

EX NO: 1	OBJECT-ORIENTED PROGRAMMING CONCEPTS
DATE: 21/01/25	

AIM:

To write Python programs that demonstrate the core concepts of Object-Oriented Programming (OOP) — **Abstraction, Encapsulation, Inheritance, and Polymorphism.**

ALGORITHM:

- **Abstraction:**
 - Use the abc module to create an abstract class with an abstract method.
 - Inherit the class and implement the method in the derived class.
- **Encapsulation:**
 - Create a class with private data members.
 - Provide getter and setter methods to access and modify private data securely.
- **Inheritance:**
 - Define a base class with common methods and attributes.
 - Inherit the base class into a child class and add more functionality.
- **Polymorphism:**
 - Create multiple classes with methods of the same name.
 - Demonstrate polymorphism using dynamic method calling (duck typing).

ABSTRACTION PROGRAM:

```
from abc import ABC, abstractmethod
```

```
class Shape(ABC):
```

```
@abstractmethod
def area(self):
    pass

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius * self.radius

circle = Circle(5)
print("Area of Circle:", circle.area())
```

OUTPUT:

Area of Circle: 78.5

ENCAPSULATION PROGRAM:

```
class Student:
    def __init__(self, name, marks):
        self.name = name
        self.__marks = marks

    def get_marks(self):
        return self.__marks

    def set_marks(self, marks):
        if 0 <= marks <= 100:
            self.__marks = marks
        else:
            print("Invalid marks")

s1 = Student("Rahul", 85)
print("Name:", s1.name)
print("Marks:", s1.get_marks())

s1.set_marks(92)
print("Updated Marks:", s1.get_marks())
```

OUTPUT:

Name: Rahul

Marks: 85

Updated Marks: 92

INHERITANCE PROGRAM:

```
class Person:
    def __init__(self, name):
        self.name = name

    def show(self):
        print("Name:", self.name)

class Student(Person):
    def __init__(self, name, roll):
        super().__init__(name)
        self.roll = roll

    def display(self):
        self.show()
        print("Roll No:", self.roll)

s = Student("Aarav", 101)
s.display()
```

OUTPUT:

Name: Aarav

Roll No: 101

POLYMORPHISM PROGRAM:

```
class Cat:
    def sound(self):
        print("Cat says Meow")

class Dog:
    def sound(self):
        print("Dog says Bark")

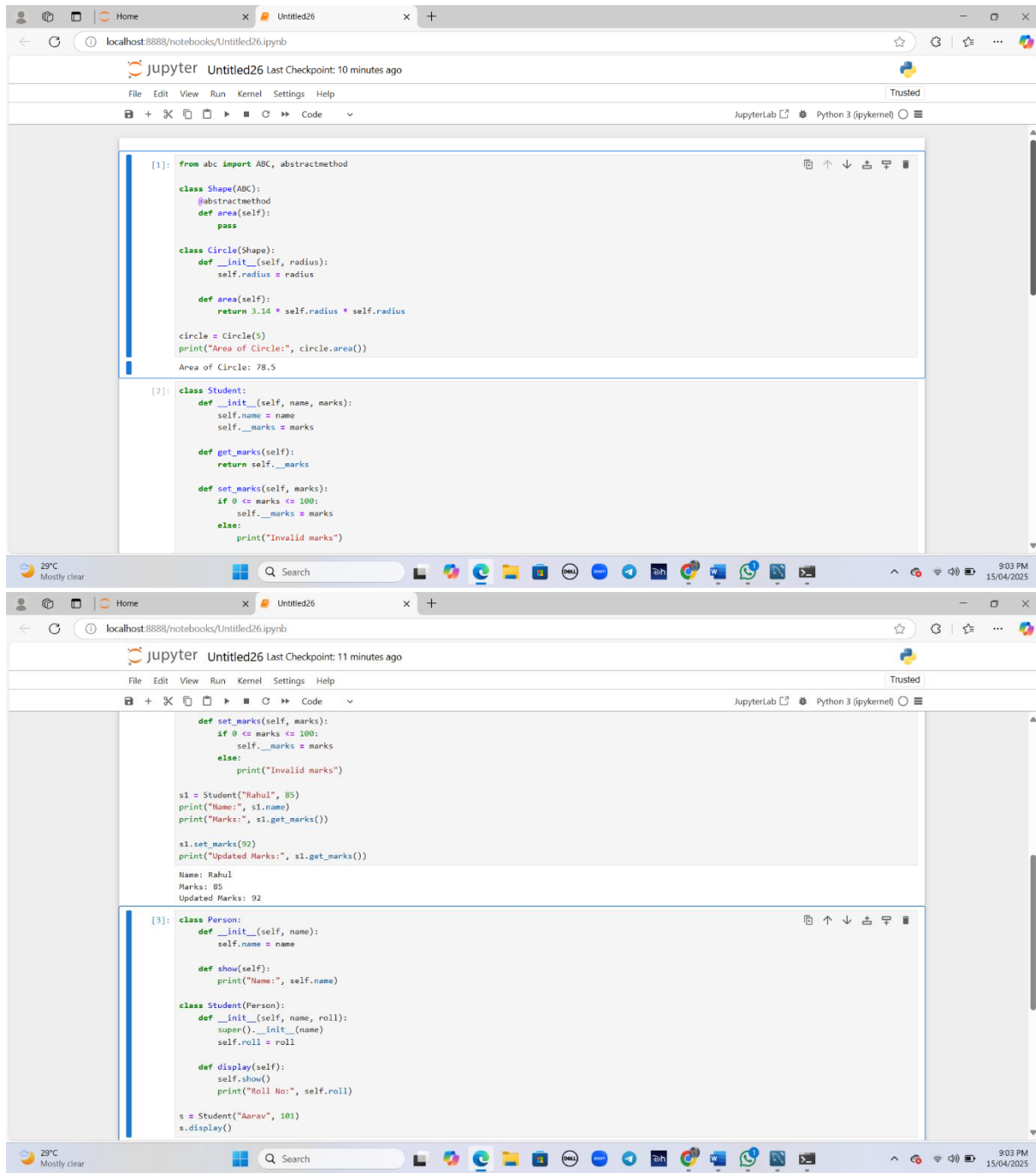
for animal in (Cat(), Dog()):
    animal.sound()
```

OUTPUT:

Cat says Meow

Dog says Bark

SCREENSHOT:



The image displays two screenshots of a Jupyter Notebook interface, likely from a web browser. The top screenshot shows the first two code cells. The bottom screenshot shows the continuation of the code, including the execution of the second cell and the definition of a new class.

Cell [1]:

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius * self.radius

circle = Circle(5)
print("Area of Circle:", circle.area())
```

Area of Circle: 78.5

Cell [2]:

```
class Student:
    def __init__(self, name, marks):
        self.name = name
        self._marks = marks

    def get_marks(self):
        return self._marks

    def set_marks(self, marks):
        if 0 <= marks <= 100:
            self._marks = marks
        else:
            print("Invalid marks")
```

Cell [3]:

```
def set_marks(self, marks):
    if 0 <= marks <= 100:
        self._marks = marks
    else:
        print("Invalid marks")

s1 = Student("Rahul", 85)
print("Name:", s1.name)
print("Marks:", s1.get_marks())

s1.set_marks(92)
print("Updated Marks:", s1.get_marks())
```

Name: Rahul
Marks: 85
Updated Marks: 92

Cell [3]:

```
class Person:
    def __init__(self, name):
        self.name = name

    def show(self):
        print("Name:", self.name)

class Student(Person):
    def __init__(self, name, roll):
        super().__init__(name)
        self.roll = roll

    def display(self):
        self.show()
        print("Roll No:", self.roll)

s = Student("Aarav", 101)
s.display()
```

The image displays two screenshots of a Jupyter Notebook interface, showing the execution of Python code related to Object-Oriented Programming (OOPs) concepts.

Top Screenshot:

```
[3]: class Person:
      def __init__(self, name):
          self.name = name

      def show(self):
          print("Name:", self.name)

      class Student(Person):
          def __init__(self, name, roll):
              super().__init__(name)
              self.roll = roll

          def display(self):
              self.show()
              print("Roll No:", self.roll)

      s = Student("Aarav", 101)
      s.display()

      Name: Aarav
      Roll No: 101
```

Bottom Screenshot:

```
[4]: class Cat:
      def sound(self):
          print("Cat says Meow")

      class Dog:
          def sound(self):
              print("Dog says Bark")

      for animal in (Cat(), Dog()):
          animal.sound()

      Cat says Meow
      Dog says Bark
```

RESULT:

Thus, the OOPs Concepts has been executed successfully.

EX.NO:2

DATE: 28.01.25	WORKING WITH API'S – REQUEST LIBRARY
---------------------------------	---

AIM:

To use the requests library in Python to perform GET and POST operations on a API.

ALGORITHM:

STEP 1: Import Required Libraries

- Import the requests library to handle HTTP requests.
- Import json to format the response data.

STEP 2: Perform a GET Request

- Use requests.get() to fetch data from <https://jsonplaceholder.typicode.com/posts>.
- Check if the response status code is 200 (OK).
- If successful, convert the JSON response to a Python object.
- Use json.dumps() to print the data in a pretty-printed format.

STEP 3: Prepare Data for POST Request

- Create a Python dictionary named new_data with keys: title, body, and userId.

STEP 4: Perform a POST Request

- Use requests.post() with the json=new_data parameter to send the data to the same API.
- Print the status code of the response.
- Pretty-print the response JSON using json.dumps().

IMPLEMENTATION:

```
import requests  
import json
```

```
# GET request  
response = requests.get('https://jsonplaceholder.typicode.com/posts')
```

```
if response.status_code == 200:
    data = response.json()
    print("GET Response:")
    print(json.dumps(data, indent=4)) # Pretty-print the JSON

# POST request
new_data = {
    "title": "New Post",
    "body": "This is a new post",
    "userId": 1
}

response = requests.post('https://jsonplaceholder.typicode.com/posts', json=new_data)
print("\nPOST Response:")
print("Status Code:", response.status_code)
print("Response JSON:")
print(json.dumps(response.json(), indent=4)) # Pretty-print the response JSON
```

OUTPUT:

GET Response:

Squeezed text (785 lines).

POST Response:

Status Code: 201

Response JSON:

```
{
  "title": "New Post",
  "body": "This is a new post",
  "userId": 1,
  "id": 101
}
>>>
```

RESULT:

Thus, the GET request successfully retrieved data from the API, and the POST request added a new post, returning a confirmation response.

EX.NO: 03	DATABASE PROGRAMMING IN PL/SQL PROCEDURES AND FUNCTIONS, TRIGGERS, CURSORS
DATE: 24/01/25	

AIM:

To debug a Python function using breakpoints with the pdb module and identify any logical errors.

A) pdb module

ALGORITHM:

Step 1: Start

Step 2: Define a function calculate_square(x)

Step 3: Inside the function, set a breakpoint using pdb.set_trace() to pause execution

Step 4: Return the square of x using the expression x ** 2

Step 5: Call the function with a sample input like calculate_square(4)

Step 6: When execution pauses, use the following pdb commands:

- **Step 6.1:** n – Move to the next line in the current function
- **Step 6.2:** s – Step into a function call
- **Step 6.3:** p x – Print the value of variable x
- **Step 6.4:** c – Continue execution of the program

Step 7: End

B) Assert Statement

ALGORITHM:

Step 1: Start

Step 2: Define a function check_even_number(num)

Step 3: Check if num is an integer

Step 3.1: If not, raise a ValueError with the message: "Input must be an integer"

Step 4: Use assert to check if num % 2 == 0

Step 4.1: If **True**, return True

Step 4.2: If **False**, raise an AssertionError with the message: "The number is not even"

Step 5: Call the function with sample inputs:

- check_even_number(4) ✓ (Even number — returns True)
- check_even_number(5) ✗ (Odd number — raises AssertionError)
- check_even_number("hello") ✗ (Invalid input — raises ValueError)

Step 6: End

PROGRAM:

```
def calculate_square(x):
    pdb.set_trace() # Debugging breakpoint
    return x ** 2

print("Square Calculation with Debugging:")
print(calculate_square(4)) # Try with a bug to debug manually

# Part 2: Using assert statements
def check_even_number(num):
    if not isinstance(num, int):
        raise ValueError("Input must be an integer")
```



```
assert num % 2 == 0, "The number is not even"  
return True
```

```
print("\nEven Number Check with Assertions:")  
print(check_even_number(4)) # Should return True  
print(check_even_number(5)) # Should raise AssertionError  
print(check_even_number("hello")) # Should raise ValueError
```

OUTPUT:

```
Square Calculation with Debugging:  
> c:\users\united-6789876545\downloads\pdb and assert.py(6)calculate_square()  
-> return x ** 2  
(Pdb)
```

RESULT:

Thus, Python function using breakpoints with the pdb module and assert statement has been implemented successfully.

EX. NO: 4	SQL - CRUD OPERATIONS
DATE: 18/02/25	

AIM:

To perform Create, Read, Update, and Delete (CRUD) operations on a MySQL database table using SQL queries.

ALGORITHM:

- Create Database and Table for storing customer data.
- Insert records into the table using INSERT query.
- Read and view data using SELECT query.
- Update specific records using UPDATE query.
- Delete specific records using DELETE query.
- Display the final table contents using SELECT.

CRUD OPERATIONS:

STEP 1: CREATE A NEW DATABASE

```
CREATE DATABASE SchoolDB;
```

STEP 2: USE THE DATABASE

```
USE SchoolDB;
```

STEP 3: CREATE A NEW TABLE

```
CREATE TABLE Students (  
    StudentID INT PRIMARY KEY,  
    Name VARCHAR(50),  
    Email VARCHAR(100),  
    Department VARCHAR(50)  
);
```

✓	2	20:19:35	CREATE DATABASE School	1 row(s) affected
✓	3	20:19:50	USE School	0 row(s) affected
✓	4	20:20:00	CREATE TABLE Students (-- Creates a table named 'Students' StudentID INT PRIMARY ...	0 row(s) affected

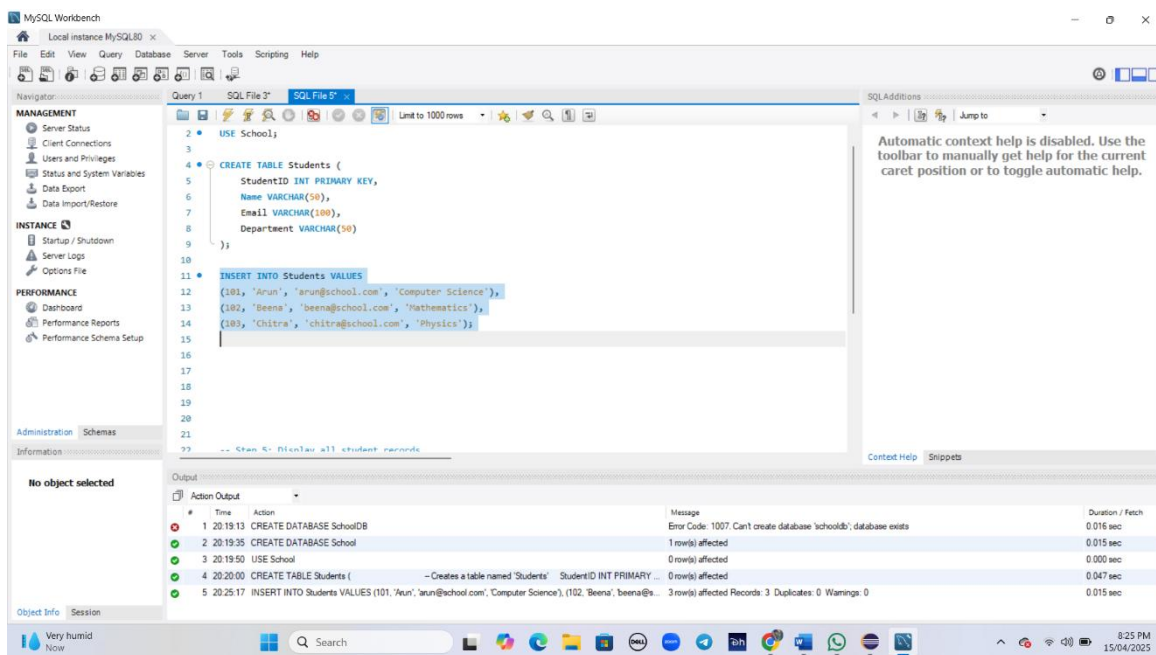
STEP 4: INSERT RECORDS INTO THE TABLE

INSERT INTO Students VALUES

(101, 'Arun', 'arun@school.com', 'Computer Science'),

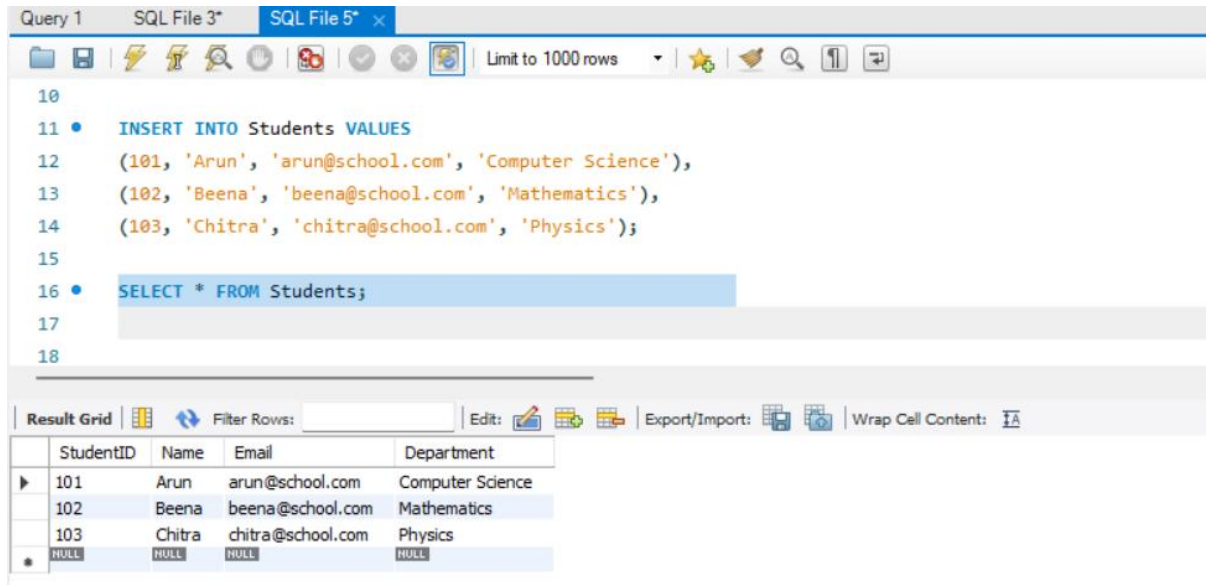
(102, 'Beena', 'beena@school.com', 'Mathematics'),

(103, 'Chitra', 'chitra@school.com', 'Physics');



STEP 5: DISPLAY ALL STUDENT RECORDS

SELECT * FROM Students;



STEP 6: UPDATE DEPARTMENT OF A STUDENT

UPDATE Students

SET Department = 'Statistics'

WHERE StudentID = 102;

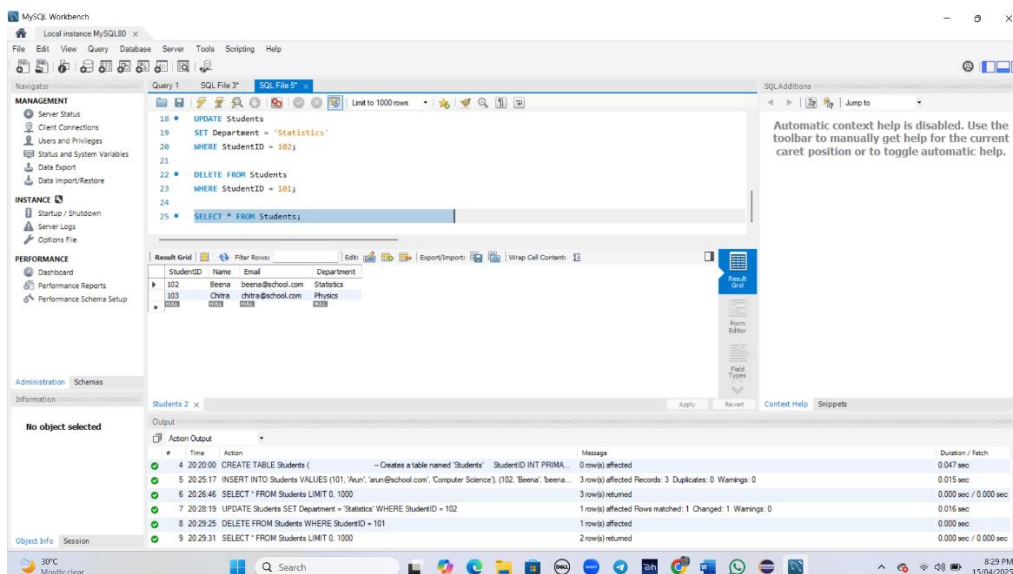
STEP 7: DELETE A STUDENT RECORD

DELETE FROM Students

WHERE StudentID = 101;

STEP 8: FINAL VIEW OF THE TABLE

SELECT * FROM Students;



RESULT:

Thus, the SQL CRUD Operations has been executed successfully.

EX. NO:5	VERSION CONTROL WITH GIT
DATE: 25/02/25	

AIM:

To understand and implement basic version control operations using Git such as repository creation, staging, committing, branching, merging, and collaboration via GitHub.

ALGORITHM:

STEP 1: Install Git and configure user identity.

STEP 2: Initialize a local Git repository.

STEP 3: Add and commit files to the repository.

STEP 4: Create and switch between branches. STEP 5: Merge branches and resolve conflicts. STEP 6: Push the local repository to GitHub.

STEP 7: Clone remote repositories and pull changes.

PROGRAM:

```
git --version
git config --global user.name "AjayKumarV"
git config --global user.email "ajaykumar2k@gmail.com.com" mkdir git-demo
cd git-demo git init
git status
git add hello.py
git commit -m "Initial commit: Added hello.py"

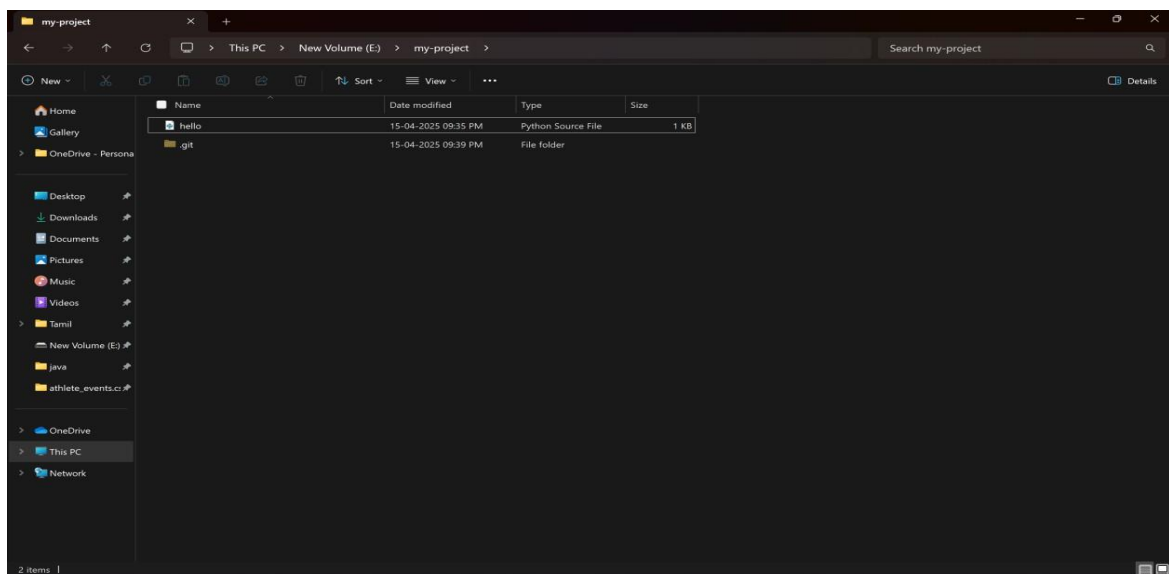
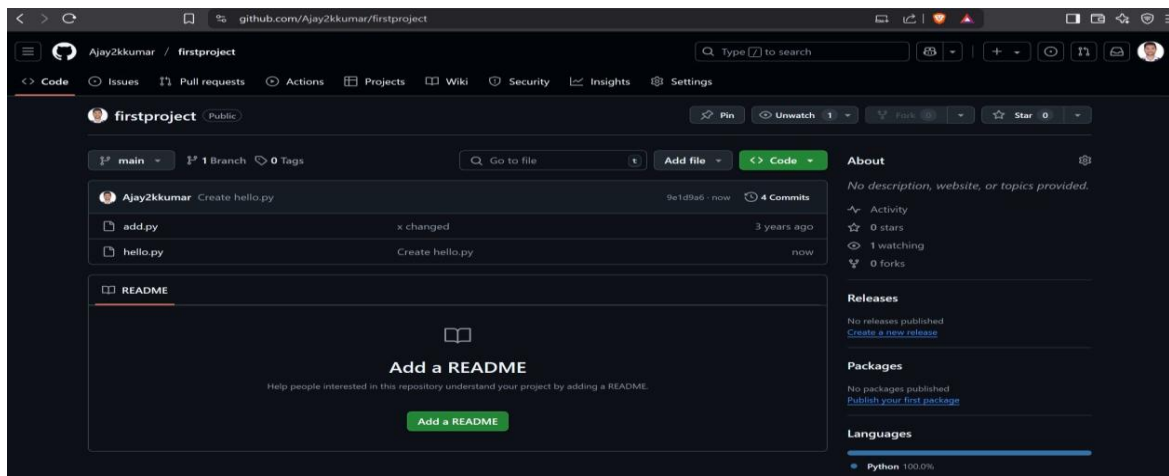
git branch feature
git checkout feature

git add hello.py
git commit -m "Updated hello.py in feature branch"

git checkout main git merge feature

git remote add origin https://github.com/ajaykumarv/git-demo.git git push -u origin main
cd ..
git clone https://github.com/ajaykumarv/git-demo.git cd git-demo
git pull origin main
```

OUTPUT:



Initialized empty Git repository in /path/to/git-demo/.git/ [main (root-commit) Initial commit:
Added hello.py] Switched to branch 'feature'

[feature Updated hello.py in feature branch] Switched to branch 'main'

Merging feature branch...

Merge made by the 'recursive' strategy. Pushing to GitHub repository...

To <https://github.com/ajaykumarv/git-demo.git>

* [new branch] main -> main Cloning into 'git-demo'...

RESULT:

Thus, the version control operations using Git and GitHub have been implemented and verified successfully.

EX. NO:6	SIMPLE PYTHON CALCULATOR PROGRAM WITH GIT AND GITHUB TO MANAGE VERSION CONTROL
DATE: 04/03/25	

AIM:

To create a basic calculator (Version 1) and an advanced calculator (Version 2) in Python, and manage their versions using Git and GitHub.

ALGORITHM:

Version 1: Normal Calculator

1. Display menu: Add, Subtract, Multiply, Divide.
2. Ask user to choose an operation.
3. Get two numbers from user.
4. Perform the selected operation.
5. Display result.
6. Repeat until user exits.

Version 2: Advanced Calculator

1. Inherit operations from the normal calculator.
2. Add more options: Square and Square Root.
3. Display all operations (1–6).
4. Get user input and execute corresponding method.
5. Loop until exit.
6. `git add .` stages all modified files.
7. `git commit -m "message"` commits the changes with a message, generating a unique commit hash.
8. `git checkout <commit-hash>` allows you to switch to a previous version of the repository by specifying the commit hash.
9. You can view your commit history on GitHub, which displays the commit hashes and messages for reference.

PROGRAM:

1. calc.py (Version 1 - Normal Calculator)

```
class Calculator:
```

```
    def add(self, a, b):
```

```
        return a + b
```

```
def subtract(self, a, b):
    return a - b

def multiply(self, a, b):
    return a * b

def divide(self, a, b):
    if b != 0:
        return a / b
    else:
        return "Cannot divide by zero!"

calc = Calculator()

while True:
    print("\n1. Add\n2. Subtract\n3. Multiply\n4. Divide\n5. Exit")
    choice = input("Choose operation (1-5): ")

    if choice == '5':
        print("Exiting calculator...")
        break

    num1 = float(input("Enter first number: "))
    num2 = float(input("Enter second number: "))

    if choice == '1':
        print("Result:", calc.add(num1, num2))
    elif choice == '2':
        print("Result:", calc.subtract(num1, num2))
    elif choice == '3':
        print("Result:", calc.multiply(num1, num2))
    elif choice == '4':
        print("Result:", calc.divide(num1, num2))
```


else:

print("Invalid choice.")

#2. advcalc.py (Version 2 - Advanced Calculator using Inheritance)

import math

class Calculator:

def add(self, a, b): return a + b

def subtract(self, a, b): return a - b

def multiply(self, a, b): return a * b

def divide(self, a, b): return a / b if b != 0 else "Cannot divide by zero"

class AdvancedCalculator(Calculator):

def square(self, a): return a ** 2

def square_root(self, a): return math.sqrt(a)

adv = AdvancedCalculator()

while True:

print("\n1. Add\n2. Subtract\n3. Multiply\n4. Divide\n5. Square\n6. Square Root\n7. Exit")

choice = input("Choose operation (1-7): ")

if choice == '7':

print("Exiting calculator...")

break

if choice in ['5', '6']:

num = float(input("Enter a number: "))

if choice == '5':

print("Result:", adv.square(num))

elif choice == '6':

print("Result:", adv.square_root(num))

else:

```
num1 = float(input("Enter first number: "))
num2 = float(input("Enter second number: "))
if choice == '1':
    print("Result:", adv.add(num1, num2))
elif choice == '2':
    print("Result:", adv.subtract(num1, num2))
elif choice == '3':
    print("Result:", adv.multiply(num1, num2))
elif choice == '4':
    print("Result:", adv.divide(num1, num2))
else:
    print("Invalid choice.")
```

Git Commit Commands:

```
git add .
```

```
git commit -m "Version 1: Normal calculator"
```

```
# Commit hash: abc1234 (example)
```

```
git add .
```

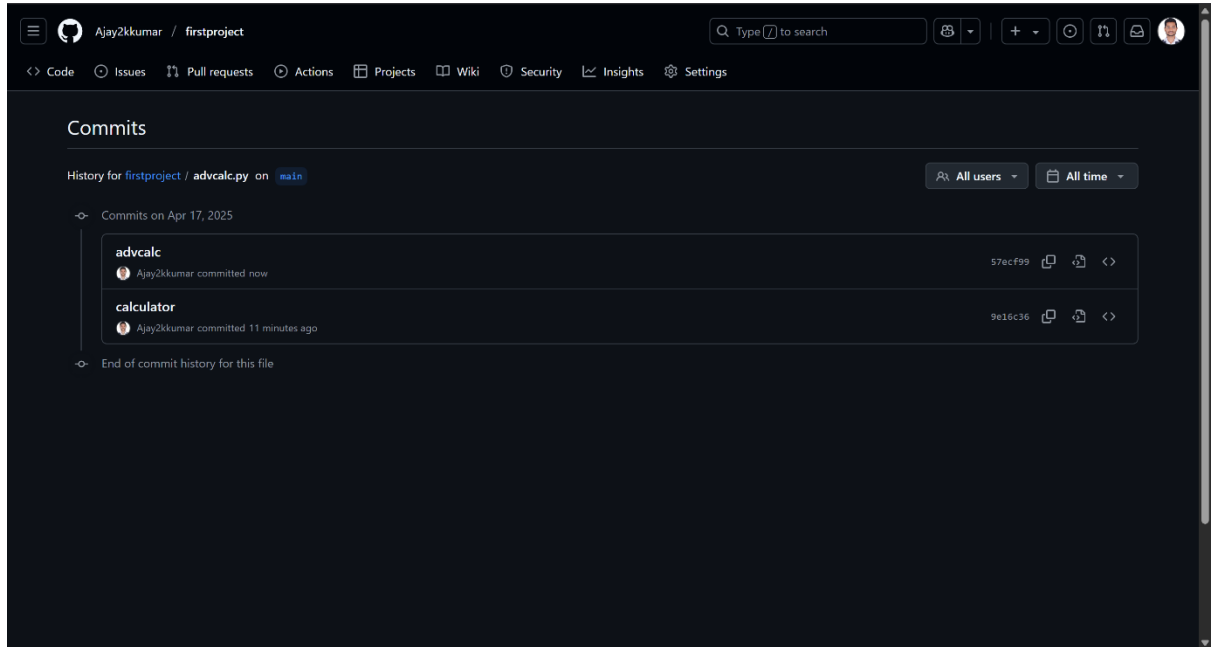
```
git commit -m "Version 2: Advanced calculator with square and root"
```

```
# Commit hash: def5678 (example)
```

```
git checkout <commit-hash> # Use GitHub to view version history:
```

NAME: SANTHOSH S
REG NO: 3122246002011

OUTPUT:



```
Delta compression using up to 16 threads
Compressing objects: 100% (4/4), done.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 1.22 KiB | 1.22 MiB/s, done.
Writing objects: 100% (4/4), 1.22 KiB | 1.22 MiB/s, done.
Total 4 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
To https://github.com/Ajay2kkumar/firstproject
   9e1d9a6..9e16c36  main -> main
PS C:\Users\ajay kumar\Desktop\firstproject> git add .
PS C:\Users\ajay kumar\Desktop\firstproject> git commit
On branch main
Your branch is up to date with 'origin/main'.

nothing to commit, working tree clean
PS C:\Users\ajay kumar\Desktop\firstproject> git add .
PS C:\Users\ajay kumar\Desktop\firstproject> git commit -m "advcalc"
[main 57ecf99] advcalc
   1 file changed, 2 insertions(+), 2 deletions(-)
PS C:\Users\ajay kumar\Desktop\firstproject> git push
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 16 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 317 bytes | 317.00 KiB/s, done.
Total 3 (delta 2), reused 0 (delta 0), pack-reused 0 (from 0)
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To https://github.com/Ajay2kkumar/firstproject
   9e16c36..57ecf99  main -> main
PS C:\Users\ajay kumar\Desktop\firstproject> 
```

```
1. Add
2. Subtract
3. Multiply
4. Divide
5. Square
6. Square Root
Choose operation (1-6): 5
Enter a number: 5
Square: 25.0
PS C:\Users\ajay kumar\Desktop\firstproject>
```

NAME: SANTHOSH S
REG NO: 3122246002011

RESULT:

A basic calculator (Version 1) and an advanced calculator (Version 2) in Python, and manage their versions using Git and GitHub has been created successfully.

NAME: SANTHOSH S
REG NO: 3122246002011

NAME: NAVEEN KUMAR. V
REG NO: 3122246002008

EX.NO:2	WORKING WITH API'S – REQUEST LIBRARY
DATE:28.01.25	

AIM:

To use the requests library in Python to perform GET and POST operations on a API.

ALGORITHM:

STEP 1: Import Required Libraries

- Import the requests library to handle HTTP requests.
- Import json to format the response data.

STEP 2: Perform a GET Request

- Use requests.get() to fetch data from <https://jsonplaceholder.typicode.com/posts>.
- Check if the response status code is 200 (OK).
- If successful, convert the JSON response to a Python object.
- Use json.dumps() to print the data in a pretty-printed format.

STEP 3: Prepare Data for POST Request

- Create a Python dictionary named new_data with keys: title, body, and userId.

STEP 4: Perform a POST Request

- Use requests.post() with the json=new_data parameter to send the data to the same API.
- Print the status code of the response.
- Pretty-print the response JSON using json.dumps().

IMPLEMENTATION:

```
import requests
import json

# GET request
response = requests.get('https://jsonplaceholder.typicode.com/posts')

if response.status_code == 200:
    data = response.json()
    print("GET Response:")
    print(json.dumps(data, indent=4)) # Pretty-print the JSON

# POST request
new_data = {
    "title": "New Post",
    "body": "This is a new post",
    "userId": 1
}

response = requests.post('https://jsonplaceholder.typicode.com/posts', json=new_data)
print("\nPOST Response:")
print("Status Code:", response.status_code)
print("Response JSON:")
print(json.dumps(response.json(), indent=4)) # Pretty-print the response JSON
```

OUTPUT:

GET Response:

Squeezed text (785 lines).

POST Response:

Status Code: 201

Response JSON:

```
{  
  "title": "New Post",  
  "body": "This is a new post",  
  "userId": 1,  
  "id": 101  
}  
>>>
```

RESULT:

Thus, the GET request successfully retrieved data from the API, and the POST request added a new post, returning a confirmation response.

EX.NO : 03	DEVELOPER TOOLS - DEBUGGING AND ASSERT STATEMENTS
DATE : 04.02.25	

AIM:

To debug a Python function using breakpoints with the pdb module and identify any logical errors.

A) PDB MODULE

ALGORITHM:

STEP 1: Start

STEP 2: Define a function calculate_square(x)

STEP 3: Inside the function, set a breakpoint using pdb.set_trace() to pause execution

STEP 4: Return the square of x using the expression x ** 2

STEP 5: Call the function with a sample input like calculate_square(4)

STEP 6: When execution pauses, use the following pdb commands:

- **Step 6.1:** n – Move to the next line in the current function
- **Step 6.2:** s – Step into a function call
- **Step 6.3:** p x – Print the value of variable x
- **Step 6.4:** c – Continue execution of the program

STEP 7: End

B) ASSERT STATEMENT

ALGORITHM:

STEP 1: Start

STEP 2: Define a function check_even_number(num)

STEP 3: Check if num is an integer

Step 3.1: If not, raise a ValueError with the message: "Input must be an integer"

STEP 4: Use assert to check if num % 2 == 0

Step 4.1: If **True**, return True

Step 4.2: If **False**, raise an AssertionError with the message: "The number is not even"

STEP 5: Call the function with sample inputs:

- check_even_number(4) (Even number — returns True)
- check_even_number(5) (Odd number — raises AssertionError)
- check_even_number("hello") (Invalid input — raises ValueError)

STEP 6: End

PROGRAM:

```
def calculate_square(x):
    pdb.set_trace() # Debugging breakpoint
    return x ** 2

print("Square Calculation with Debugging:")
print(calculate_square(4)) # Try with a bug to debug manually

# Part 2: Using assert statements
def check_even_number(num):
    if not isinstance(num, int):
        raise ValueError("Input must be an integer")
    assert num % 2 == 0, "The number is not even"
    return True

print("\nEven Number Check with Assertions:")
print(check_even_number(4)) # Should return True
print(check_even_number(5)) # Should raise AssertionError
print(check_even_number("hello")) # Should raise ValueErro
```

OUTPUT:

```
Square Calculation with Debugging:
> c:\users\united-6789876545\downloads\pdp and assert.py(6)calculate_square()
-> return x ** 2
(Pdb)
```

RESULT:

Thus, Python function using breakpoints with the pdb module and assert statement has been implemented successfully.

EX. NO: 4	SQL - CRUD OPERATIONS
DATE: 18/02/25	

AIM:

To perform Create, Read, Update, and Delete (CRUD) operations on a MySQL database table using SQL queries.

ALGORITHM:

- Create Database and Table for storing customer data.
- Insert records into the table using INSERT query.
- Read and view data using SELECT query.
- Update specific records using UPDATE query.
- Delete specific records using DELETE query.
- Display the final table contents using SELECT.

CRUD OPERATIONS:

STEP 1: CREATE A NEW DATABASE

CREATE DATABASE SchoolDB;

STEP 2: USE THE DATABASE

USE SchoolDB;

STEP 3: CREATE A NEW TABLE

CREATE TABLE Students (
StudentID INT PRIMARY KEY,
Name VARCHAR(50),
Email VARCHAR(100),
Department VARCHAR(50)
);

✓	2	20:19:35	CREATE DATABASE School	1 row(s) affected
✓	3	20:19:50	USE School	0 row(s) affected
✓	4	20:20:00	CREATE TABLE Students (- Creates a table named 'Students' StudentID INT PRIMARY ...	0 row(s) affected

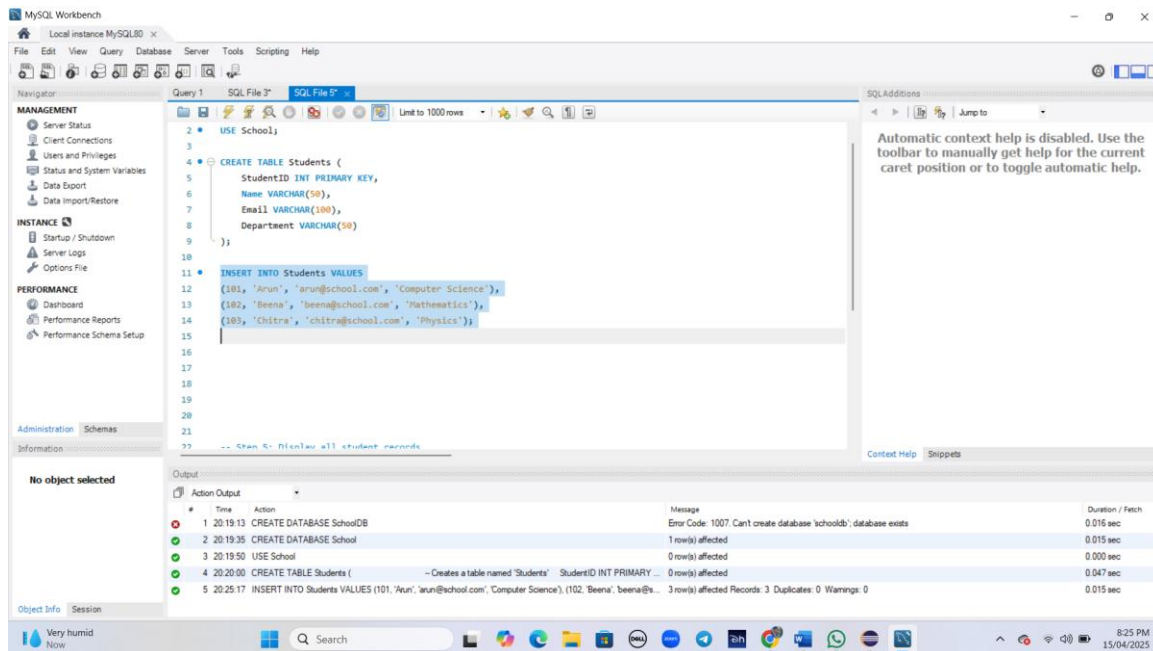
STEP 4: INSERT RECORDS INTO THE TABLE

INSERT INTO Students VALUES

(101, 'Arun', 'arun@school.com', 'Computer Science'),

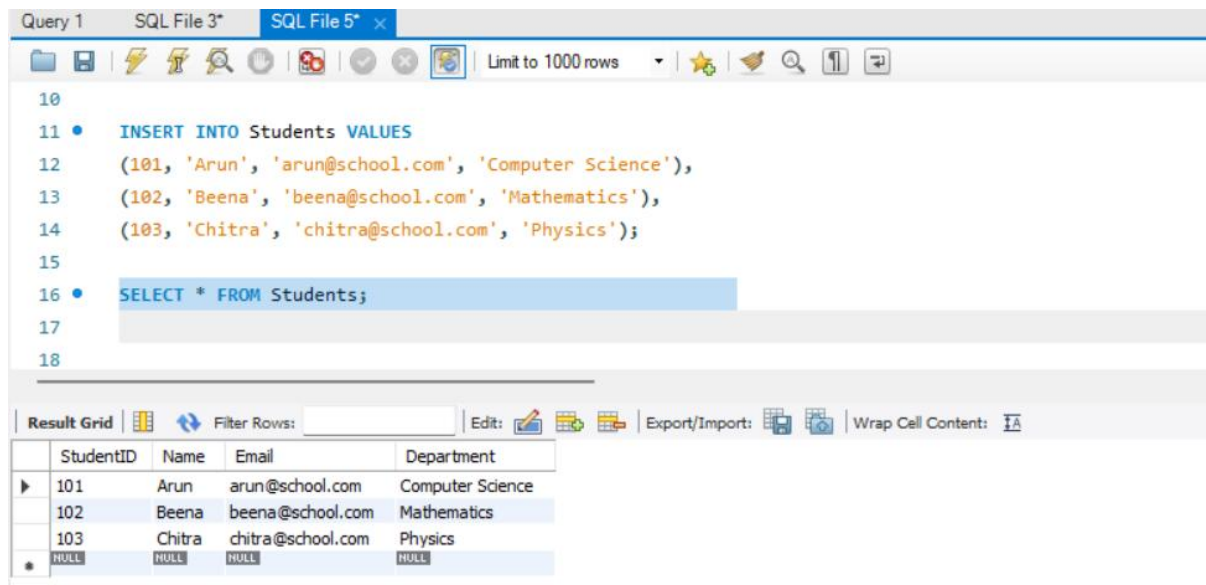
(102, 'Beena', 'beena@school.com', 'Mathematics'),

(103, 'Chitra', 'chitra@school.com', 'Physics');



STEP 5: DISPLAY ALL STUDENT RECORDS

SELECT * FROM Students;



STEP 6: UPDATE DEPARTMENT OF A STUDENT

UPDATE Students

SET Department = 'Statistics'

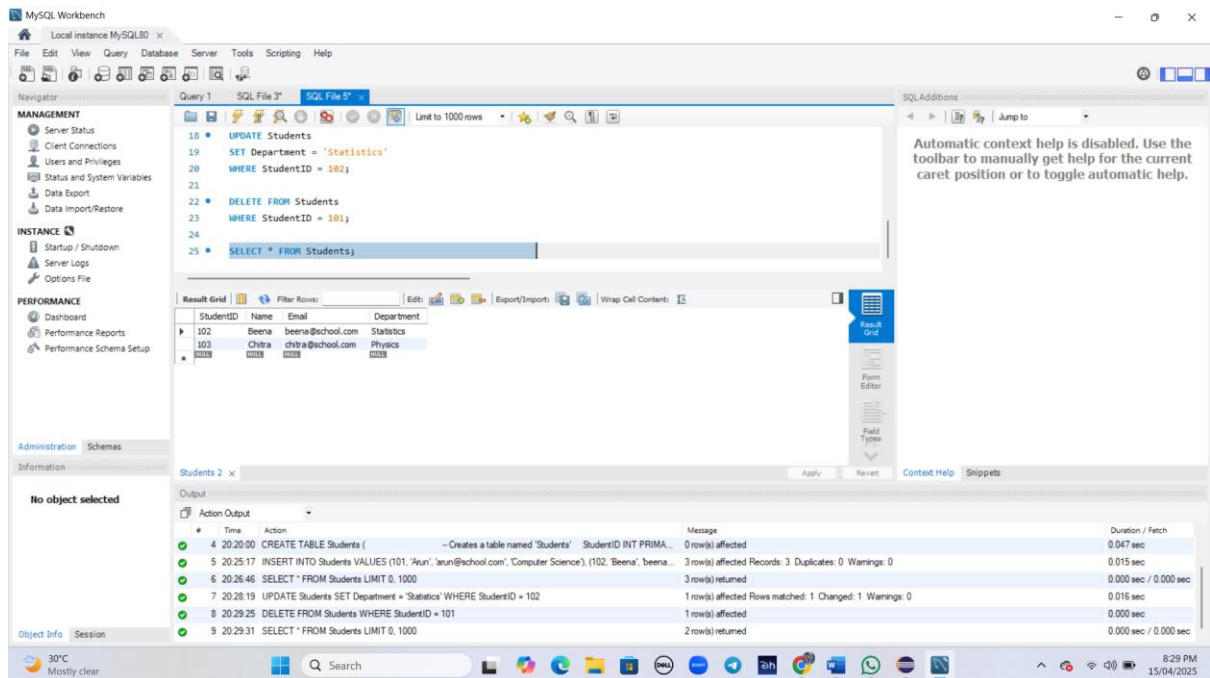
WHERE StudentID = 102;

STEP 7: DELETE A STUDENT RECORD

DELETE FROM Students
WHERE StudentID = 101;

STEP 8: FINAL VIEW OF THE TABLE

SELECT * FROM Students;



RESULT:

Thus, the SQL CRUD Operations has been executed successfully.

EX. NO:5	VERSION CONTROL WITH GIT
DATE:25/02/25	

AIM:

To understand and implement basic version control operations using Git such as repository creation, staging, committing, branching, merging, and collaboration via GitHub.

ALGORITHM:

STEP 1: Install Git and configure user identity.

STEP 2: Initialize a local Git repository.

STEP 3: Add and commit files to the repository.

STEP 4: Create and switch between branches.

STEP 5: Merge branches and resolve conflicts.

STEP 6: Push the local repository to GitHub.

STEP 7: Clone remote repositories and pull changes.

PROGRAM:

```
git --version
git config --global user.name "AjayKumarV"
git config --global user.email "ajaykumar2k@gmail.com.com"
mkdir git-demo
cd git-demo
git init
git status
git add hello.py
git commit -m "Initial commit: Added hello.py"

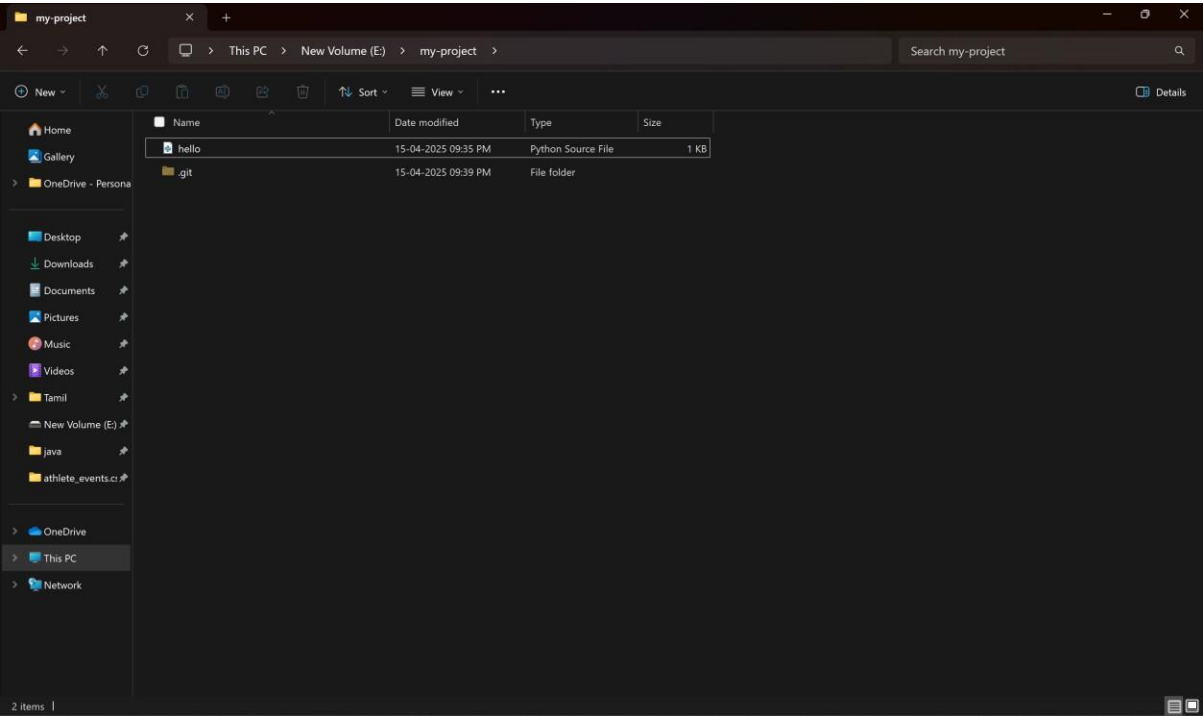
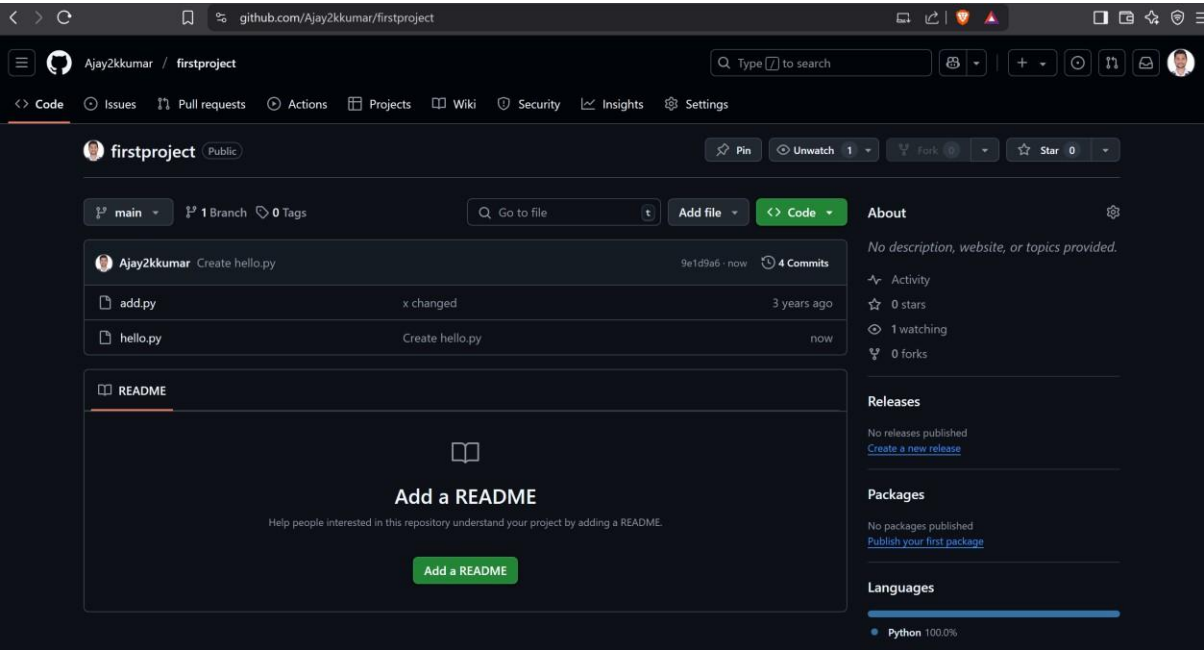
git branch feature
git checkout feature

git add hello.py
git commit -m "Updated hello.py in feature branch"

git checkout main
git merge feature

git remote add origin https://github.com/ajaykumarv/git-demo.git
git push -u origin main
cd ..
git clone https://github.com/ajaykumarv/git-demo.git
cd git-demo
git pull origin main
```


OUTPUT:



```
Initialized empty Git repository in /path/to/git-demo/.git/  
[main (root-commit) Initial commit: Added hello.py]  
Switched to branch 'feature'  
[feature Updated hello.py in feature branch]  
Switched to branch 'main'  
Merging feature branch...  
Merge made by the 'recursive' strategy.  
Pushing to GitHub repository...  
To https://github.com/ajaykumarv/git-demo.git  
* [new branch]    main -> main  
Cloning into 'git-demo'...
```

RESULT:

Thus, the version control operations using Git and GitHub have been implemented and verified successfully.

EX. NO:6	SIMPLE PYTHON CALCULATOR PROGRAM WITH GIT AND GITHUB TO MANAGE VERSION CONTROL
DATE:04/03/25	

AIM:

To create a basic calculator (Version 1) and an advanced calculator (Version 2) in Python, and manage their versions using Git and GitHub.

ALGORITHM:

Version 1: Normal Calculator

1. Display menu: Add, Subtract, Multiply, Divide.
2. Ask user to choose an operation.
3. Get two numbers from user.
4. Perform the selected operation.
5. Display result.
6. Repeat until user exits.

Version 2: Advanced Calculator

1. Inherit operations from the normal calculator.
2. Add more options: Square and Square Root.
3. Display all operations (1–6).
4. Get user input and execute corresponding method.
5. Loop until exit.
6. `git add .` stages all modified files.
7. `git commit -m "message"` commits the changes with a message, generating a unique commit hash.
8. `git checkout <commit-hash>` allows you to switch to a previous version of the repository by specifying the commit hash.
9. You can view your commit history on GitHub, which displays the commit hashes and messages for reference.

PROGRAM:

1. calc.py (Version 1 - Normal Calculator)

class Calculator:

```
def add(self, a, b):
    return a + b

def subtract(self, a, b):
    return a - b

def multiply(self, a, b):
    return a * b

def divide(self, a, b):
    if b != 0:
        return a / b
    else:
        return "Cannot divide by zero!"

calc = Calculator()

while True:
    print("\n1. Add\n2. Subtract\n3. Multiply\n4. Divide\n5. Exit")
    choice = input("Choose operation (1-5): ")

    if choice == '5':
        print("Exiting calculator...")
        break

    num1 = float(input("Enter first number: "))
    num2 = float(input("Enter second number: "))

    if choice == '1':
        print("Result:", calc.add(num1, num2))
    elif choice == '2':
        print("Result:", calc.subtract(num1, num2))
    elif choice == '3':
        print("Result:", calc.multiply(num1, num2))
    elif choice == '4':
        print("Result:", calc.divide(num1, num2))
    else:
        print("Invalid choice.")
```

#2. advcalc.py (Version 2 - Advanced Calculator using Inheritance)
import math

```
class Calculator:
    def add(self, a, b): return a + b
    def subtract(self, a, b): return a - b
    def multiply(self, a, b): return a * b
```

```
def divide(self, a, b): return a / b if b != 0 else "Cannot divide by zero"

class AdvancedCalculator(Calculator):
    def square(self, a): return a ** 2
    def square_root(self, a): return math.sqrt(a)

adv = AdvancedCalculator()

while True:
    print("\n1. Add\n2. Subtract\n3. Multiply\n4. Divide\n5. Square\n6. Square Root\n7. Exit")
    choice = input("Choose operation (1-7): ")

    if choice == '7':
        print("Exiting calculator...")
        break

    if choice in ['5', '6']:
        num = float(input("Enter a number: "))
        if choice == '5':
            print("Result:", adv.square(num))
        elif choice == '6':
            print("Result:", adv.square_root(num))
    else:
        num1 = float(input("Enter first number: "))
        num2 = float(input("Enter second number: "))
        if choice == '1':
            print("Result:", adv.add(num1, num2))
        elif choice == '2':
            print("Result:", adv.subtract(num1, num2))
        elif choice == '3':
            print("Result:", adv.multiply(num1, num2))
        elif choice == '4':
            print("Result:", adv.divide(num1, num2))
        else:
            print("Invalid choice.")
```

Git Commit Commands:

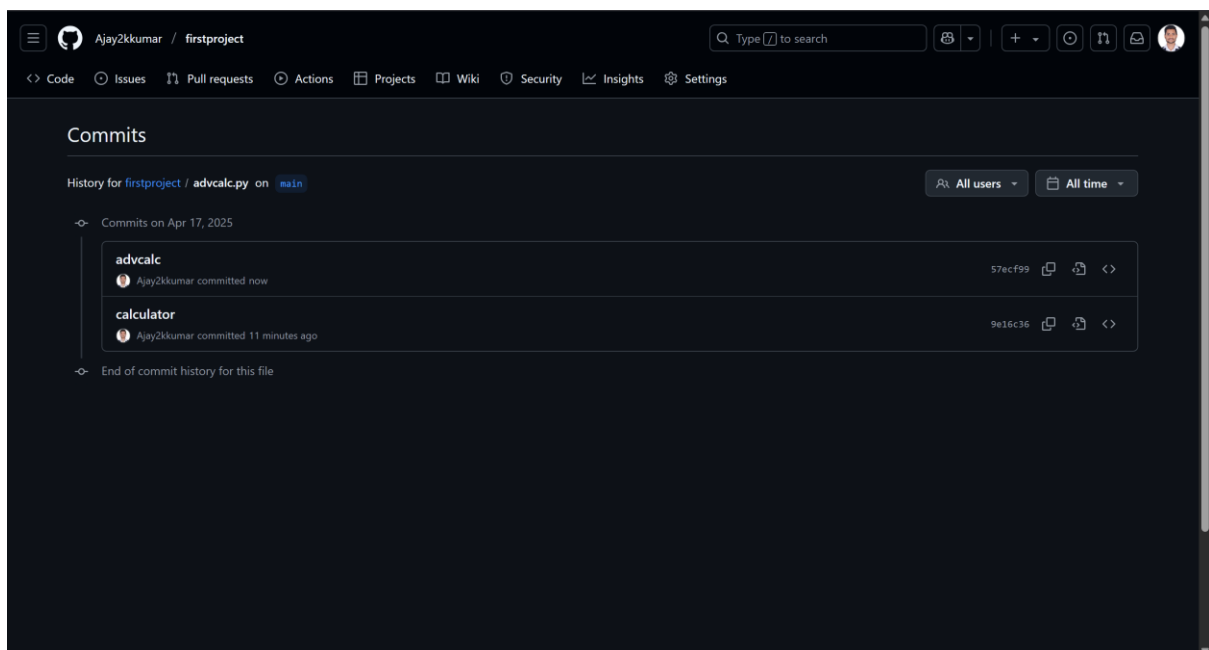
```
git add .
git commit -m "Version 1: Normal calculator"
# Commit hash: abc1234 (example)

git add .
git commit -m "Version 2: Advanced calculator with square and root"
# Commit hash: def5678 (example)

git checkout <commit-hash> # Use GitHub to view version history:
```

NAME: SANTHOSH S
REG NO: 3122246002011

OUTPUT:



```
Delta compression using up to 16 threads
Compressing objects: 100% (4/4), done.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 1.22 KiB | 1.22 MiB/s, done.
Writing objects: 100% (4/4), 1.22 KiB | 1.22 MiB/s, done.
Total 4 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
To https://github.com/Ajay2kkumar/firstproject
   9e1d9a6..9e16c36  main -> main
PS C:\Users\ajay kumar\Desktop\firstproject> git add .
PS C:\Users\ajay kumar\Desktop\firstproject> git commit
On branch main
Your branch is up to date with 'origin/main'.

nothing to commit, working tree clean
PS C:\Users\ajay kumar\Desktop\firstproject> git add .
PS C:\Users\ajay kumar\Desktop\firstproject> git commit -m "advcalc"
[main 57ecf99] advcalc
   1 file changed, 2 insertions(+), 2 deletions(-)
PS C:\Users\ajay kumar\Desktop\firstproject> git push
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 16 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 317 bytes | 317.00 KiB/s, done.
Total 3 (delta 2), reused 0 (delta 0), pack-reused 0 (from 0)
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To https://github.com/Ajay2kkumar/firstproject
   9e16c36..57ecf99  main -> main
PS C:\Users\ajay kumar\Desktop\firstproject> █
```

```
1. Add
2. Subtract
3. Multiply
4. Divide
5. Square
6. Square Root
Choose operation (1-6): 5
Enter a number: 5
Square: 25.0
PS C:\Users\ajay kumar\Desktop\firstproject> █
```

RESULT:

A basic calculator (Version 1) and an advanced calculator (Version 2) in Python, and manage their versions using Git and GitHub has been created successfully.

EX.NO:7	WEB APP DEVELOPMENT USING DJANGO PYTHON FRAMEWORK
DATE:11/03/25	

AIM:

To develop a **Student Management System** using **Django and MongoDB** to perform CRUD operations. It demonstrates database integration, model creation, and web application functionality.

Django:

Django is a high-level Python web framework that simplifies web development by providing built-in features for security, scalability, and rapid development. It follows the **Model-View-Template (MVT)** architecture, separating data handling, logic, and presentation. Django includes an Object-Relational Mapping (ORM) system, allowing developers to interact with databases using Python code instead of SQL. It supports multiple databases like MySQL, PostgreSQL, and SQLite. The framework offers automatic admin panel generation, authentication, middleware support, and built-in security against SQL injection and CSRF attacks. Django's modularity enables developers to reuse code and build applications faster. It is widely used for building scalable and secure web applications, including content management systems, e-commerce platforms, and APIs. Large companies like Instagram, Pinterest, and Mozilla use Django due to its efficiency and robustness. With a strong community and extensive documentation, Django remains one of the most popular web frameworks in the Python ecosystem.

MongoDB:

MongoDB is a **NoSQL database** designed for handling large-scale data efficiently. Unlike traditional relational databases, it stores data in flexible **JSON-like documents** instead of tables, making it highly scalable and adaptable. MongoDB supports **schemaless** data storage, allowing developers to store and retrieve complex data structures with ease. It is widely used for applications requiring high performance, real-time analytics, and cloud-based storage. MongoDB uses collections instead of tables, and each document can have different fields, making it ideal for dynamic applications. It supports **horizontal scaling**, meaning data can be distributed across multiple servers to handle large amounts of traffic. The database also includes **indexing, aggregation, and replication** for better performance and data redundancy. MongoDB integrates well with modern technologies, including Django, via libraries like **django** or **MongoEngine**. Its speed, flexibility, and ability to handle unstructured data make MongoDB a popular choice for modern web applications.

PREREQUISITES:

Python installed on your system.

Django framework.

MongoDB installed on your system and running.

Basic knowledge of Python and Django.

A text editor or IDE (such as VSCode or PyCharm).

CODE:

Step 1: Set Up the Environment

1.1. Install Django

`pip install django`

1.2. Install MongoDB Dependencies

Django does not natively support MongoDB. To connect Django with MongoDB, so need a special package called Djongo.

Install Djongo using pip:

`pip install djongo`

Djongo allows you to use MongoDB as the backend database in Django.

Step 2: Create a Django Project

2.1. Start a New Django Project

Once Django and Djongo are installed, let's create a new Django project:

`django-admin startproject student_management`

This creates a new directory called `student_management` containing the basic structure of a Django project.

2.2. Move into the Project Directory

Navigate into the project directory:

`cd student_management`

2.3. Create a Django App

In Django, an app is a component of the project. Create an app called `students` to handle student data.

Run the following command to create the app:

`python manage.py startapp students`

Now, check `students` directory inside the project.

Step 3: Configure MongoDB in Django

3.1. Update settings.py

Configure Django to use MongoDB. Open the settings.py file in your project directory (student_management/student_management/settings.py).

Find the DATABASES setting and update it to use Django with MongoDB.

```
DATABASES = {  
'default': {  
'ENGINE': 'djongo',  
'NAME': 'student_db', # Name of the MongoDB database  
'CLIENT': {  
'host': 'mongodb://localhost:27017', # MongoDB connection string  
}  
}  
}
```

Note:

ENGINE: Specifies the use of Django.

NAME: The name of the MongoDB database.

CLIENT: The connection string to MongoDB (default is localhost:27017).

3.2. Install MongoDB (If Not Installed)

If MongoDB is not installed, install it by following the instructions provided on the MongoDB LMS. After installation, make sure the MongoDB server is running:

```
mongod
```

Step 4: Create a Model for Students

In Django, models define the structure of the database. Create a model for the Student in the students/models.py file.

4.1. Define the Student Model

Open students/models.py and define a Student model like this:

```
from djongo import models  
class Student(models.Model):  
    name = models.CharField(max_length=100)  
    age = models.IntegerField()  
    major = models.CharField(max_length=100)  
    grade = models.CharField(max_length=5)  
    def __str__(self):  
        return self.name
```

Note:

name: A string field for the student's name.

age: An integer field for the student's age.

major: A string field for the student's major.

grade: A string field for the student's grade (e.g., A, B, C).

4.2. Make Migrations

Once the model is defined, run the following commands to create the database structure in MongoDB:

```
python manage.py makemigrations
```

```
python manage.py migrate
```

This will create the necessary collections in MongoDB for storing student data.

Step 5: Create Views and Templates

Create views to interact with our students' data.

5.1. Create Views for CRUD Operations

students/views.py

```
from .models import Student
# View to list all students
def student_list(request):
    students = Student.objects.all()
    return render(request, 'student_list.html', {'students': students})
# View to add a new student
def add_student(request):
    if request.method == 'POST':
        name = request.POST['name']
        age = request.POST['age']
        major = request.POST['major']
        grade = request.POST['grade']
        student = Student(name=name, age=age, major=major, grade=grade)
        student.save()
    return redirect('student_list')
    return render(request, 'add_student.html')
# View to update student details
def update_student(request, id):
    student = Student.objects.get(id=id)
    if request.method == 'POST':
        student.name = request.POST['name']
        student.age = request.POST['age']
        student.major = request.POST['major']
        student.grade = request.POST['grade']
        student.save()
    return redirect('student_list')
    return render(request, 'update_student.html', {'student': student})
# View to delete a student
def delete_student(request, id):
```

```
student = Student.objects.get(id=id)
student.delete()
return redirect('student_list')
```

5.2. Create Templates for the Views

Create HTML templates to display these views.

student_list.html: List all students.

```
<!DOCTYPE html>
<html>
<head>
<title>Student List</title>
</head>
<body>
<h1>Student List</h1>
<a href="{ % url 'add_student' % }">Add New Student</a>
<table border="1">
<tr>
<th>Name</th>
<th>Age</th>
<th>Major</th>
<th>Grade</th>
<th>Actions</th>
</tr>
{ % for student in students % }
<tr>
<td>{{ student.name }}</td>
<td>{{ student.age }}</td>
<td>{{ student.major }}</td>
<td>{{ student.grade }}</td>
<td>
<a href="{ % url 'update_student' student.id % }">Edit</a>
<a href="{ % url 'delete_student' student.id % }">Delete</a>
</td>
</tr>
{ % endfor % }
</table>
</body>
</html>
```

add_student.html :

```
<!DOCTYPE html>
<html>
<head>
```

```
<title>Add Student</title>
</head>
<body>
<h1>Add Student</h1>
<form method="post">
{% csrf_token %}
Name: <input type="text" name="name"><br>
Age: <input type="number" name="age"><br>
Major: <input type="text" name="major"><br>
Grade: <input type="text" name="grade"><br>
<input type="submit" value="Add Student">
</form>
<br>
<a href="{% url 'student_list' %}">Back to Student List</a>
</body>
</html>
```

Update.html:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Update Student</title>
</head>
<body>
  <div class="container">
    <h1>Update Student</h1>
    <form method="post">
      {% csrf_token %}
      Name: <input type="text" name="name" value="{ { student.name
}}"><br>
      Age: <input type="number" name="age" value="{ { student.age
}}"><br>
      Major: <input type="text" name="major" value="{ { student.major
}}"><br>
      Grade: <input type="text" name="grade" value="{ { student.grade
}}"><br>
      <input type="submit" value="Update Student">
    </form>
    <br>
    <a href="{% url 'student_list' %}">Back to Student List</a>
  </div>
```

```
</body>
</html>
```

Delete.html:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Delete Student</title>
</head>
<body>
  <div class="container">
    <h1>Delete Student</h1>
    <p>Are you sure you want to delete {{ student.name }}?</p>
    <form method="post">
      {% csrf_token %}
      <input type="submit" value="Yes, Delete">
    </form>
    <br>
    <a href="{% url 'student_list' %}">Cancel</a>
  </div>
</body>
</html>
```

Step 6: Set Up URLs

In students/urls.py, define the URL routes for the views:

```
from django.urls import path
from . import views
urlpatterns = [
    path("", views.student_list, name='student_list'),
    path('add/', views.add_student, name='add_student'),
    path('update/<int:id>/', views.update_student, name='update_student'),
    path('delete/<int:id>/', views.delete_student, name='delete_student'),
]
```

Then, include these URLs in the main urls.py of the project:

```
from django.contrib import admin
from django.urls import path, include
urlpatterns = [
    path('admin/', admin.site.urls),
    path('students/', include('students.urls')),
]
```

Step 7: Run the Project

7.1. Start the Development Server

Run the following command to start the Django development server:

```
python manage.py runserver
```

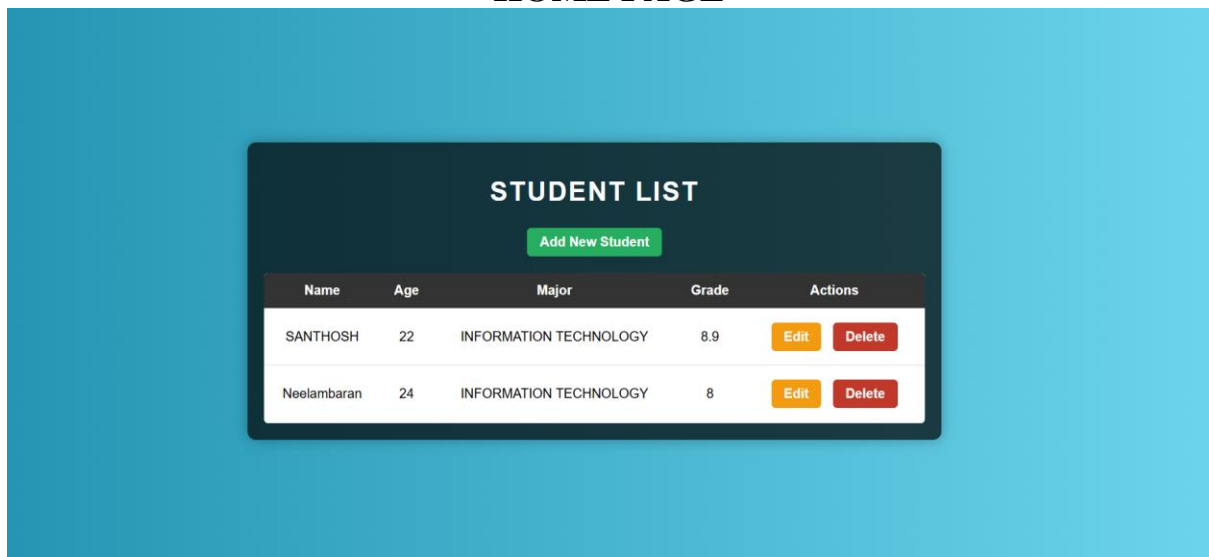
Visit <http://127.0.0.1:8000/students/> in Web browser. Check to find the student list, add a new student, edit existing students, and delete students.

OUTPUT:

MongoDB Connection

```
74 # Database
75 # https://docs.djangoproject.com/en/3.1/ref/settings/#databases
76
77 DATABASES = {
78     'default': {
79         'ENGINE': 'django',
80         'NAME': 'student_db',
81         'CLIENT': {
82             'host': 'mongodb://localhost:27017', # MongoDB connection string
83         }
84     }
85 }
86
```

HOME PAGE



ADD USERS PAGE

ADD Student DETAILS

Name:

Age:

Major:

Grade:

Update Student

Back to Student List

UPDATE PAGE

Update Student

Name:

Age:

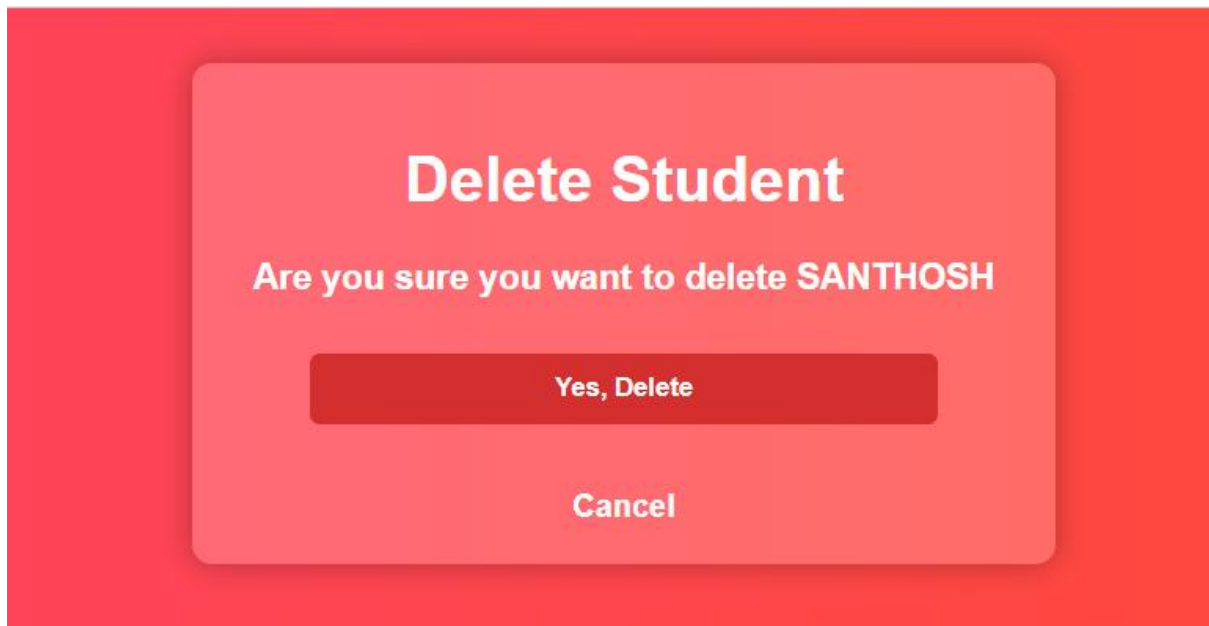
Major:

Grade:

Update Student

Back to Student List

DELETE PAGE



RESULT:

The experiment successfully implemented a Student Management System using Django and MongoDB, enabling CRUD operations. The system allows users to add, view, update, and delete student records through a web interface.

EX.NO:8	WEB APP DEVELOPMENT USING FLASK PYTHON FRAMEWORK
DATE:18/03/25	

AIM:

The aim of this project is to develop a Student Management System using Flask and MySQL that allows users to perform CRUD (Create, Read, Update, Delete) operations on student records. This system enables administrators to add new students, update student details, delete student records, and view the list of students in an organized manner.

About MySQL:

MySQL is an open-source relational database management system (RDBMS) that is widely used for web applications. It supports structured data storage and retrieval using SQL (Structured Query Language).

Key Features of MySQL:

- Relational Database: Data is stored in tables with relationships.
- Scalability: Efficient handling of large datasets.
- SQL Support: Uses structured queries for data manipulation.
- Transaction Management: Supports ACID (Atomicity, Consistency, Isolation, Durability).
- Security: Provides user authentication and access controls.

About Flask:

Flask is a **lightweight, micro web framework** for Python that allows developers to create web applications quickly and efficiently. It is widely used for building RESTful APIs and dynamic web applications.

Key Features of Flask:

- Lightweight & Minimalistic – No unnecessary dependencies.
- Built-in Development Server – Quick testing without external tools.
- Jinja2 Templating – Dynamic HTML rendering.
- Extensibility – Can integrate with SQL databases, authentication, and more.
- RESTful API Support – Ideal for backend development.

Why Use Flask?

- Easy to learn and use.
- Scalable for small and large applications.
- Compatible with many databases (MySQL, SQLite, PostgreSQL).

Creating a Flask Environment:

Step 1: Install Python

```
python --version
```

Step 2: Create a Virtual Environment

```
python -m venv Flask
```

Step 3: Activate the Virtual Environment

```
Flask\Scripts\activate
```

Step 4: Install Flask

```
pip install flask
```

```
python -m flask --version
```

Step 5: Install Flask and Flask-MySQLdb

```
pip install flask flask-mysqldb
```

CODE:

app.py:

```
from flask import Flask, render_template, url_for, redirect, request, flash
from flask_mysqldb import MySQL
```

```
app=Flask(__name__)
#MYSQL CONNECTION
app.config["MYSQL_HOST"]="localhost"
app.config["MYSQL_USER"]="root"
app.config["MYSQL_PASSWORD"]="sandy"
```

```
app.config["MYSQL_DB"]="crud"  
app.config["MYSQL_CURSORCLASS"]="DictCursor"  
mysql=MySQL(app)
```

#Loading Home Page

```
@app.route("/")  
def home():  
    con=mysql.connection.cursor()  
    sql="SELECT * FROM users"  
    con.execute(sql)  
    res=con.fetchall()  
    return render_template("home.html",datas=res)
```

#New User

```
@app.route("/addUsers",methods=['GET','POST'])  
def addUsers():  
    if request.method=='POST':  
        name=request.form['name']  
        city=request.form['city']  
        age=request.form['age']  
        con=mysql.connection.cursor()  
        sql="insert into users(NAME,CITY,AGE) value (%s,%s,%s)"  
        con.execute(sql,[name,city,age])  
        mysql.connection.commit()  
        con.close()  
        flash('User Details Added')  
        return redirect(url_for("home"))  
    return render_template("addUsers.html")
```

#update User

```
@app.route("/editUser/<string:id>",methods=['GET','POST'])  
  
def editUser(id):  
    con=mysql.connection.cursor()  
    if request.method=='POST':  
        name=request.form['name']  
        city=request.form['city']  
        age=request.form['age']  
        sql="update users set NAME=%s,CITY=%s,AGE=%s where ID=%s"  
        con.execute(sql,[name,city,age,id])  
        mysql.connection.commit()  
        con.close()  
        flash('User Detail Updated')  
        return redirect(url_for("home"))
```

```
con=mysql.connection.cursor()

sql="select * from users where ID=%s"
con.execute(sql,[id])
res=con.fetchone()
return render_template("editUser.html",datas=res)
#Delete User
@app.route("/deleteUser/<string:id>",methods=['GET','POST'])
def deleteUser(id):
    con=mysql.connection.cursor()
    sql="delete from users where ID=%s"
    con.execute(sql,id)
    mysql.connection.commit()
    con.close()
    flash('User Details Deleted')
    return redirect(url_for("home"))

if(__name__=='__main__'):
    app.secret_key="abc123"
    app.run(debug=True)
```

templates\index.html:

```
<html>
<head>
    <title>SSN STUDENT MANAGEMENT</title>
    <link rel="stylesheet" href="{{ url_for('static',filename='style.css')}} ">
</head>
<body>
    <h1>STUDENT DETAILS </h1>

    <h4>
        {% with messages = get_flashed_messages() %}
        {% if messages %}
            {% for message in messages %}
                {{ message }}
            {% endfor %}
        {% endif %}
        {% endwith %}
    </h4>

    <p><a href="{{ url_for('addUsers')}}">Add Users</a></p>
    <table>
        <tr>
```

```
<th>ID</th>
<th>NAME</th>
<th>AGE</th>
<th>CITY</th>
<th>EDIT</th>
<th>DELETE</th>
</tr>
{ % for data in datas % }
<tr>
<td>{{ data.ID }}</td>
<td>{{ data.NAME }}</td>
<td>{{ data.AGE }}</td>
<td>{{ data.CITY }}</td>
<td><a href="{{ url_for('editUser',id=data.ID) }}">Edit</a></td>
<td><a onclick="return confirm('Are you sure to delete?');"
href="{{ url_for('deleteUser',id=data.ID) }}">Delete</a></td>
</tr>
{ % endfor % }
</table>
</body>
</html>
```

templates\adduser.html:

```
<html>
<head>
<title>ADD STUDENT DETAILS</title>
<link rel="stylesheet" href="{{ url_for('static',filename='style.css') }}">
</head>
<body>
<h1>Add New Student Details</h1>
<form action="" method='post'>
<label>Name:</label>
<input type="text" name="name" required>
<label>Age:</label>
<input type="text" name="age" required>
<label>City:</label>
<input type="text" name="city" required>
<input type='submit' value='Save Details'>
</form>
</body>
</html>
```

templates\updateuser.html:

```
<html>
<head>
  <title>UPDATE STUDENT DETAILS</title>
  <link rel="stylesheet" href="{{ url_for('static',filename='style.css')}}" >
</head>
<body>
<h1>Update Student Details</h1>
  <form action="" method='post'>
    <label>Name:</label>
    <input type="text" name="name" value="{{ datas.NAME }}" required>
    <label>Age:</label>
    <input type="text" name="age" value="{{ datas.AGE }}" required>
    <label>City:</label>
    <input type="text" name="city" value="{{ datas.CITY }}" required>
    <input type='submit' value='Update Details'>
  </form>
</body>
</html>
```

OUTPUT:

Home Page

STUDENT DETAILS

User Details Deleted

Add Users

ID	NAME	AGE	CITY	EDIT	DELETE
1	SANTHOSH	22	trichy	Edit	Delete
4	neelambaran	21	rameshwaram	Edit	Delete

Add User Page

Add New Student Details

Name:

Age:

City:

Save Details

Update Page

Update Student Details

Name:

SANTHOSH

Age:

22

City:

trichy

Update Details

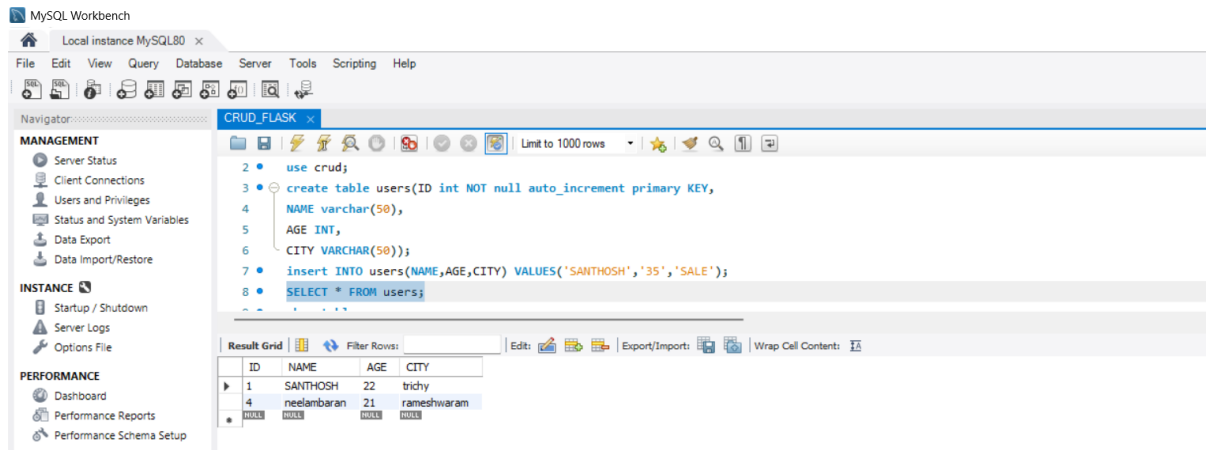
Delete Page



Add Users

ID	NAME	AGE	CITY	EDIT	DELETE
1	SANTHOSH	22	trichy	Edit	Delete
4	neelambaran	21	rameshwaram	Edit	Delete

MYSQL DATABASE



RESULT:

The experiment successfully implemented a Student Management System using Flask and MYSQLDB, enabling CRUD operations. The system allows users to add, view, update, and delete student records through a web interface.

EX.NO : 09	DJANGO AND DJANGO REST FRAMEWORK
DATE : 01/04/25	

AIM:

To develop a **RESTful API using Django and Django REST Framework** for performing CRUD operations (Create, Read, Update, Delete) on an Item model that includes fields like category, subcategory, name, and amount.

ALGORITHM:

Step 1: Install and Set Up Django + DRF

- Install Django and DRF using pip.
- Create a Django project and app (api).
- Register the app and rest_framework in settings.py.

Step 2: Design the Database Model

- Define a model Item with fields: category, subcategory, name, and amount.
- Run migrations to apply the model to the database.

Step 3: Create a Serializer

- Use ModelSerializer from DRF to convert model instances to JSON and validate input.

Step 4: Write Views for CRUD

- **Overview View:** Lists all available API routes.
- **Create View:** Accepts POST data to create a new item if it doesn't already exist.
- **Read View:** Returns all items or filters by query params (like ?category=food).
- **Update View:** Updates an existing item based on pk.
- **Delete View:** Deletes an item using pk.

Step 5: Set Up URLs

- Link each view to a specific route using Django's path() function in api/urls.py.
- Include api.urls in the project-level urls.py under the prefix api/.

Step 6: Run and Test the Server

- Use python manage.py runserver to start the server.
- Access endpoints like:
 - GET /api/ – API overview
 - POST /api/create/ – Create an item
 - GET /api/all/ – List or filter items
 - POST /api/update/<pk>/ – Update an item
 - DELETE /api/item/<pk>/delete/ – Delete an item

PROGRAM:

```
pip install django djangorestframework
django-admin startproject myproject
cd myproject
python manage.py startapp api
```

```
# myproject/settings.py
INSTALLED_APPS = [
    ...
    'rest_framework',
    'api.apps.ApiConfig', # Your custom app
]
```

```
#In project/urls.py:
from django.contrib import admin
from django.urls import path, include
urlpatterns = [
    path('admin/', admin.site.urls),
    path('api/', include('api.urls')),
]
```

```
#In api/models.py
from django.db import models
class Item(models.Model):
    category = models.CharField(max_length=255)
    subcategory = models.CharField(max_length=255)
    name = models.CharField(max_length=255)
    amount = models.PositiveIntegerField()

    def __str__(self):
        return self.name
```

```
#migrate
python manage.py makemigrations
python manage.py migrate
```

```
#In api/serializers.py
from rest_framework import serializers
from .models import Item
```

```
class ItemSerializer(serializers.ModelSerializer):
    class Meta:
        model = Item
        fields = ('category', 'subcategory', 'name', 'amount')
```

```
#In api/views.py
a. Overview
from rest_framework.decorators import api_view
from rest_framework.response import Response
```

```
@api_view(['GET'])
def ApiOverview(request):
    api_urls = {
        'All Items': '/all/',
        'Search by Category': '/all/?category=food',
        'Add': '/create/',
        'Update': '/update/<pk>/',
        'Delete': '/item/<pk>/delete/',
    }
    return Response(api_urls)
```

b. Create Item

```
from rest_framework import status, serializers
from .models import Item
from .serializers import ItemSerializer
```

```
@api_view(['POST'])
def add_items(request):
    item = ItemSerializer(data=request.data)
    if Item.objects.filter(**request.data).exists():
        raise serializers.ValidationError("This data already exists")
    if item.is_valid():
        item.save()
        return Response(item.data)
    return Response(status=status.HTTP_404_NOT_FOUND)
```

c. Read Items

```
@api_view(['GET'])
def view_items(request):
    if request.query_params:
        items = Item.objects.filter(**request.query_params.dict())
    else:
        items = Item.objects.all()

    if items:
        serializer = ItemSerializer(items, many=True)
        return Response(serializer.data)
    return Response(status=status.HTTP_404_NOT_FOUND)
```

d. Update Item

```
@api_view(['POST'])
def update_items(request, pk):
    item = Item.objects.get(pk=pk)
    data = ItemSerializer(instance=item, data=request.data)
    if data.is_valid():
        data.save()
        return Response(data.data)
    return Response(status=status.HTTP_404_NOT_FOUND)
```

e. Delete Item

```
from django.shortcuts import get_object_or_404
```

```
@api_view(['DELETE'])
def delete_items(request, pk):
    item = get_object_or_404(Item, pk=pk)
    item.delete()
    return Response(status=status.HTTP_202_ACCEPTED)
```

```
#in api/urls.py
```

```
from django.urls import path
from . import views
```

```
urlpatterns = [
    path("", views.ApiOverview, name='home'),
    path('create/', views.add_items, name='add-items'),
    path('all/', views.view_items, name='view-items'),
    path('update/<int:pk>', views.update_items, name='update-items'),
    path('item/<int:pk>/delete/', views.delete_items, name='delete-items'),
```

```
#Run the Server
```

```
python manage.py runserver
```

OUTPUT:

```
(env) C:\Windows\System32\myproject\myproject\myproject>python manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).
April 16, 2025 - 14:37:37
Django version 5.2, using settings 'myproject.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

http://127.0.0.1:8000/api/

Api Overview

OPTIONS

GET

GET /api/

```
HTTP 200 OK
Allow: GET, OPTIONS
Content-Type: application/json
Vary: Accept

{
  "all_items": "/",
  "Search by Category": "/?category=category_name",
  "Search by Subcategory": "/?subcategory=category_name",
  "Add": "/create",
  "Update": "/update/pk",
  "Delete": "/item/pk/delete"
}
```

http://127.0.0.1:8000/api/create/

Add Items

OPTIONS

POST /api/create/

```
HTTP 200 OK
Allow: POST, OPTIONS
Content-Type: application/json
Vary: Accept
```

```
{
  "category": "food",
  "subcategory": "vegetables",
  "name": "potato",
  "amount": 25
}
```

Media type:

application/json

Content:

```
{
  "category": "food",
  "subcategory": "vegetables",
  "name": "potato",
  "amount": 25
}
```



POST

http://127.0.0.1:8000/api/all/

View Items

OPTIONS

GET

GET /api/all/

```
HTTP 200 OK
Allow: OPTIONS, GET
Content-Type: application/json
Vary: Accept
```

```
[
  {
    "category": "food",
    "subcategory": "vegetables",
    "name": "potato",
    "amount": 25
  },
  {
    "category": "food",
    "subcategory": "vegetables",
    "name": "tomato",
    "amount": 80
  },
  {
    "category": "house hold",
    "subcategory": "kitchen",
    "name": "gas stove",
    "amount": 3500
  },
  {
    "category": "food",
    "subcategory": "fruit",
    "name": "apple",
    "amount": 100
  }
]
```

http://127.0.0.1:8000/api/all/?category

View Items OPTIONS GET

GET /api/all/?subcatgeory=vegetables

HTTP 200 OK
Allow: OPTIONS, GET
Content-Type: application/json
Vary: Accept

```
[
  {
    "category": "food",
    "subcatgeory": "vegetables",
    "name": "potato",
    "amount": 25
  },
  {
    "category": "food",
    "subcatgeory": "vegetables",
    "name": "tomato",
    "amount": 89
  }
]
```

http://127.0.0.1:8000/api/all/?name=potato

Update Items OPTIONS

GET /api/update/1/

HTTP 405 Method Not Allowed
Allow: POST, OPTIONS
Content-Type: application/json
Vary: Accept

```
{
  "detail": "Method \"GET\" not allowed."
}
```

Media type:
application/json

Content:

```
{
  "category": "food",
  "subcatgeory": "vegetables",
  "name": "potato",
  "amount": 30
}
```

POST

http://127.0.0.1:8000/api/item/pk/delete/

Delete Items DELETE OPTIONS

DELETE /api/item/3/delete/

HTTP 202 Accepted
Allow: DELETE, OPTIONS
Content-Type: application/json
Vary: Accept

RESULT:

This project implements a fully functional RESTful API using Django REST Framework to manage Item data with support for creating, reading, updating, and deleting items. All interactions return JSON responses, and the API can be tested using tools like Postman, curl, or the DRF web interface.

SANTHOSH S
3122246002011