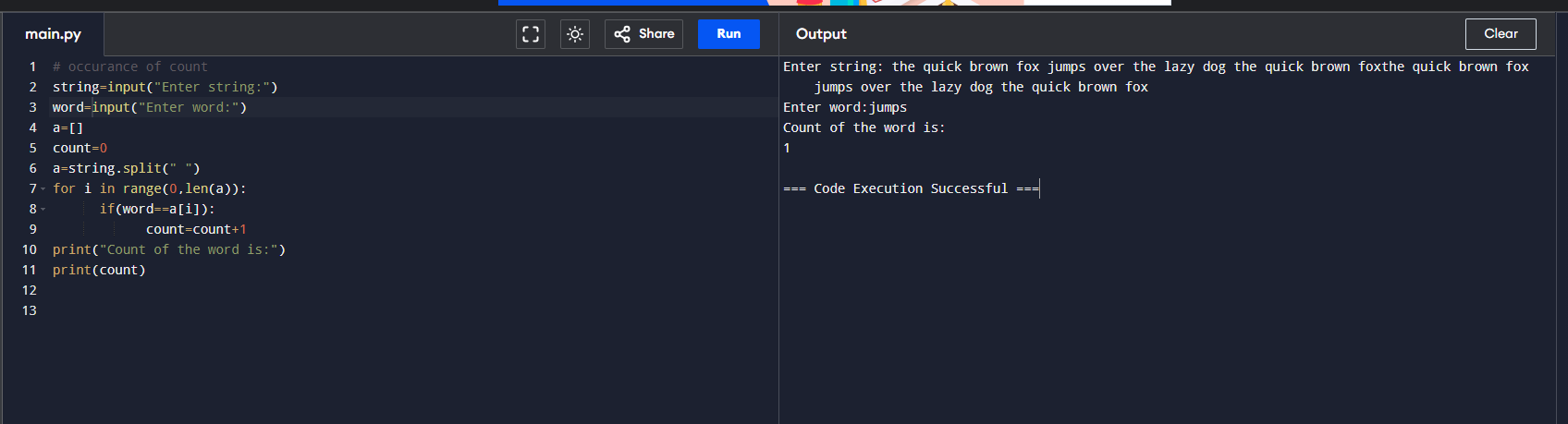
for i in range(0,len(a)):

if(word==a[i]):

count=count+1

print("Count of the word is:")

print(count)



**4. Implement a program for given an array *‘a’* of size *‘n’*-1 with elements of range 1 to ‘n’. The array does not contain any duplicates. Your task is to find the missing number.**

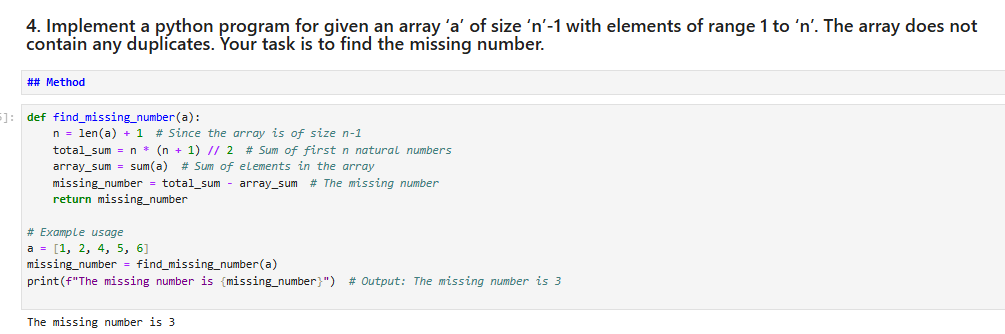
**For example:**

**Input:**

**'a' = [1, 2, 4, 5], 'n' = 5**

**Output :**

**3**



def find\_missing\_number(a):

n = len(a) + 1 # Since the array is of size n-1

total\_sum = n \* (n + 1) // 2 # Sum of first n natural numbers

array\_sum = sum(a) # Sum of elements in the array

missing\_number = total\_sum - array\_sum # The missing number

return missing\_number

# Example usage

a = [1, 2, 4, 5, 6]

missing\_number = find\_missing\_number(a)

print(f"The missing number is {missing\_number}") # Output: The missing number is 3

**5. Implement oops concepts for Bank management system.**

from abc import ABC, abstractmethod

# Abstract Base Class for BankAccount

class BankAccount(ABC):

def \_\_init\_\_(self):

# Initialize the variables

self.uid = 0

self.name = ""

self.age = 0

self.address = ""

self.amount = 0

# Abstract methods

@abstractmethod

def account\_creation(self):

pass

@abstractmethod

def account\_deletion(self):

pass

@abstractmethod

def deposit\_money(self):

pass

@abstractmethod

def withdraw\_money(self):

pass

@abstractmethod

def show\_balance(self):

pass

@abstractmethod

def update\_address(self):

pass

# subclass implementing the abstract methods

class currentaccount(BankAccount):

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.customer\_id\_list = []

def account\_creation(self):

# Implementation of abstract method

self.uid += 1

self.customer\_id\_list.append(self.uid)

print("Enter Your Name:")

self.name = input()

print("Enter Your Age:")

self.age = int(input())

print("Enter Your Address:")

self.address = input()

print(f"Account Created Successfully. Your Bank Account Number is {self.uid}")

# Implementation of abstract method

def account\_deletion(self):

print("Account deletion is not allowed in BasicBankAccount.")

def deposit\_money(self):

amount = float(input("Enter the amount to deposit: ₹"))

if amount > 0:

self.amount += amount

print(f"Successfully deposited ₹{amount}. Total balance is ₹{self.amount}.")

else:

print("Invalid amount. Deposit must be positive.")

def withdraw\_money(self):

amount = float(input("Enter the amount to withdraw: ₹"))

if amount > 0 and amount <= self.amount:

self.amount -= amount

print(f"Successfully withdrew ₹{amount}. Total balance is ₹{self.amount}.")

else:

print("Invalid amount or insufficient balance.")

def show\_balance(self):

print(f"Total money in your account is ₹{self.amount}.")

def update\_address(self):

print(f"Current Address: {self.address}")

print("Enter new address:")

self.address = input()

print(f"Address updated to {self.address}.")

# Inheritance - savingsacccount extends currentaccount

class savingsacccount(currentaccount):

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.reward\_points = 0

def account\_creation(self):

# Overriding method

super().account\_creation() # Calls the base class method

print("Premium Account created. Enjoy additional benefits!")

def deposit\_money(self):

super().deposit\_money()

self.reward\_points += 1

print(f"Reward points earned: {self.reward\_points}.")

def withdraw\_money(self):

super().withdraw\_money()

def show\_balance(self):

super().show\_balance()

print(f"Reward points: {self.reward\_points}")

accounts = {}

while True:

phone\_number = input("Enter your phone number: ")

if phone\_number in accounts:

account = accounts[phone\_number]

print("Welcome to bank services")

print("1. Account Deletion")

print("2. Deposit Money")

print("3. Withdraw Money")

print("4. Show Balance")

print("5. Update Address")

print("6. Exit")

choice = int(input())

if choice == 1:

account.account\_deletion() # Polymorphism

elif choice == 2:

account.deposit\_money()

elif choice == 3:

account.withdraw\_money()

elif choice == 4:

account.show\_balance()

elif choice == 5:

account.update\_address()

elif choice == 6:

break

else:

print("You don't have an account. Please select an option to create an account.")

print("1. Savings Account Creation")

print("2. Current Account Creation")

print("3. Exit")

choice = int(input())

if choice == 1:

new\_account = savingsacccount()

new\_account.account\_creation()

accounts[phone\_number] = new\_account

elif choice == 2:

new\_account = currentaccount()

new\_account.account\_creation()

accounts[phone\_number] = new\_account

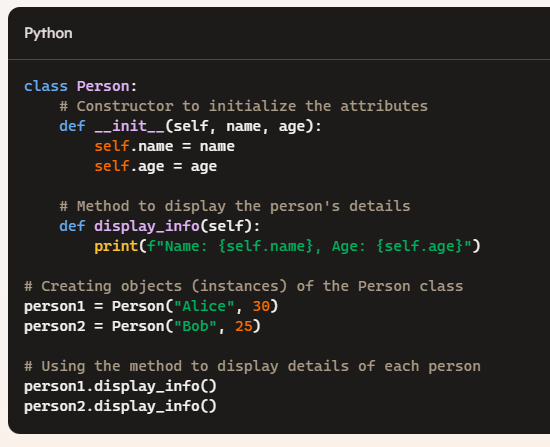
elif choice == 3:

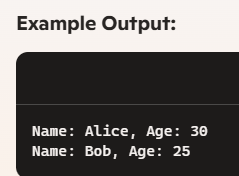
break

**Output :**

Enter your phone number: 1234  
You don't have an account. Please select an option to create an account.  
1. Savings Account Creation  
2. Current Account Creation  
3. Exit  
1  
Enter Your Name:  
Harini  
Enter Your Age:  
20  
Enter Your Address:  
hyderabad  
Account Created Successfully. Your Bank Account Number is 1  
Premium Account created. Enjoy additional benefits!  
Enter your phone number: 123  
Welcome to bank services  
1. Account Deletion  
2. Deposit Money  
3. Withdraw Money  
4. Show Balance  
5. Update Address  
6. Exit  
2  
Enter the amount to deposit: ₹200000  
Successfully deposited ₹200000.0. Total balance is ₹200000.0.  
Reward points earned: 1.  
Enter your phone number: 8886366338  
Welcome to bank services  
1. Account Deletion  
2. Deposit Money  
3. Withdraw Money  
4. Show Balance  
5. Update Address  
6. Exit  
3  
Enter the amount to withdraw: ₹56123  
Successfully withdrew ₹56123.0. Total balance is ₹143877.0.  
Enter your phone number: 8886366338  
Welcome to bank services  
1. Account Deletion  
2. Deposit Money  
3. Withdraw Money  
4. Show Balance  
5. Update Address  
6. Exit  
4  
Total money in your account is ₹143877.0.  
Reward points: 1

**6. Implement a Class, object and constructors with supporting program.**

****

****

**7. Create an array using methods add new element, access the element, remove the element in array using any programming language.**

# creating array, accessing, removing

import array

# Create an array of floating values

arr = array.array('f', [1.2, 2.5, 3.5, 4.5, 5.5])

print("Array:", arr)

# Add a new element to the array

arr.append(6.5)

print("Array after adding an element:", arr)

# Access elements in the array

first\_element = arr[0]

third\_element = arr[2]

print("First element:", first\_element)

print("Third element:", third\_element)

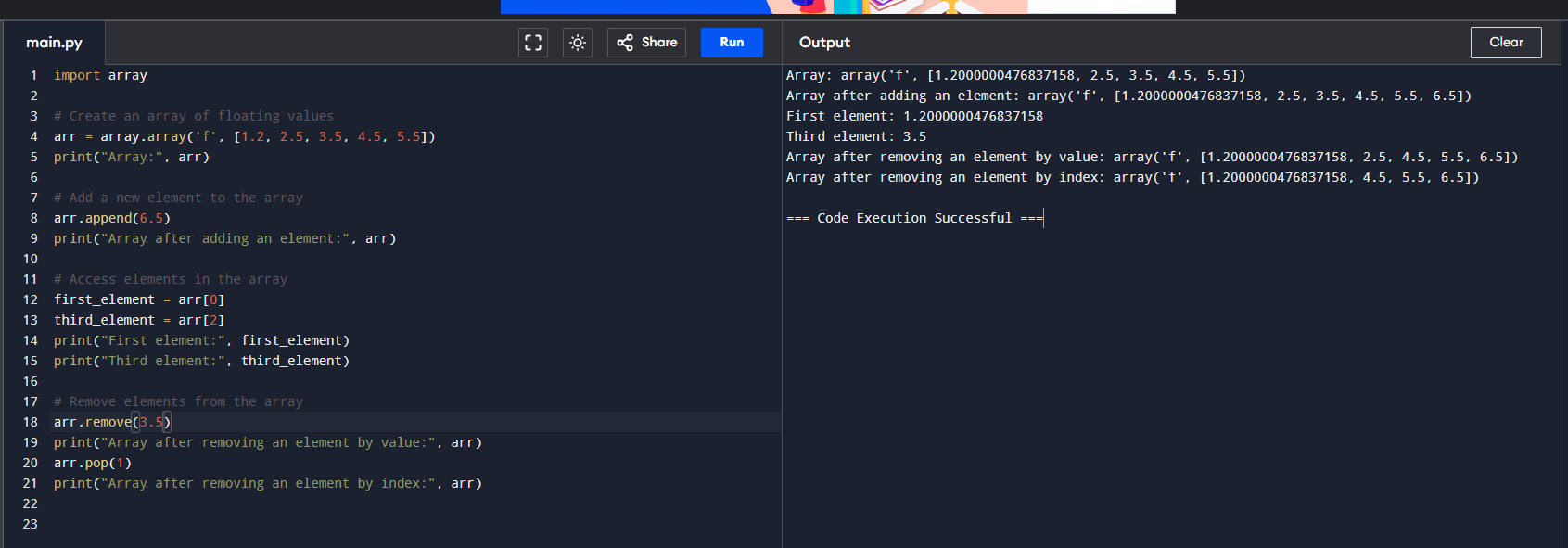
# Remove elements from the array

arr.remove(3.5)

print("Array after removing an element by value:", arr)

arr.pop(1)

print("Array after removing an element by index:", arr)



**8. Implement a program for given an array of integers nums and an integer target, return indices of the two numbers such that they add up to target.**

**You may assume that each input would have exactly one solution, and you may not use the same element twice.**

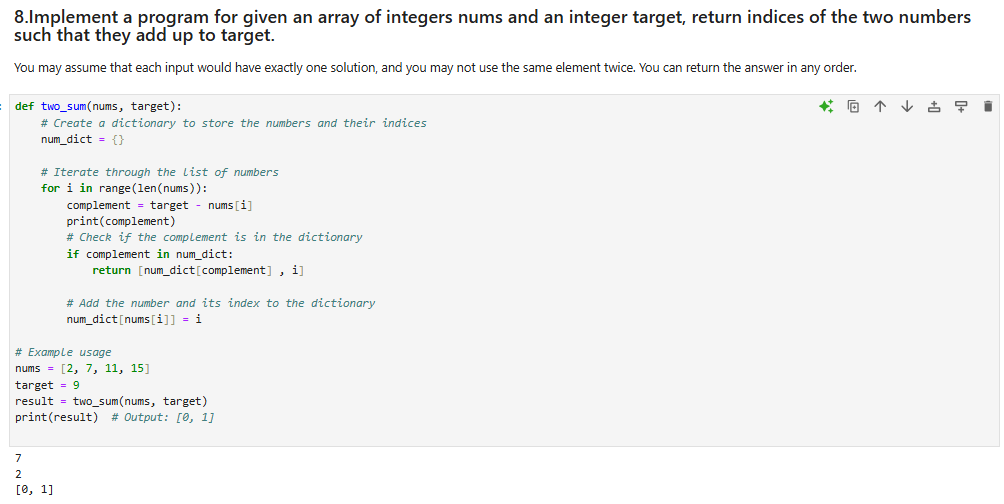
**You can return the answer in any order.**

**Example 1:**

**Input: nums = [2,7,11,15], target = 9**

**Output: [0,1]**

**Explanation: Because nums[0] + nums[1] == 9, we return [0, 1].**



def two\_sum(nums, target):

# Create a dictionary to store the numbers and their indices

num\_dict = {}

# Iterate through the list of numbers

for i in range(len(nums)):

complement = target - nums[i]

print(complement)

# Check if the complement is in the dictionary

if complement in num\_dict:

return [num\_dict[complement] , i]

# Add the number and its index to the dictionary

num\_dict[nums[i]] = i

# Example usage

nums = [2, 7, 11, 15]

target = 9

result = two\_sum(nums, target)

print(result) # Output: [0, 1]

**9. Implement a program for given an array nums with n objects colored red, white, or blue, sort them**[**in-place**](https://en.wikipedia.org/wiki/In-place_algorithm)**so that objects of the same color are adjacent, with the colors in the order red, white, and blue.**

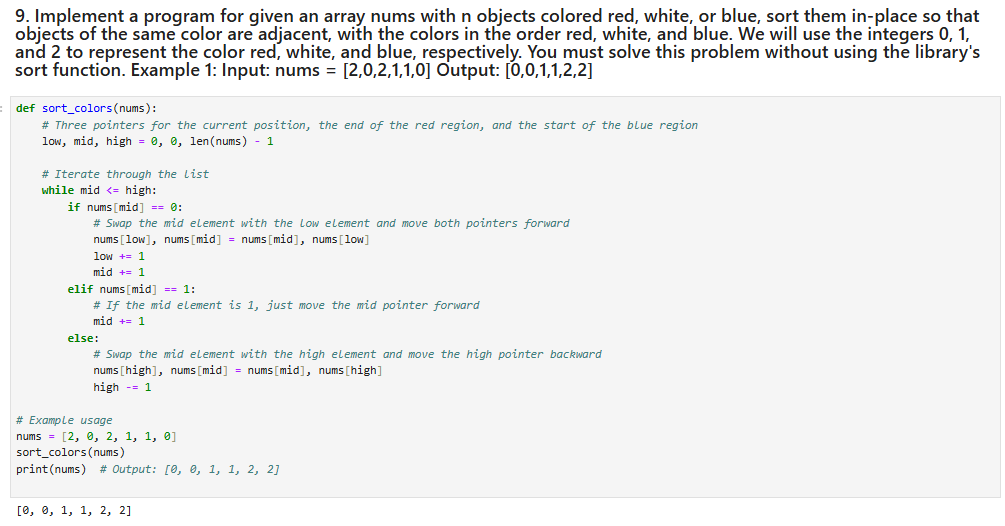
**We will use the integers 0, 1, and 2 to represent the color red, white, and blue, respectively.**

**You must solve this problem without using the library's sort function.**

**Example 1:**

**Input: nums = [2,0,2,1,1,0]**

**Output: [0,0,1,1,2,2]**



def sort\_colors(nums):

# Three pointers for the current position, the end of the red region, and the start of the blue region

low, mid, high = 0, 0, len(nums) - 1

# Iterate through the list

while mid <= high:

if nums[mid] == 0:

# Swap the mid element with the low element and move both pointers forward

nums[low], nums[mid] = nums[mid], nums[low]

low += 1

mid += 1

elif nums[mid] == 1:

# If the mid element is 1, just move the mid pointer forward

mid += 1

else:

# Swap the mid element with the high element and move the high pointer backward

nums[high], nums[mid] = nums[mid], nums[high]

high -= 1

# Example usage

nums = [2, 0, 2, 1, 1, 0]

sort\_colors(nums)

print(nums) # Output: [0, 0, 1, 1, 2, 2]

**10.Explain Exception Handling with suitable example. Also name 5 common Exceptions**.

def divide\_numbers(x, y):

try:

# Try to perform the division

result = x / y

except ZeroDivisionError:

# Handle the case where division by zero is attempted

print("Error: Cannot divide by zero.")

return None

else:

# Execute if no exceptions were raised

print(f"The result is {result}.")

return result

finally:

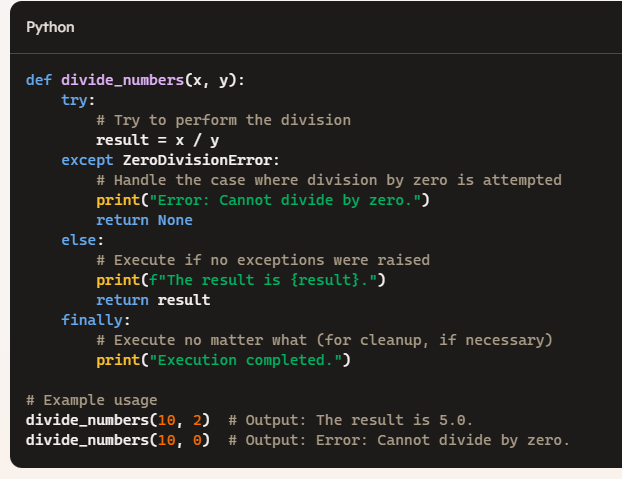
# Execute no matter what (for cleanup, if necessary)

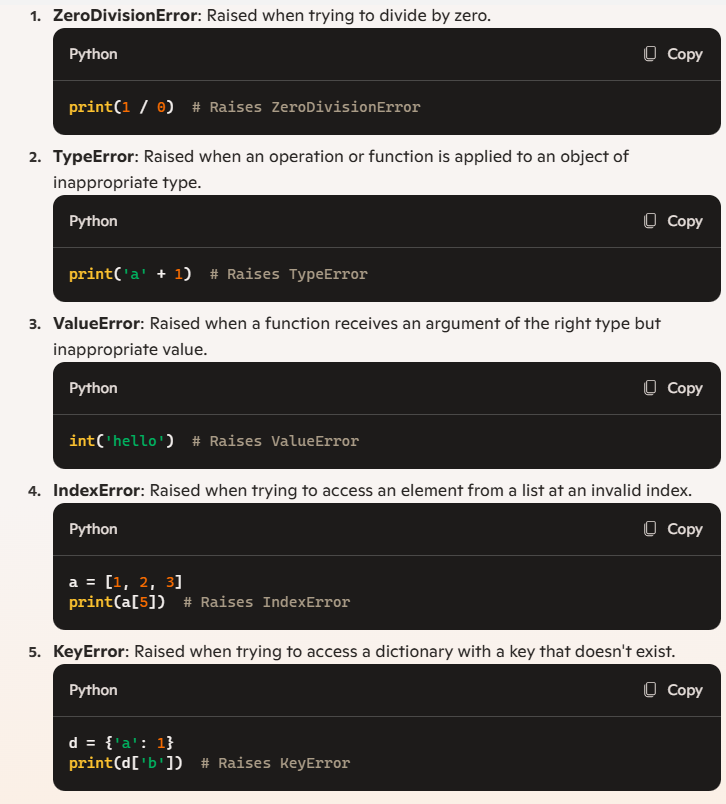
print("Execution completed.")

# Example usage

divide\_numbers(10, 2) # Output: The result is 5.0.

divide\_numbers(10, 0) # Output: Error: Cannot divide by zero.





**11. Explain ACID Properties with suitable transaction example.**

The ACID properties are crucial in ensuring reliable, consistent, and secure transactions in database management systems. They stand for Atomicity, Consistency, Isolation, and Durability.

ACID Properties

1. Atomicity:

- Definition: Ensures that each transaction is treated as a single unit, which either completes entirely or does not happen at all. If any part of the transaction fails, the entire transaction is rolled back.

- Example: Consider a bank transaction where Alice transfers \$100 to Bob. The transaction involves two operations: debiting \$100 from Alice's account and crediting \$100 to Bob's account. Atomicity ensures that both operations either succeed or fail together. If the debit succeeds but the credit fails, the transaction is rolled back, and Alice's account is not debited.

2. Consistency:

- Definition: Ensures that a transaction brings the database from one valid state to another, maintaining database rules, such as constraints, cascades, triggers, etc.

- Example: Continuing the bank example, consistency ensures that after Alice’s \$100 transfer, the total balance remains consistent (the sum of all accounts remains unchanged). If Alice had \$1000 and Bob had \$500, after the transaction, Alice has \$900 and Bob has \$600, maintaining the total \$1500.

3. Isolation:

- Definition: Ensures that concurrently executing transactions do not affect each other’s execution. Intermediate transaction results are hidden from other transactions until they are finalized.

- Example: Imagine two transactions occurring simultaneously: Alice transferring \$100 to Bob while Bob transfers \$50 to Charlie. Isolation ensures that each transaction occurs as if it were the only transaction in the system. Even if both transactions are processed at the same time, they do not interfere with each other.

4. Durability:

- Definition: Ensures that once a transaction has been committed, it will remain committed even in the case of a system failure (e.g., power outage, crash). The changes made by the transaction are permanently recorded.

- Example: After Alice's \$100 transfer to Bob is successfully committed, the changes (Alice’s balance debited and Bob’s balance credited) persist even if the database system crashes immediately afterward. When the system recovers, the committed changes are retained.

Example Transaction

Let's walk through an example of an ACID-compliant transaction involving a simple bank transfer:

1. Atomicity:

- Start transaction.

- Debit \$100 from Alice's account.

- Credit \$100 to Bob's account.

- If any operation fails, roll back the transaction.

2. Consistency:

- Ensure debit and credit operations adhere to the bank’s rules (e.g., sufficient balance, valid account numbers).

- Post-transaction, check that all accounts' total balances remain consistent.

3. Isolation:

- While the transfer is processing, ensure no other transactions can read intermediate states of Alice’s and Bob’s accounts.

- Only once the transfer is complete do other transactions see the final state.

4. Durability:

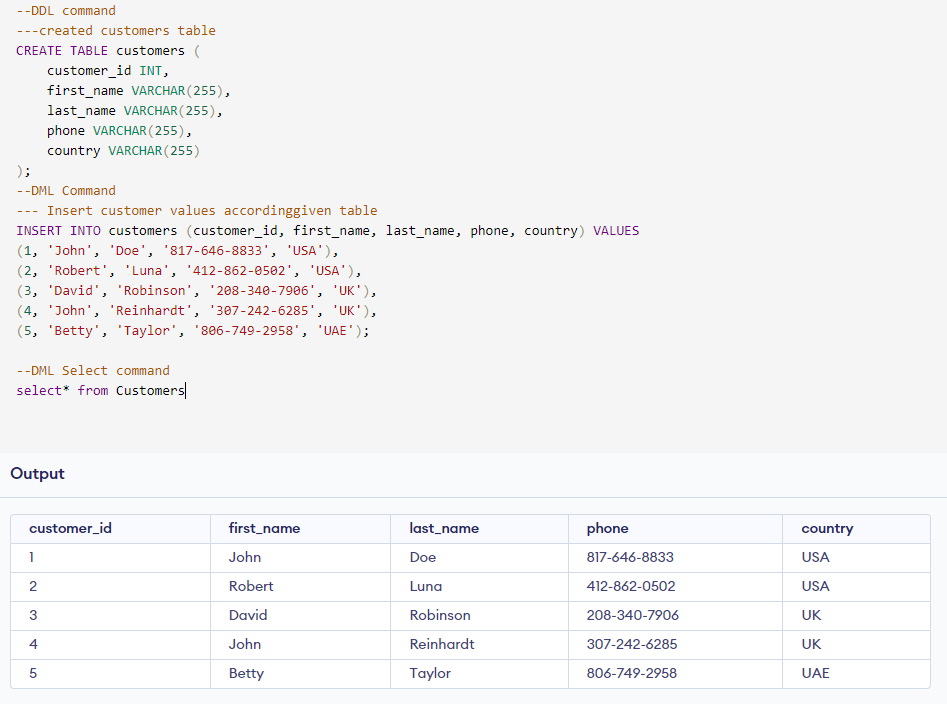
- Once the transfer is committed, write the changes to stable storage.

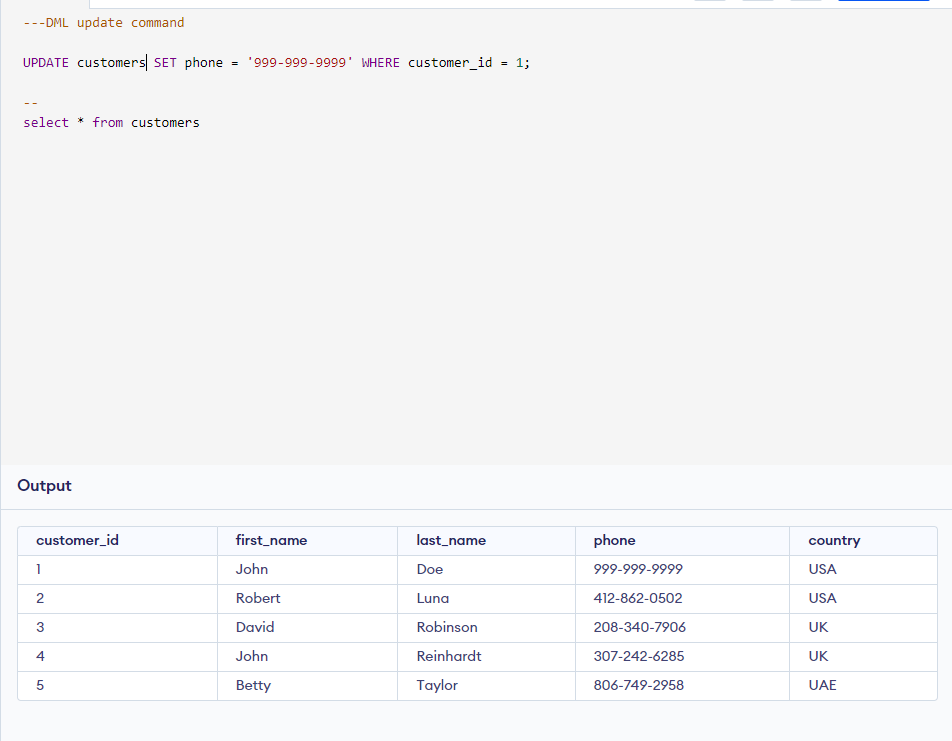
- Ensure changes are logged, so they persist even if the system fails immediately after committing.

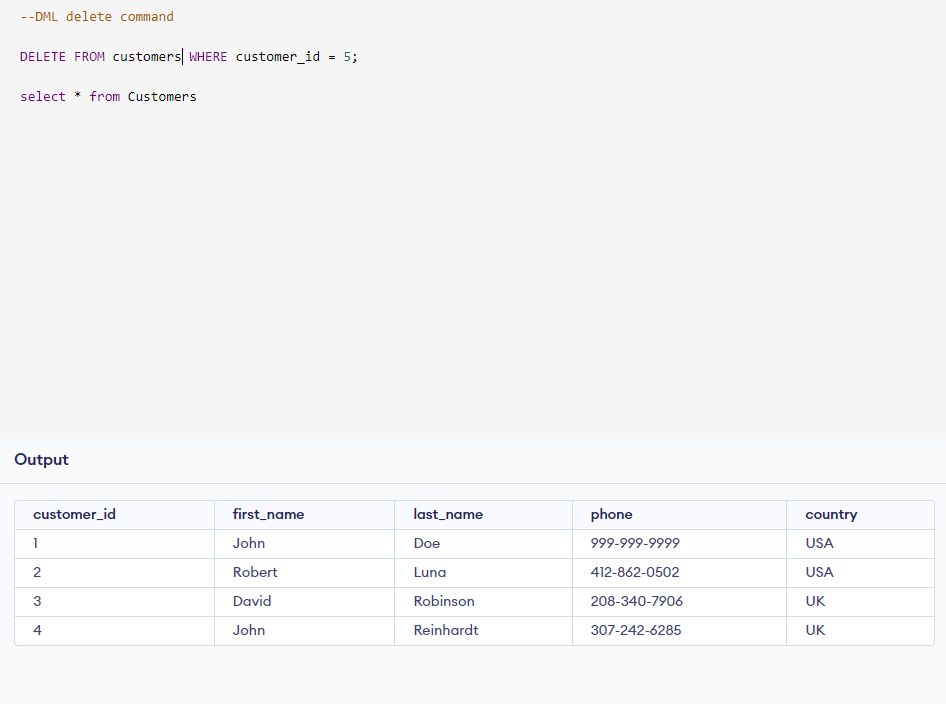
**12.Perform all DDL and DML commands on below given table.**

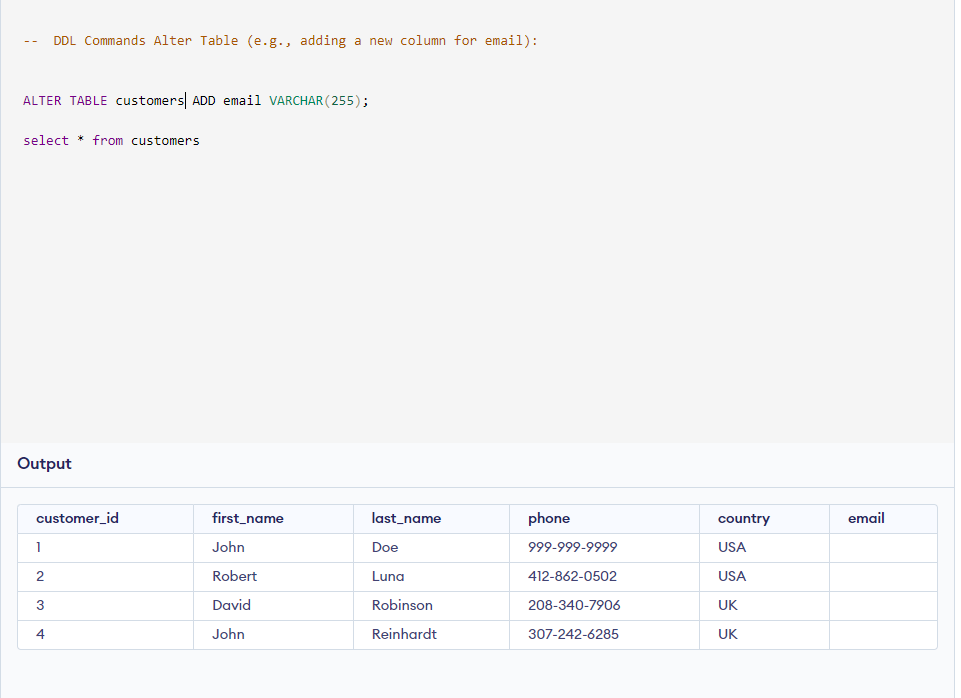


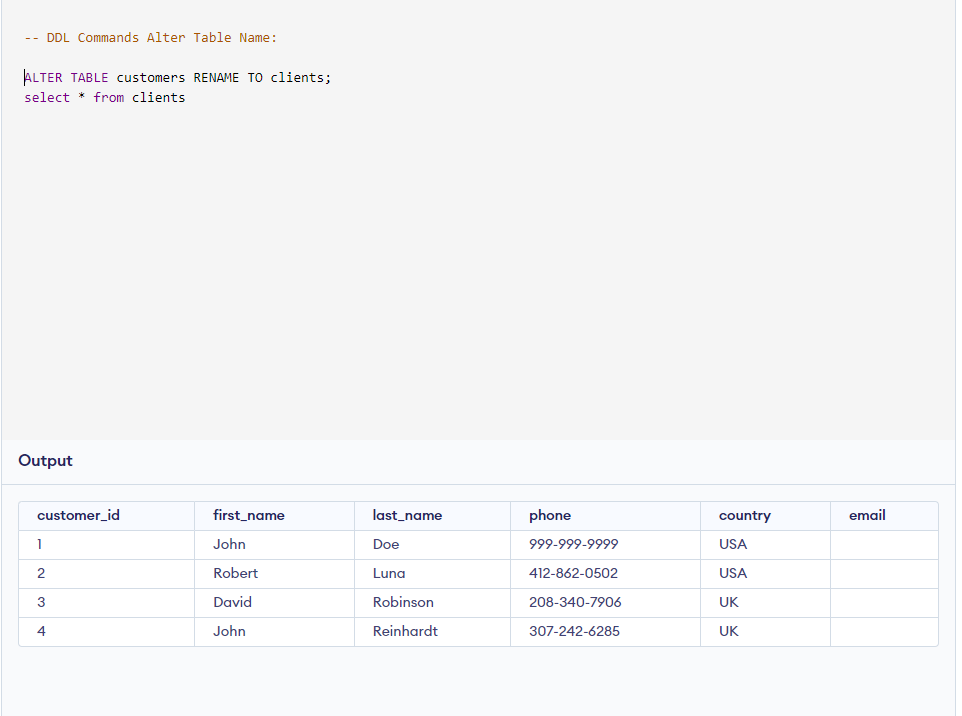
--Create customer table and insert values

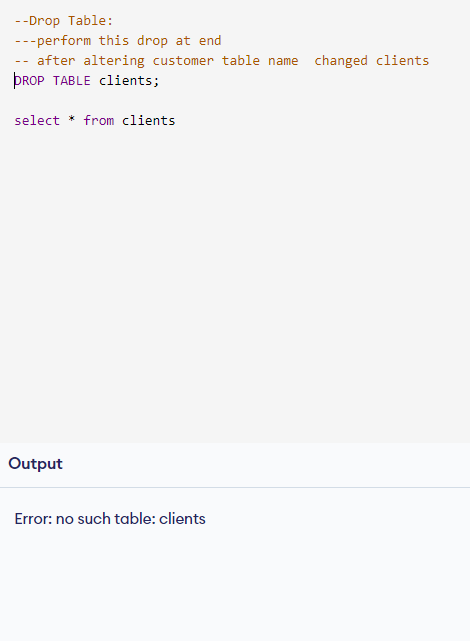






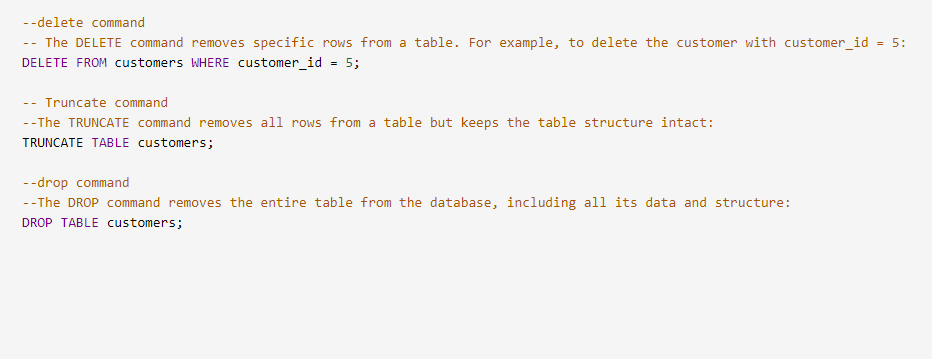






**13. Perform delete, drop and truncate commands on given customers table.**





**14. Explain Purpose of Normalization and issues in row level and column level with suitable example.**

Purpose of Normalization

Normalization is a database design process that organizes data to reduce redundancy and improve data integrity. The main objectives of normalization are to:

- Minimize Data Redundancy: Ensures that the same piece of data is not stored in multiple places, reducing storage space and maintenance.

- Enhance Data Integrity: Maintains consistent and accurate data across the database.

- Optimize Query Performance: Simplifies and speeds up database queries by reducing the number of joins required.

Normalization involves decomposing a table into smaller, related tables and defining relationships between them.

Normal Forms

There are several normal forms, each addressing a specific type of redundancy and dependency:

1. First Normal Form (1NF): Ensures that the table has atomic (indivisible) values and each record is unique.

2. Second Normal Form (2NF): Builds on 1NF, ensuring that all non-key attributes are fully functionally dependent on the primary key.

3. Third Normal Form (3NF): Ensures that all the attributes are not only fully functionally dependent on the primary key but also non-transitively dependent.

Issues at Row Level and Column Level

# Row-Level Issues:

1. Insertion Anomalies: Problems that occur when trying to insert data into a table.

- Example: Consider a table where each row contains information about students and the courses they enroll in. If a new course is introduced but no students are enrolled yet, we cannot insert the course details without a student record.

2. Deletion Anomalies: Issues arising when trying to delete data.

- Example: If a student drops the last course they are enrolled in, deleting the student’s course record would also delete their details from the table, losing important student information.

3. Update Anomalies: Difficulties encountered when updating data in a table.

- Example: If a student's address changes, we need to update multiple rows. Missing one row would result in inconsistent data.

# Column-Level Issues:

1. Redundancy: Storing the same piece of data in multiple columns.

- Example: In a table with customer information, storing the same customer's address in multiple columns if they have multiple accounts.

2. Partial Dependency: When an attribute depends only on part of a composite primary key.

- Example: In a table where the composite key is (StudentID, CourseID), having the student’s name depend only on StudentID.

3. Transitive Dependency: When non-key attributes depend on other non-key attributes.

- Example: In a table with student information, if the student’s department depends on the student’s major, which in turn depends on the StudentID.

Example of Normalization

Consider a table with student information that is not normalized:

# Unnormalized Table

| StudentID | StudentName | CourseID | CourseName | Instructor |

|-----------|-------------|----------|------------|--------------|

| 1 | Alice | 101 | Math | Prof. Smith |

| 2 | Bob | 102 | Science | Prof. Johnson|

| 1 | Alice | 102 | Science | Prof. Johnson|

# Normalized Tables

## Student Table (1NF)

| StudentID | StudentName |

|-----------|-------------|

| 1 | Alice |

| 2 | Bob |

## Course Table (1NF)

| CourseID | CourseName |

|----------|------------|

| 101 | Math |

| 102 | Science |

## Enrollment Table (2NF)

| StudentID | CourseID | Instructor |

|-----------|----------|---------------|

| 1 | 101 | Prof. Smith |

| 1 | 102 | Prof. Johnson |

| 2 | 102 | Prof. Johnson |

By normalizing the tables, we eliminate redundancy and insertion, deletion, and update anomalies, ensuring data integrity and optimized performance.

**15. Explain Different keys with suitable example.**

In database management, keys are critical for uniquely identifying records and establishing relationships between tables. Here are the main types of keys with suitable examples:

1. Primary Key

- Definition: A primary key is a unique identifier for a record in a table. Each table can have only one primary key, and it cannot contain NULL values.

- Example: In a `Students` table, the `StudentID` could be the primary key.

CREATE TABLE Students (

StudentID INT PRIMARY KEY,

StudentName VARCHAR(100),

Age INT

);

2. Foreign Key

- Definition: A foreign key is a column or set of columns in one table that refers to the primary key in another table. It establishes a relationship between the two tables.

- Example: In an `Enrollments` table, the `StudentID` is a foreign key that references the `StudentID` in the `Students` table.

CREATE TABLE Enrollments (

EnrollmentID INT PRIMARY KEY,

StudentID INT,

CourseID INT,

FOREIGN KEY (StudentID) REFERENCES Students(StudentID)

);

3. Unique Key

- Definition: A unique key ensures that all values in a column or set of columns are unique across the table. Unlike the primary key, a table can have multiple unique keys, and they can contain NULL values.

- Example: In a `Users` table, the `Email` could be a unique key to ensure no two users have the same email address.

CREATE TABLE Users (

UserID INT PRIMARY KEY,

UserName VARCHAR(100),

Email VARCHAR(100) UNIQUE

);

4. Candidate Key

- Definition: A candidate key is a column or set of columns that can uniquely identify any record in a table. Each table can have multiple candidate keys, one of which is chosen as the primary key.

- Example: In a `Books` table, both `ISBN` and `BookID` could be candidate keys because they both uniquely identify a book.

CREATE TABLE Books (

BookID INT,

ISBN VARCHAR(13),

Title VARCHAR(200),

PRIMARY KEY (BookID)

);

5. Composite Key

- Definition: A composite key is a primary key composed of two or more columns to uniquely identify a record. It is useful when a single column is not sufficient for unique identification.

- Example: In a `StudentCourses` table, the combination of `StudentID` and `CourseID` could be a composite key to uniquely identify each enrollment.

CREATE TABLE StudentCourses (

StudentID INT,

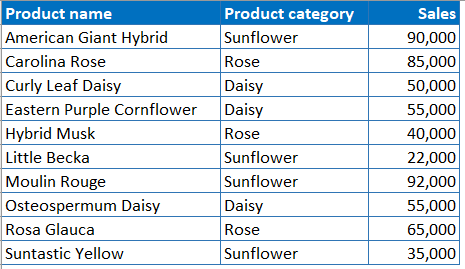
CourseID INT,

EnrollmentDate DATE,

PRIMARY KEY (StudentID, CourseID)

);

**16 .Perform Aggregation functions in SQL for below given table.**



-- create products table

CREATE TABLE Products (

product\_name VARCHAR(255),

product\_category VARCHAR(255),

sales INT

);

---insert values

INSERT INTO Products (product\_name, product\_category, sales) VALUES

('American Giant Hybrid', 'Sunflower', 90000),

('Carolina Rose', 'Rose', 85000),

('Curly Leaf Daisy', 'Daisy', 50000),

('Eastern Purple Cornflower', 'Daisy', 55000),

('Hybrid Musk', 'Rose', 40000),

('Little Becka', 'Sunflower', 22000),

('Moulin Rouge', 'Sunflower', 92000),

('Osteospermum Daisy', 'Daisy', 55000),

('Rosa Glauca', 'Rose', 65000),

('Suntastic Yellow', 'Sunflower', 35000);

---

--Aggregation Functions

--Total Sales for Each Category:

SELECT product\_category, SUM(sales) AS total\_sales

FROM Products

GROUP BY product\_category;

--Average Sales for Each Category:

SELECT product\_category, AVG(sales) AS average\_sales

FROM Products

GROUP BY product\_category;

--Maximum Sales for Each Category:

SELECT product\_category, MAX(sales) AS max\_sales

FROM Products

GROUP BY product\_category;

--Minimum Sales for Each Category:

SELECT product\_category, MIN(sales) AS min\_sales

FROM Products

GROUP BY product\_category;

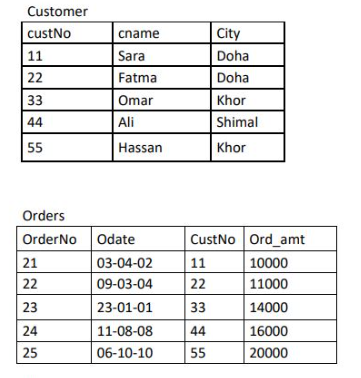
--Count of Products in Each Category:

SELECT product\_category, COUNT(\*) AS product\_count

FROM Products

GROUP BY product\_category;

**17. Perform Joins (inner, outer, left, right, cross and self)on customer and orders table below?**



--Creating the Customer Table

CREATE TABLE Customer (

custNo INT,

cname VARCHAR(255),

City VARCHAR(255)

);

--Insert values in customer table

INSERT INTO Customer (custNo, cname, City) VALUES

(11, 'Sara', 'Doha'),

(22, 'Fatma', 'Doha'),

(33, 'Omar', 'Khor'),

(44, 'Ali', 'Shimal'),

(55, 'Hassan', 'Khor');

---creating order table

CREATE TABLE Orders (

OrderNo INT,

Odate DATE,

CustNo INT,

Ord\_amt DECIMAL(10, 2)

);

--Insert values in orders table

INSERT INTO Orders (OrderNo, Odate, CustNo, Ord\_amt) VALUES

(21, '2002-04-03', 11, 10000),

(22, '2004-03-09', 22, 11000),

(23, '2001-01-23', 33, 14000),

(24, '2008-08-11', 44, 16000),

(25, '2010-10-06', 55, 20000);

--different types of SQL joins using the Customer and Orders tables:

--Inner Join

--An inner join returns only the rows that have matching values in both tables.

SELECT Customer.custNo, Customer.cname, Orders.OrderNo, Orders.Odate, Orders.Ord\_amt

FROM Customer

INNER JOIN Orders ON Customer.custNo = Orders.CustNo;

--Left Join (or Left Outer Join)

--A left join returns all rows from the left table (Customer), and the matched rows from the right table (Orders).

If there is no match, the result is NULL on the right side.

SELECT Customer.custNo, Customer.cname, Orders.OrderNo, Orders.Odate, Orders.Ord\_amt

FROM Customer

LEFT JOIN Orders ON Customer.custNo = Orders.CustNo;

--Right Join (or Right Outer Join)

--A right join returns all rows from the right table (Orders), and the matched rows from the left table (Customer).

If there is no match, the result is NULL on the left side.

SELECT Customer.custNo, Customer.cname, Orders.OrderNo, Orders.Odate, Orders.Ord\_amt

FROM Customer

RIGHT JOIN Orders ON Customer.custNo = Orders.CustNo;

--Full Outer Join

--A full outer join returns all rows when there is a match in either left (Customer) or right (Orders) table.

If there is no match, the result is NULL from the side where there is no match.

SELECT Customer.custNo, Customer.cname, Orders.OrderNo, Orders.Odate, Orders.Ord\_amt

FROM Customer

FULL OUTER JOIN Orders ON Customer.custNo = Orders.CustNo;

--Cross Join

--A cross join returns the Cartesian product of the two tables, meaning it returns all possible combinations of rows from the two tables.

SELECT Customer.custNo, Customer.cname, Orders.OrderNo, Orders.Odate, Orders.Ord\_amt

FROM Customer

CROSS JOIN Orders;

--Self Join

--A self join is a regular join but the table is joined with itself. This can be useful for comparing rows within the same table.

SELECT A.custNo AS CustNo\_A, A.cname AS CustName\_A, B.custNo AS CustNo\_B, B.cname AS CustName\_B

FROM Customer A, Customer B

WHERE A.City = B.City AND A.custNo <> B.custNo;

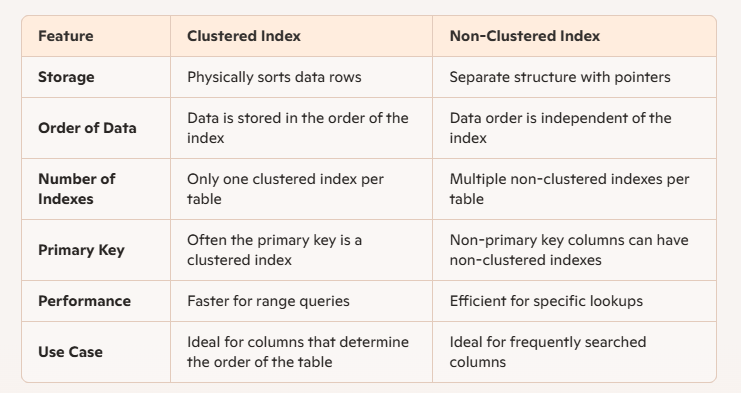
**18. Explain the view and create the view on customer table.**

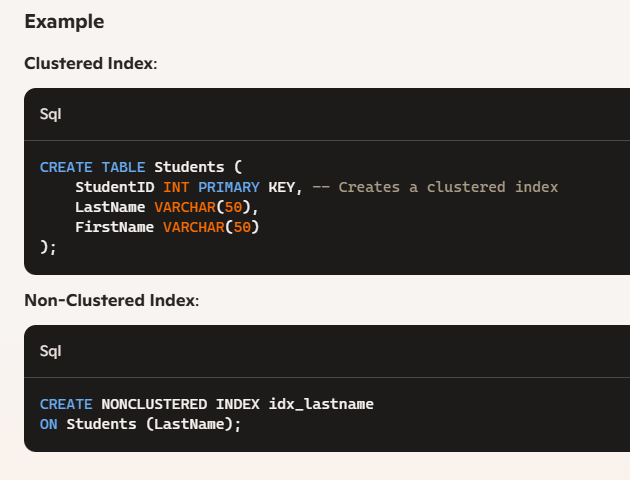




==================================================================================

**19. Differentiate between** [**Clustered**](https://www.sqltutorial.net/clustered-index.html)  **Indexes and non-**[**Clustered**](https://www.sqltutorial.net/clustered-index.html)  **Indexes in SQL.**





**20. Write short note on**

**(i) stored procedure (ii) Triggers(iii) Cursors .**

(i) Stored Procedure

Definition: A stored procedure is a set of SQL statements that can be stored in the database and executed as a single unit. Stored procedures are used to encapsulate logic, promote code reuse, and enhance performance.

Key Features:

- Reusable: Once created, it can be executed multiple times without rewriting the SQL code.

- Modular: Encapsulates complex SQL operations into a single callable procedure.

- Security: Can help enforce business rules and security through encapsulation.

- Performance: Reduces client-server communication overhead and can improve performance by pre-compiling SQL code.

Example:

CREATE PROCEDURE GetCustomerOrders @CustomerId INT

AS

BEGIN

SELECT \* FROM Orders WHERE CustomerId = @CustomerId;

END

This example creates a stored procedure to get all orders for a given customer ID.

(ii) Triggers

Definition: Triggers are special types of stored procedures that automatically execute in response to certain events on a particular table or view, such as `INSERT`, `UPDATE`, or `DELETE`.

Key Features:

- Automation: Automatically triggers in response to specified database events.

- Data Integrity: Enforces business rules and ensures data consistency.

- Audit: Tracks changes to data, useful for logging and auditing purposes.

- Complex Actions: Can initiate complex actions based on changes in data.

Example:

CREATE TRIGGER trgAfterInsert

ON Employees

AFTER INSERT

AS

BEGIN

INSERT INTO AuditLog (EventDescription, EventDate)

VALUES ('New employee added', GETDATE());

END

This example creates a trigger that inserts a log entry into an `AuditLog` table whenever a new employee is added.

(iii) Cursors

Definition: Cursors are database objects used to retrieve, manipulate, and traverse through a set of rows returned by a query. They are often used in scenarios where row-by-row processing is needed.

Key Features:

- Row-by-Row Processing: Allows processing individual rows one at a time, which is useful for complex data manipulation.

- Control Flow: Provides better control over the retrieval of result sets.

- Memory Intensive: Can be resource-intensive, so should be used judiciously.

- Iterative Operations: Enables iterative operations on each row of the result set.

Example:

DECLARE @EmployeeId INT

DECLARE cursorEmployee CURSOR FOR

SELECT EmployeeId FROM Employees

OPEN cursorEmployee

FETCH NEXT FROM cursorEmployee INTO @EmployeeId

WHILE @@FETCH\_STATUS = 0

BEGIN

PRINT 'Processing Employee ID: ' + CAST(@EmployeeId AS VARCHAR)

FETCH NEXT FROM cursorEmployee INTO @EmployeeId

END

CLOSE cursorEmployee

DEALLOCATE cursorEmployee

This example declares a cursor to iterate through each `EmployeeId` in the `Employees` table, printing the ID of each employee as it processes.

Each of these SQL components plays a unique role in managing data and automating tasks within a database, enhancing the overall efficiency and maintainability of database operations