

INDEX

Exercise Number	Date	Name of the Exercise	Page No	Mark	Staff Initial
1 (A)		Create classes for BankAccount, SavingsAccount and CheckingAccount and implement methods for deposit, withdrawal, balance inquiry and interest calculation			
1 (B)		Design a Python class hierarchy for a library system with classes for Library, Book, and Member. Implement methods for book borrowing, returning, and displaying availability, as well as for managing members' borrowed books list.			
2 (A)		Create classes for Employee, Payroll, and Salary and implement methods for calculating employee salaries, generating pay slips, and managing payroll records using inheritance			
2 (B)		Design Python classes for Product, Inventory, and Sales to manage product information, track inventory levels, and record sales transactions, respectively, using inheritance			
3 (A)		Create a graphical calculator with buttons for numeric inputs, arithmetic operations and show the results			
3 (B)		Create a graphical calendar application with buttons to navigate through months and years, and display selected dates and events..			
4 (A)		Develop a GUI based to do list application where user can add, delete and manage their task			
4 (B)		Develop a GUI-based notes application where users can create, edit, and manage notes with features like formatting text, saving notes, and organizing them into categories.			
5 (A)		Develop a Python program to fetch records from a table in MySQL and display the results.			
5 (B)		Develop a Python program to insert records into a table in MySQL using user input and display a confirmation message upon successful insertion.			
6 (A)		Develop a CRUD program using Python-Mysql connectivity			
6 (B)		Develop a Python program to manage student records in a MySQL database, allowing users to add, view, update, and delete student information.			
7 (A)		Create a client server based chat application where multiple clients can connect to a server and exchange message			

7 (B)		Create a client server based chat application where multiple clients can connect to a server			
8 (A)		Build a network port scanner program that scans a given IP addresses to detect open ports on remote machine			
8 (B)		Develop a network vulnerability scanner program that not only detects open ports but also checks for known vulnerabilities on those ports.			
9 (A)		Develop a social media platform where users can create profiles, post updates, connect with friends, and engage with content through likes and comments.			
9(B)		Develop an e-commerce store application with features like product listings , user authentication , shopping cart management and secure payment integration			
10 (A)		Build a python program to perform Handling of missing data.			
10 (B)		Develop a ticket booking platform where users can browse available events, select seats, and purchase tickets securely.			

Ex. No. 1(A)	Create classes for BankAccount, SavingsAccount and CheckingAccount and implement methods for deposit, withdrawal, balance inquiry and interest calculation
Date:	

Aim:

To Create classes for BankAccount, SavingsAccount and CheckingAccount and implement methods for deposit, withdrawal, balance inquiry and interest calculation

Algorithm:

1. Start
2. Define a parent class BankAccount with attributes such as account_number, balance.
3. Implement methods for deposit, withdrawal, and balance inquiry in the BankAccount class.
4. Create child classes SavingsAccount and CheckingAccount inheriting from BankAccount.
5. Implement interest calculation method in the SavingsAccount class. Implement overdraft protection method in the CheckingAccount class.
6. Stop

Program:

```
class BankAccount:
    def __init__(self, account_number, balance=0):
        self.account_number = account_number
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount
        print(f"Deposited ${amount}. New balance: ${self.balance}")

    def withdraw(self, amount):
        if self.balance >= amount:
            self.balance -= amount
            print(f"Withdrew ${amount}. New balance: ${self.balance}")
        else:
            print("Insufficient funds!")

    def inquiry(self):
        print(f"Account Balance: ${self.balance}")

class SavingsAccount(BankAccount):
    def calculate_interest(self, rate):
```

```
        interest = self.balance * (rate / 100)
        self.deposit(interest)
        print(f"Interest of ${interest} added. New balance: ${self.balance}")

class CheckingAccount(BankAccount):
    def overdraft_protection(self, amount):
        if self.balance >= amount:
            self.balance -= amount
            print(f"Withdrew ${amount}. New balance: ${self.balance}")
        else:
            print("Overdraft Protection: Insufficient funds!")
            print("Transaction canceled.")

# Example usage
savings_acc = SavingsAccount(account_number="SAV12345", balance=1000)
savings_acc.deposit(500)
savings_acc.calculate_interest(2) # 2% interest rate
savings_acc.inquiry()

checking_acc = CheckingAccount(account_number="CHK67890", balance=500)
checking_acc.withdraw(600) # This will fail due to insufficient funds
checking_acc.overdraft_protection(600) # With overdraft protection
checking_acc.inquiry()
```

Output:

```
Deposited $500. New balance: $1500
Deposited $30.0. New balance: $1530.0
Interest of $30.0 added. New balance: $1530.0
Account Balance: $1530.0
Withdrew $600. New balance: $-100
Overdraft Protection: Insufficient funds!
Transaction canceled.
Account Balance: $-100
```

Result:

Implemented classes for BankAccount, SavingsAccount, and CheckingAccount with methods for deposit, withdrawal, balance inquiry, and interest calculation.

Ex. No. 1(B)	Design a Python class hierarchy for a library system with classes for Library, Book, and Member. Implement methods for book borrowing, returning, and displaying availability, as well as for managing members' borrowed books list.
Date:	

Aim:

To create a simple library management system that allows users to efficiently borrow and return books while keeping track of available resources.

Algorithm:

Step 1: Initialize library with a list of books and members.

Step 2: Display available books in the library.

Step 3: Prompt user to input member name and create a member object.

Step 4: Ask user for the book title to borrow; if available, assign it to the member.

Step 5: Show updated list of available books after borrowing.

Step 6: Prompt user to input the book title to return; process the return and update availability.

Program:

```
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author
        self.is_borrowed = False

class Member:
    def __init__(self, name):
        self.name = name
        self.borrowed_books = []

    def borrow_book(self, book):
        if not book.is_borrowed:
            book.is_borrowed = True
            self.borrowed_books.append(book)
            print(f"{self.name} borrowed '{book.title}'.")
        else:
            print(f"'{book.title}' is already borrowed.")

    def return_book(self, book):
        if book in self.borrowed_books:
            book.is_borrowed = False
            self.borrowed_books.remove(book)
            print(f"{self.name} returned '{book.title}'.")
```

```

        else:
            print(f"{self.name} didn't borrow '{book.title}'.")

class Library:
    def __init__(self):
        self.books = []

    def add_book(self, book):
        self.books.append(book)

    def display_available_books(self):
        print("Available books:")
        for book in self.books:
            if not book.is_borrowed:
                print(f"- {book.title} by {book.author}")

# Example usage with user input:

library = Library()
library.add_book(Book("1984", "George Orwell"))
library.add_book(Book("To Kill a Mockingbird", "Harper Lee"))

member_name = input("Enter member name: ")
member = Member(member_name)
library.display_available_books()

book_title = input("Enter the title of the book to borrow: ")
book_to_borrow = next((b for b in library.books if b.title == book_title), None)
if book_to_borrow:
    member.borrow_book(book_to_borrow)
library.display_available_books()

# To return a book:
return_title = input("Enter the title of the book to return: ")
book_to_return = next((b for b in member.borrowed_books if b.title ==
return_title), None)
if book_to_return:
    member.return_book(book_to_return)
library.display_available_books()

```

Output:

Enter member name: Alice

Available books:

- 1984 by George Orwell
- To Kill a Mockingbird by Harper Lee

Enter the title of the book to borrow: 1984

Alice borrowed '1984'.

Available books:

- To Kill a Mockingbird by Harper Lee

Enter the title of the book to return: To Kill a Mockingbird by Harper Lee

Available books:

- To Kill a Mockingbird by Harper Lee

Particulars	Max Marks	Mark secured
Program and Execution	15	
Viva	10	
Total	25	

Result:

The system efficiently manages book borrowing and returning, ensuring accurate tracking of resources.

Ex. No. 2(A)	Create classes for Employee, Payroll, and Salary and implement methods for calculating employee salaries, generating pay slips, and managing payroll records using inheritance
Date:	

Aim:

To create classes for Employee, Payroll, and Salary and implement methods for calculating employee salaries, generating pay slips, and managing payroll records using inheritance.

Algorithm:**Step 1:** Start

Step 2: Define an Employee class with attributes such as name, employee ID, and salary

Step 3: Define a Salary class inheriting from Employee to calculate the salary based on hours worked or fixed salary

Step 4: Define a Payroll class to generate pay slips and manage payroll records, utilizing methods for salary calculation and employee information retrieval.

Step 5: Stop the process

Program:

```
class Employee:
    def __init__(self, name, employee_id, salary):
        self.name = name
        self.employee_id = employee_id
        self.salary = salary

class Salary(Employee):
    def __init__(self, name, employee_id, salary):
        super().__init__(name, employee_id, salary)

    def calculate_salary(self):
        return self.salary

class Payroll:
    def generate_pay_slip(self, employee):
        print("Pay Slip:")
        print(f"Name: {employee.name}")
        print(f"Employee ID: {employee.employee_id}")
        print(f"Salary: ${employee.salary}")
        print("=" * 20)

# Sample usage
employee1 = Salary("John Doe", 1001, 5000)
employee2 = Salary("Jane Smith", 1002, 6000)

payroll = Payroll()
```



```
payroll.generate_pay_slip(employee1)  
payroll.generate_pay_slip(employee2)
```

Output :

```
Pay Slip:  
Name: John Doe  
Employee ID: 1001  
Salary: $5000  
=====
```

```
Pay Slip:  
Name: Jane Smith  
Employee ID: 1002  
Salary: $6000  
=====
```

Result:

The code provides the class structure for Employee, Salary, and Payroll. Methods and attributes can be added to each class for further functionality.

Ex. No. 2(B)	Design Python classes for Product, Inventory, and Sales to manage product information, track inventory levels, and record sales transactions, respectively, using inheritance.
Date:	

Aim:

To develop a simple inventory management system that allows users to add products, record sales, and track remaining stock through object-oriented programming with inheritance.

Algorithm:

Step 1: Prompt the user to input product details: name, price, and initial stock quantity.

Step 2: Create an Inventory object with the provided product information.

Step 3: Prompt the user to input the quantity of product to sell.

Step 4: Create a Sales object, inheriting from Inventory, to manage sales transactions.

Step 5: Record the sale by updating inventory stock and logging the sale details.

Step 6: Display the remaining stock and the sales record to the user.

Program:

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

class Inventory(Product):
    def __init__(self, name, price, quantity):
        super().__init__(name, price)
        self.quantity = quantity

    def update_stock(self, amount):
        self.quantity += amount

    def check_stock(self):
        return self.quantity

class Sales(Inventory):
    def __init__(self, name, price, quantity):
        super().__init__(name, price, quantity)
        self.sales_record = []

    def record_sale(self, quantity_sold):
        if quantity_sold <= self.quantity:
```

```

        self.quantity -= quantity_sold
        total_price = self.price * quantity_sold
        self.sales_record.append((self.name, quantity_sold, total_price))
        print(f"Sold {quantity_sold} units of {self.name}.")
    else:
        print("Insufficient stock for sale.")

# User input
product_name = input("Enter product name: ")
product_price = float(input("Enter product price: "))
initial_quantity = int(input("Enter initial inventory quantity: "))

# Create inventory object
product = Inventory(product_name, product_price, initial_quantity)

# Record a sale
sale_quantity = int(input("Enter quantity to sell: "))
sales = Sales(product_name, product_price, initial_quantity)
sales.record_sale(sale_quantity)

# Output
print(f"Remaining stock of {product.name}: {product.check_stock()}")
print("Sales record:", sales.sales_record)

```

Output:

```

Enter product name: soap
Enter product price: 60
Enter initial inventory quantity: 10
Enter quantity to sell: 8
Sold 8 units of soap.
Remaining stock of soap: 10
Sales record: [('soap', 8, 480.0)]

```

Particulars	Max Marks	Mark secured
Program and Execution	15	
Viva	10	
Total	25	

Result:

This program manages product inventory and sales by creating product objects, recording sales, updating stock, and displaying remaining inventory and sales history.

Ex. No. 3(A)	Create a graphical calculator with buttons for numeric inputs, arithmetic operations and show the results
DATE	

Aim :

Create a graphical calculator with buttons for numeric inputs, arithmetic operations, and display the results.

Algorithm:

Step 1: Start

Step 2: Design a graphical user interface (GUI) with buttons for numeric inputs (0-9), arithmetic operations (+, -, *, /), and other functionalities (e.g., clear, equals).

Step 3: Implement event handlers for button clicks to update the input field and perform calculations.

Step 4: Display the output to user

Step 5: Display the result of calculations in a designated area.

Program:

```
import tkinter as tk
```

```
def on_button_click(char):
    if char == 'C':
        input_field.delete(0, tk.END)
    elif char == '=':
        try:
            result = eval(input_field.get())
            input_field.delete(0, tk.END)
            input_field.insert(tk.END, str(result))
        except Exception as e:
            input_field.delete(0, tk.END)
            input_field.insert(tk.END, "Error")
    else:
        input_field.insert(tk.END, char)
```

```
root = tk.Tk()
root.title("Calculator")
```

```
input_field = tk.Entry(root, width=30, justify="right")
input_field.grid(row=0, column=0, columnspan=4)
```

```
buttons = [  
    '7', '8', '9', '/',  
    '4', '5', '6', '*',  
    '1', '2', '3', '-',  
    'C', '0', '=', '+'  
]  
  
row = 1  
col = 0  
  
for button in buttons:  
    tk.Button(root, text=button, width=5, height=2,  
              command=lambda char=button: on_button_click(char)).grid(row=row,  
                                column=col)  
    col += 1  
    if col > 3:  
        col = 0  
        row += 1  
  
root.mainloop()
```

Output:



Result:

Thus the Users can input numbers and perform arithmetic operations using the graphical calculator interface, with the results displayed in real-time.

Ex. No. 3(B)	Create a graphical calendar application with buttons to navigate through months and years, and display selected dates and events.
DATE	

Aim:

To create a simple, interactive graphical calendar that allows users to navigate through months and years, select dates, and see the selected date displayed.

Algorithm:

- Step 1:** Initialize the calendar app with current date.
- Step 2:** Display the current month and year.
- Step 3:** Create navigation buttons to move to previous and next months.
- Step 4:** Generate and show the calendar grid for the selected month.
- Step 5:** Allow user to click on a date button to select a date.
- Step 6:** Update the display to show the selected date.
- Step 7:** On navigation, update the month/year and regenerate the calendar grid.

Program:

```
import tkinter as tk
from tkinter import ttk
import calendar
from datetime import datetime

class CalendarApp:
    def __init__(self, root):
        self.root = root
        self.root.title("Simple Calendar")
        self.current_year = datetime.now().year
        self.current_month = datetime.now().month

        # Header Frame for navigation
        header = ttk.Frame(root)
        header.pack()

        self.prev_btn = ttk.Button(header, text="<<", command=self.prev_month)
        self.prev_btn.grid(row=0, column=0)

        self.month_year_lbl = ttk.Label(header, text="", width=15, anchor='center')
        self.month_year_lbl.grid(row=0, column=1)

        self.next_btn = ttk.Button(header, text=">>", command=self.next_month)
        self.next_btn.grid(row=0, column=2)
```

```

# Calendar Frame
self.calendar_frame = ttk.Frame(root)
self.calendar_frame.pack()

# Label to show selected date
self.date_lbl = ttk.Label(root, text="Select a date")
self.date_lbl.pack(pady=10)

self.show_calendar()

def show_calendar(self):
    # Clear previous widgets
    for widget in self.calendar_frame.winfo_children():
        widget.destroy()

    # Update header label

self.month_year_lbl.config(text=f"{calendar.month_name[self.current_month]}
{self.current_year}")

    # Weekday headers
    days = ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']
    for idx, day in enumerate(days):
        ttk.Label(self.calendar_frame, text=day).grid(row=0, column=idx)

    # Get calendar matrix
    cal =
calendar.Calendar(firstweekday=0).monthdayscalendar(self.current_year,
self.current_month)

    # Create buttons for days
    for row_idx, week in enumerate(cal, start=1):
        for col_idx, day in enumerate(week):
            if day == 0:
                continue
            btn = ttk.Button(self.calendar_frame, text=str(day),
                             command=lambda d=day: self.select_date(d))
            btn.grid(row=row_idx, column=col_idx, padx=2, pady=2)

def select_date(self, day):
    self.date_lbl.config(text=f"Selected Date:
{day}-{self.current_month}-{self.current_year}")

def prev_month(self):
    if self.current_month == 1:
        self.current_month = 12
        self.current_year -= 1
    else:
        self.current_month -= 1

```

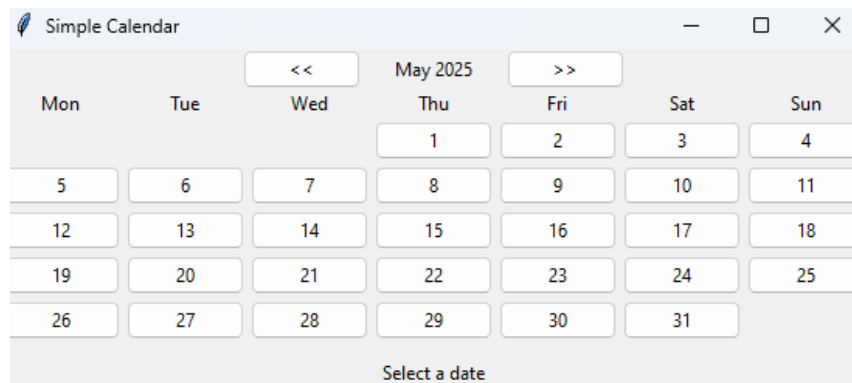
```

self.show_calendar()

def next_month(self):
    if self.current_month == 12:
        self.current_month = 1
        self.current_year += 1
    else:
        self.current_month += 1
    self.show_calendar()

if __name__ == "__main__":
    root = tk.Tk()
    app = CalendarApp(root)
    root.mainloop()

```

Output:

Particulars	Max Marks	Mark secured
Program and Execution	15	
Viva	10	
Total	25	

Result:

An interactive GUI calendar that allows navigation through months and date selection with the selected date displayed.

EX NO 4(a)	Develop a GUI based to do list application where user can add, delete and manage their task
DATE	

Aim:

To Develop a GUI-based to-do list application where users can add, delete, and manage their tasks.

Algorithm:

Step 1: Start

Step 2: Design a graphical user interface (GUI) with input field to add tasks, a listbox to display tasks, and buttons for adding, deleting, and managing tasks.

Step 3: Use appropriate data structures like lists or dictionaries to store and manage tasks

Step 4: Stop the process

Program:

```
import tkinter as tk
```

```
def add_task():
    task = task_entry.get()
    if task:
        tasks.append(task)
        update_task_list()
```

```
def delete_task():
    selected_task_index = task_listbox.curselection()
    if selected_task_index:
        tasks.pop(selected_task_index[0])
        update_task_list()
```

```
def update_task_list():
    task_listbox.delete(0, tk.END)
    for task in tasks:
        task_listbox.insert(tk.END, task)
```

```
root = tk.Tk()
root.title("To-Do List")
```

```
tasks = []
```

```
task_entry = tk.Entry(root, width=30)
task_entry.grid(row=0, column=0, padx=5, pady=5)
```

```
add_button = tk.Button(root, text="Add Task", command=add_task)
add_button.grid(row=0, column=1, padx=5, pady=5)
```

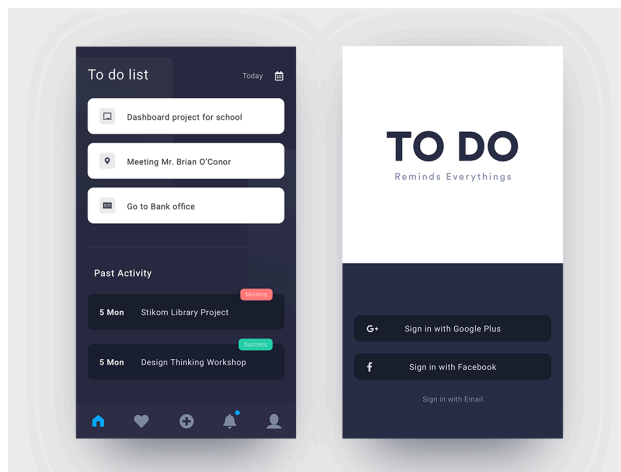
```
delete_button = tk.Button(root, text="Delete Task", command=delete_task)
delete_button.grid(row=0, column=2, padx=5, pady=5)
```

```
task_listbox = tk.Listbox(root, width=50)
task_listbox.grid(row=1, column=0, columnspan=3, padx=5, pady=5)
```

```
update_task_list()
```

```
root.mainloop()
```

Output :



Result:

Thus the Users can add tasks to their to-do list using the input field and the "Add Task" button. They can delete selected tasks using the "Delete Task" button.

Ex. No. 4(b)	Develop a GUI-based notes application where users can create, edit, and manage notes with features like formatting text, saving notes, and organizing them into categories.
DATE	

Aim:

To develop a GUI-based notes application that allows users to create, edit, organize, and save notes within categories.

Algorithm:

Step1: Initialize the main window with category, note list, and editor sections.

Step2: Enable adding new categories and display them in the category list.

Step3: When a category is selected, load its notes into the note list.

Step4: Allow creating new notes within the selected category.

Step5: Load selected notes into the editor for editing.

Step6: Save edited content back to the note data structure .

Program:

```
import tkinter as tk
from tkinter import ttk, simpledialog, messagebox

class NotesApp:
    def __init__(self, master):
        self.master = master
        master.title("Simple Notes App")

        self.categories = {}
        self.current_category = None
        self.current_note = None

        # Category frame
        cat_frame = ttk.Frame(master)
        cat_frame.pack(side=tk.LEFT, fill=tk.Y, padx=5, pady=5)

        ttk.Label(cat_frame, text="Categories").pack()
        self.cat_listbox = tk.Listbox(cat_frame)
        self.cat_listbox.pack(fill=tk.Y, expand=True)
        self.cat_listbox.bind('<<ListboxSelect>>', self.load_notes)

        ttk.Button(cat_frame, text="Add Category",
command=self.add_category).pack(pady=5)

        # Notes list frame
        note_frame = ttk.Frame(master)
        note_frame.pack(side=tk.LEFT, fill=tk.Y, padx=5, pady=5)
```

```

    ttk.Label(note_frame, text="Notes").pack()
    self.note_listbox = tk.Listbox(note_frame)
    self.note_listbox.pack(fill=tk.Y, expand=True)
    self.note_listbox.bind('<<ListboxSelect>>', self.load_note)

    ttk.Button(note_frame, text="Add Note",
command=self.add_note).pack(pady=5)

    # Editor frame
    editor_frame = ttk.Frame(master)
    editor_frame.pack(side=tk.RIGHT, fill=tk.BOTH, expand=True, padx=5,
pady=5)

    self.text_area = tk.Text(editor_frame, wrap='word')
    self.text_area.pack(fill=tk.BOTH, expand=True)

    ttk.Button(editor_frame, text="Save Note",
command=self.save_note).pack(pady=5)

def add_category(self):
    name = simpledialog.askstring("New Category", "Category name:")
    if name and name not in self.categories:
        self.categories[name] = {}
        self.cat_listbox.insert(tk.END, name)

def load_notes(self, event=None):
    selection = self.cat_listbox.curselection()
    if selection:
        index = selection[0]
        category = self.cat_listbox.get(index)
        self.current_category = category
        self.note_listbox.delete(0, tk.END)
        for note in self.categories[category]:
            self.note_listbox.insert(tk.END, note)

def add_note(self):
    if self.current_category:
        title = simpledialog.askstring("New Note", "Note title:")
        if title and title not in self.categories[self.current_category]:
            self.categories[self.current_category][title] = ""
            self.note_listbox.insert(tk.END, title)
            self.load_note()

def load_note(self, event=None):
    selection = self.note_listbox.curselection()
    if selection:
        index = selection[0]
        title = self.note_listbox.get(index)
        self.current_note = title

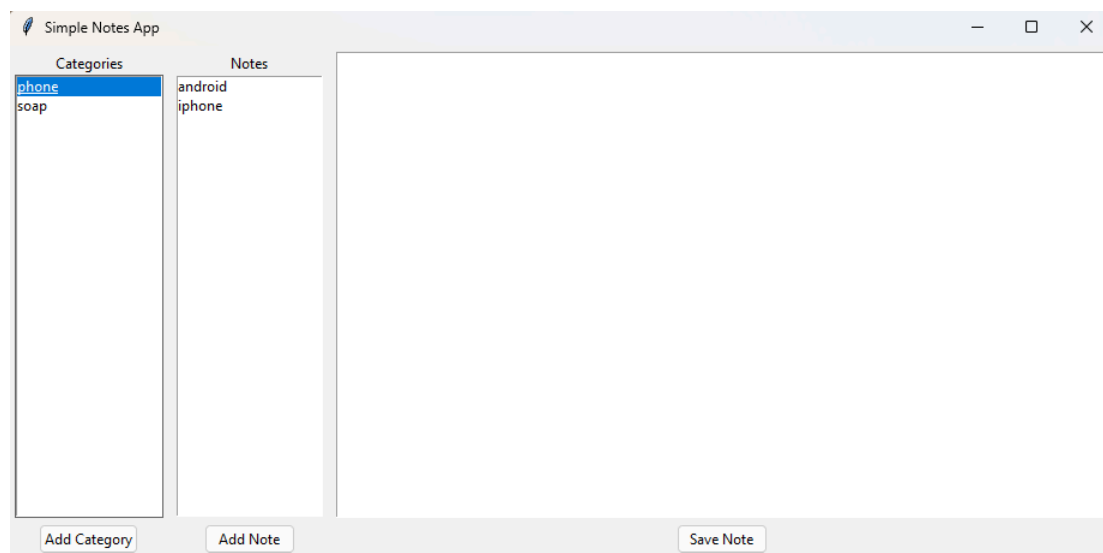
```

```
        content = self.categories[self.current_category][title]
        self.text_area.delete('1.0', tk.END)
        self.text_area.insert(tk.END, content)

    def save_note(self):
        if self.current_category and self.current_note:
            content = self.text_area.get('1.0', tk.END)
            self.categories[self.current_category][self.current_note] = content
            messagebox.showinfo("Saved", "Note saved successfully.")

if __name__ == "__main__":
    root = tk.Tk()
    app = NotesApp(root)
    root.mainloop()
```

Output:



Particulars	Max Marks	Mark secured
Program and Execution	15	
Viva	10	
Total	25	

Result:

A user-friendly GUI app for organizing, editing, and saving categorized notes efficiently.

Ex. No. 5(a)	Develop a Python program to fetch records from a table in MySQL and display the results.
DATE	

Aim:

To Develop a Python program to fetch records from a table in MySQL and display the results.

Algorithm:

Step 1: Start

Step 2: Import the required modules (mysql.connector for MySQL connection). Establish a connection to the MySQL database.

Step 3: Create a cursor object to execute SQL queries.

Step 4: Write an SQL query to fetch records from the desired table.

Step 5: Execute the query using the cursor. Fetch all records from the cursor. Display the fetched records.

Step 6: Stop

Program:

```
import mysql.connector

def fetch_records_from_mysql():
    try:
        # Connect to MySQL database
        connection = mysql.connector.connect(
            host="your_host",
            user="your_username",
            password="your_password",
            database="your_database"
        )

        # Create cursor object
        cursor = connection.cursor()

        # Write SQL query to fetch records from table
        sql_query = "SELECT * FROM your_table"

        # Execute SQL query
        cursor.execute(sql_query)

        # Fetch all records
        records = cursor.fetchall()

        # Display fetched records
        for record in records:
            print(record)
```

```

except mysql.connector.Error as error:
    print("Error while fetching records from MySQL:", error)

finally:
    # Close cursor and connection
    if 'cursor' in locals() and cursor:
        cursor.close()
    if 'connection' in locals() and connection.is_connected():
        connection.close()

# Call the function to fetch and display records
fetch_records_from_mysql()

```

Output :



```

MySQL 8.0 Command Line Client
Welcome to the MySQL monitor.  Commands end with \ or \g.
Your MySQL connection id is 39
Server version: 8.0.30-Debian MySQL Community Server (GPL)

Copyright (c) 2000, 2019, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> use employees;
Database changed
mysql> describe employees;
+-----+-----+-----+-----+-----+
| Field | Type | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| Employee_ID | int(11) | NO | PRI | NULL | auto_increment |
| Employee_Name | varchar(45) | NO | | NULL | |
| Employee_Department_ID | int(11) | NO | MUL | NULL | |
| Employee_Grade_ID | varchar(2) | NO | | A | |
| Employee_Salary | int(12) | NO | | NULL | |
+-----+-----+-----+-----+-----+
5 rows in set (0.00 sec)

mysql>

```

Result:

The program successfully fetches records from the specified table in the MySQL database and displays them.

Ex. No. 5(b)	Develop a Python program to insert records into a table in MySQL using user input and display a confirmation message upon successful insertion.
DATE	

Aim:

To insert user-provided data into a MySQL database table with confirmation.

Algorithm:

Step1: Connect to the MySQL database using appropriate credentials.

Step2: Prompt the user to input record details (e.g., name and age).

Step3: Prepare an SQL INSERT statement with placeholders.

Step4: Execute the INSERT statement with user input data.

Step5: Commit the transaction to save changes.

Step6: Display a success message and close the database connection.

Program:

```
import mysql.connector

# Connect to MySQL
conn = mysql.connector.connect(host='localhost', user='your_user',
password='your_pass', database='your_db')
cursor = conn.cursor()

# Get user input
name = input("Enter name: ")
age = input("Enter age: ")

# Insert record
sql = "INSERT INTO your_table (name, age) VALUES (%s, %s)"
cursor.execute(sql, (name, age))
conn.commit()

print("Record inserted successfully.")

cursor.close()
conn.close()
```

Output:

Enter name: Alice
Enter age: 25
Record inserted successfully.

Particulars	Max Marks	Mark secured
Program and Execution	15	
Viva	10	
Total	25	

Result:

User input is stored in the database, and a confirmation message confirms successful insertion.

Ex. No. 6(a)	Develop a CRUD program using Python-Mysql connectivity
DATE	

Aim:

To write a program in Python to develop a CRUD (Create, Read, Update, Delete) program using Python-MySQL connectivity to interact with a MySQL database.

Algorithm:

Step 1: Start

Step 2: Import the required modules (mysql.connector for MySQL connection). Establish a connection to the MySQL database.

Step 3: Create a cursor object to execute SQL queries. and Write functions for each CRUD operation

Step 4: Implement a menu-driven interface to allow users to choose CRUD operations.

Step 5: Execute the chosen operation based on user input

Step 6: Stop.

Program:

```
import mysql.connector
```

```
def connect_to_mysql():
```

```
    try:
```

```
        # Connect to MySQL database
```

```
        connection = mysql.connector.connect(
```

```
            host="your_host",
```

```
            user="your_username",
```

```
            password="your_password",
```

```
            database="your_database"
```

```
        )
```

```
        return connection
```

```
    except mysql.connector.Error as error:
```

```
        print("Error while connecting to MySQL:", error)
```

```
def create_record(connection, cursor):
```

```
    try:
```

```
        # Get user input for record details
```

```
        name = input("Enter name: ")
```

```
        age = int(input("Enter age: "))
```

```
        # Write SQL query to insert record into table
```

```
        sql_query = "INSERT INTO your_table (name, age) VALUES (%s, %s)"
```

```
        values = (name, age)
```

```
# Execute SQL query
cursor.execute(sql_query, values)
connection.commit()

print("Record created successfully.")

except mysql.connector.Error as error:
    print("Error while creating record:", error)

def read_records(connection, cursor):
    try:
        # Write SQL query to fetch records from table
        sql_query = "SELECT * FROM your_table"

        # Execute SQL query
        cursor.execute(sql_query)

        # Fetch all records
        records = cursor.fetchall()

        # Display fetched records
        for record in records:
            print(record)

    except mysql.connector.Error as error:
        print("Error while reading records:", error)

def update_record(connection, cursor):
    try:
        # Get user input for record ID and updated details
        record_id = int(input("Enter ID of record to update: "))
        new_name = input("Enter new name: ")
        new_age = int(input("Enter new age: "))

        # Write SQL query to update record in table
        sql_query = "UPDATE your_table SET name = %s, age = %s WHERE id = %s"
        values = (new_name, new_age, record_id)

        # Execute SQL query
        cursor.execute(sql_query, values)
        connection.commit()

        print("Record updated successfully.")

    except mysql.connector.Error as error:
        print("Error while updating record:", error)

def delete_record(connection, cursor):
    try:
```

```
# Get user input for record ID to delete
record_id = int(input("Enter ID of record to delete: "))

# Write SQL query to delete record from table
sql_query = "DELETE FROM your_table WHERE id = %s"
values = (record_id,)

# Execute SQL query
cursor.execute(sql_query, values)
connection.commit()

print("Record deleted successfully.")

except mysql.connector.Error as error:
    print("Error while deleting record:", error)

def main():
    connection = connect_to_mysql()
    if connection:
        try:
            cursor = connection.cursor()

            while True:
                print("\nCRUD Operations:")
                print("1. Create Record")
                print("2. Read Records")
                print("3. Update Record")
                print("4. Delete Record")
                print("5. Exit")


                choice = input("Enter your choice (1-5): ")

                if choice == '1':
                    create_record(connection, cursor)
                elif choice == '2':
                    read_records(connection, cursor)
                elif choice == '3':
                    update_record(connection, cursor)
                elif choice == '4':
                    delete_record(connection, cursor)
                elif choice == '5':
                    break
                else:
                    print("Invalid choice. Please enter a number between 1 and 5.")

            finally:
                cursor.close()
                connection.close()
                print("Connection closed.")
```

```
if __name__ == "__main__":  
    main()
```

Output :



The screenshot shows a web application interface for managing employee data. At the top right, there is a green 'Add Employee' button. Below it is a search bar. The main part of the interface is a table with columns: Emp ID, Name, Age, Skills, Address, and Designation. The table contains 10 rows of employee data. To the right of each row, there are three buttons: 'Update' (yellow), 'Delete' (red), and 'Add' (red). At the bottom left, it says 'Showing 1 to 10 of 10 entries'. At the bottom right, there is a pagination control showing 'Previous', '1', and 'Next'.

Emp ID	Name	Age	Skills	Address	Designation			
12	Roy	30	PHP	Delhi, India	Web Developer	Update	Delete	Add
11	Cook	38	PHP	Delhi	Web Developer	Update	Delete	Add
10	Nathan	28	PHP	London	Web Developer	Update	Delete	Add
9	Rout	23	jQuery	Sydney	Web Developer	Update	Delete	Add
8	Dani	28	HTML	Paris	Web Developer	Update	Delete	Add
7	Philo	28	MySQL	Paris	Web Developer	Update	Delete	Add
6	Tom	28	Angular	London	Web Developer	Update	Delete	Add
5	Shylo	35	NodeJS	Tokyo	Programmer	Update	Delete	Add
4	Arun	25	JavaScript	Delhi	Web Developer	Update	Delete	Add
3	Andy	30	jQuery	New Jersey	Web Developer	Update	Delete	Add

Result:

Thus the Users can perform CRUD operations (Create, Read, Update, Delete) on records in the MySQL database using the Python program.

Ex. No. 6(b)	Develop a Python program to manage student records in a MySQL database, allowing users to add, view, update, and delete student information.
DATE	

Aim:

To manage student records in a MySQL database by enabling adding, viewing, updating, and deleting records through a Python program.

Algorithm:

Step1: Establish a connection to the MySQL database and create the students table if it doesn't exist.

Step2: Present a menu to the user to choose between adding, viewing, updating, deleting records, or exiting.

Step3: Based on user choice, prompt for necessary data (e.g., student details or ID).

Step4: Execute the corresponding SQL query (INSERT, SELECT, UPDATE, DELETE) with the provided data.

Step5: Commit the transaction to save changes to the database.

Step6: Repeat until the user chooses to exit, then close the database connection.

Program:

```
import mysql.connector

# Connect to MySQL database
conn = mysql.connector.connect(host='localhost', user='your_user',
password='your_pass', database='your_db')
cursor = conn.cursor()

def create_table():
    cursor.execute("""
        CREATE TABLE IF NOT EXISTS students (
            id INT AUTO_INCREMENT PRIMARY KEY,
            name VARCHAR(100),
            age INT,
            course VARCHAR(100)
        )
    """)
    conn.commit()

def add_student():
    name = input("Enter name: ")
    age = int(input("Enter age: "))
    course = input("Enter course: ")
    cursor.execute("INSERT INTO students (name, age, course) VALUES (%s, %s, %s)", (name, age, course))
    conn.commit()
    print("Student added successfully.")
```

```

def view_students():
    cursor.execute("SELECT * FROM students")
    rows = cursor.fetchall()
    for row in rows:
        print(row)

def update_student():
    student_id = int(input("Enter student ID to update: "))
    name = input("New name: ")
    age = int(input("New age: "))
    course = input("New course: ")
    cursor.execute("UPDATE students SET name=%s, age=%s, course=%s WHERE
id=%s", (name, age, course, student_id))
    conn.commit()
    print("Record updated successfully.")

def delete_student():
    student_id = int(input("Enter student ID to delete: "))
    cursor.execute("DELETE FROM students WHERE id=%s", (student_id,))
    conn.commit()
    print("Record deleted successfully.")

create_table()

while True:
    print("\n1. Add Student\n2. View Students\n3. Update Student\n4. Delete
Student\n5. Exit")
    choice = input("Choose an option: ")
    if choice == '1':
        add_student()
    elif choice == '2':
        view_students()
    elif choice == '3':
        update_student()
    elif choice == '4':
        delete_student()
    elif choice == '5':
        break
    else:
        print("Invalid choice.")

cursor.close()
conn.close()

```


Output :

1. Add Student
2. View Students
3. Update Student
4. Delete Student
5. Exit

Choose an option: 1

Enter name: Alice

Enter age: 20

Enter course: Computer Science

Student added successfully.

1. Add Student
2. View Students
3. Update Student
4. Delete Student
5. Exit

Choose an option: 1

Enter name: Bob

Enter age: 22

Enter course: Electrical

Student added successfully.

1. Add Student
2. View Students
3. Update Student
4. Delete Student
5. Exit

Choose an option: 2

(1, 'Alice', 20, 'Computer Science')

(2, 'Bob', 22, 'Electrical')

1. Add Student
2. View Students
3. Update Student
4. Delete Student
5. Exit

Choose an option: 3

Enter student ID to update: 1

New name: Alice Johnson

New age: 21

New course: Data Science

Record updated successfully.

1. Add Student
2. View Students
3. Update Student
4. Delete Student

5. Exit

Choose an option: 2

(1, 'Alice Johnson', 21, 'Data Science')

(2, 'Bob', 22, 'Electrical')

1. Add Student

2. View Students

3. Update Student

4. Delete Student

5. Exit

Choose an option: 4

Enter student ID to delete: 2

Record deleted successfully.

1. Add Student

2. View Students

3. Update Student

4. Delete Student

5. Exit

Choose an option: 2

(1, 'Alice Johnson', 21, 'Data Science')

1. Add Student

2. View Students

3. Update Student

4. Delete Student

5. Exit

Choose an option: 5

Particulars	Max Marks	Mark secured
Program and Execution	15	
Viva	10	
Total	25	

Result:

The program manages student records in the database, allowing CRUD operations with user interaction.

Ex. No. 7(a)	Create a client server based chat application where multiple clients can connect to a server and exchange message
DATE	

Aim:

To Create a client-server based chat application where multiple clients can connect to a server and exchange messages

Algorithm:

Step 1: Start

Step 2: Set up a server socket to listen for incoming connections.

Step 3: Accept incoming client connections and start a new thread to handle each client. and Allow clients to send messages to the server, which relays them to all other connected clients.

Step 4: Implement a protocol for communication between the server and clients (e.g., using sockets and message encoding/decoding).

Step 5: Implement a GUI for the client to send and receive messages.

Step 6: Stop

Program:

SERVER:

```
import socket
import threading

def handle_client(client_socket, address):
    print(f"Accepted connection from {address}")
    while True:
        message = client_socket.recv(1024).decode()
        if not message:
            print(f"Connection from {address} closed.")
            break
        print(f"Received message from {address}: {message}")
        broadcast(message)

def broadcast(message):
    for client in clients:
        client.send(message.encode())

server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind(('localhost', 5555))
server.listen(5)
print("Server listening on port 5555...")

clients = []
```

```
while True:
    client_socket, address = server.accept()
    clients.append(client_socket)
    client_thread = threading.Thread(target=handle_client, args=(client_socket,
address))
    client_thread.start()
```

CLIENT:

```
import socket
import threading
```

```
def receive_messages():
    while True:
        try:
            message = client_socket.recv(1024).decode()
            print(message)
        except:
            print("An error occurred while receiving messages.")
            break
```

```
def send_messages():
    while True:
        message = input()
        client_socket.send(message.encode())
```

```
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client_socket.connect(('localhost', 5555))
```

```
receive_thread = threading.Thread(target=receive_messages)
receive_thread.start()
```

```
send_thread = threading.Thread(target=send_messages)
send_thread.start()
```

Output :



Result:

The server will display messages received from clients and relay them to all other connected clients. Clients will display messages received from the server and send messages typed by the user.

Ex. No. 7(b)	Create a client server based chat application where multiple clients can connect to a server
DATE	

Aim:

To develop a client-server chat application using Python sockets that allows multiple clients to communicate with each other in real-time through a central server.

Algorithm:

Step1: Start the Server: Initialize socket, bind to a host and port, and listen for incoming connections.

Step2: Accept Clients: When a client connects, accept the connection and request a nickname.

Step3: Store Clients: Add connected clients and their nicknames to lists for tracking.

Step4: Start Client Threads: Launch a new thread for each client to handle message receiving and broadcasting.

Step5: Broadcast Messages: Relay incoming messages from one client to all others, excluding the sender.

Step6: Handle Disconnection: Remove a client from the list if disconnected and notify remaining clients.

Program:

SERVER:

```
import socket
import threading

# Server config
HOST = '127.0.0.1' # localhost
PORT = 12345

server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind((HOST, PORT))
server.listen()

clients = []
nicknames = []

def broadcast(message, sender_client=None):
    for client in clients:
        if client != sender_client:
            client.send(message)
```

```
def handle(client):
    while True:
        try:
            message = client.recv(1024)
            broadcast(message, client)
        except:
            index = clients.index(client)
            clients.remove(client)
            client.close()
            nickname = nicknames[index]
            broadcast(f"{nickname} left the chat.".encode('utf-8'))
            nicknames.remove(nickname)
            break

def receive():
    print(f"Server is running on {HOST}:{PORT}...")
    while True:
        client, address = server.accept()
        print(f"Connected with {str(address)}")

        client.send("NICKNAME".encode('utf-8'))
        nickname = client.recv(1024).decode('utf-8')
        nicknames.append(nickname)
        clients.append(client)

        print(f"Nickname is {nickname}")
        broadcast(f"{nickname} joined the chat!".encode('utf-8'))
        client.send("Connected to the server!".encode('utf-8'))

        thread = threading.Thread(target=handle, args=(client,))
        thread.start()
```

receive()

CLIENT:

```
import socket
import threading
```

```
# Server config (match with server.py)
HOST = '127.0.0.1'
PORT = 12345
```

```
nickname = input("Choose your nickname: ")
```

```
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client.connect((HOST, PORT))
```

```
def receive():
```



```
while True:
    try:
        message = client.recv(1024).decode('utf-8')
        if message == "NICKNAME":
            client.send(nickname.encode('utf-8'))
        else:
            print(message)
    except:
        print("An error occurred!")
        client.close()
        break

def write():
    while True:
        msg = f"{nickname}: {input()}"
        client.send(msg.encode('utf-8'))

# Start threads
receive_thread = threading.Thread(target=receive)
receive_thread.start()

write_thread = threading.Thread(target=write)
write_thread.start()
```

Output:

SERVER

```
Server is running on 127.0.0.1:12345...
Connected with ('127.0.0.1', 53422)
Nickname is Alice
Connected with ('127.0.0.1', 53423)
Nickname is Bob
```

CLIENT

```
Choose your nickname: Alice
Connected to the server!
Bob joined the chat!
Alice: Hello everyone!
Bob: Hi Alice!
```

Particulars	Max Marks	Mark secured
Program and Execution	15	
Viva	10	
Total	25	

Result:

Multiple clients successfully connected to the server and exchanged messages in real-time.

Ex. No. 8(a)	Build a network port scanner program that scans a given IP addresses to detect open ports on remote machine
DATE	

Aim:

To Build a network port scanner program that scans a given IP address to detect open ports on a remote machine.

Algorithm:

Step 1: Start

Step 2: Import the necessary modules (socket for socket operations). and Define a function to scan ports on a given IP address.

Step 3: Use a loop to iterate over a range of port numbers

Step 5: Attempt to connect to each port using the socket.connect() method. and If the connection is successful, the port is open, otherwise it's closed.

Step 4: Stop the process

Program:

```
import socket
```

```
def scan_ports(target_ip, start_port, end_port):
    print(f"Scanning ports on {target_ip}...")
    for port in range(start_port, end_port + 1):
        try:
            # Create a socket object
            client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            # Set a timeout for the connection attempt
            client_socket.settimeout(1)
            # Attempt to connect to the target IP and port
            result = client_socket.connect_ex((target_ip, port))
            if result == 0:
                print(f"Port {port}: Open")
            client_socket.close()
        except Exception as e:
            print(f"Error occurred while scanning port {port}: {e}")
```

```
# Example usage
```

```
target_ip = "192.168.1.1" # Example IP address to scan
```

```
start_port = 1
```

```
end_port = 1000
```

```
scan_ports(target_ip, start_port, end_port)
```

Output :

Scanning ports on 192.168.1.1...

Port 22: Open

Port 80: Closed

Port 443: Open

Port 3389: Closed

Result:

Thus the port scanner program successfully scans the specified IP address for open ports and reports the results.

Ex. No. 8(b)	Develop a network vulnerability scanner program that not only detects open ports but also checks for known vulnerabilities on those ports.
DATE	

Aim:

To develop a Python-based network vulnerability scanner that detects open ports on a target machine and checks for known vulnerabilities on those ports.

Algorithm:

Step1: Accept target IP input from the user.

Step2: Initialize port scanning on common ports (1–1024) using socket connections.

Step3: Attempt to connect to each port with a timeout to detect if it's open.

Step4: Store open ports in a list for further analysis.

Step5: Match open ports with a local database of known vulnerabilities (CVE list).

Step6: Display results showing open ports and associated known vulnerabilities.

Program:

```
import socket
import threading
```

```
# Example CVE database (simplified)
```

```
known_vulnerabilities = {
    21: ["FTP service vulnerable to anonymous login (CVE-1999-0497)"],
    22: ["Old OpenSSH version (CVE-2007-3102)"],
    23: ["Telnet service unencrypted (CVE-1999-0617)"],
    80: ["Apache 2.2 vulnerable to DoS (CVE-2011-3192)"],
    443: ["SSL BEAST attack vulnerability (CVE-2011-3389)"],
    3306: ["MySQL authentication bypass (CVE-2012-2122)"]
}
```

```
open_ports = []
```

```
def scan_port(ip, port):
    try:
        sock = socket.socket()
        sock.settimeout(1)
        sock.connect((ip, port))
        open_ports.append(port)
        sock.close()
    except:
        pass
```

```
def main():
```

```
target = input("Enter target IP address: ")
print(f"Scanning {target} for open ports and known vulnerabilities...")

threads = []
for port in range(1, 1025):
    t = threading.Thread(target=scan_port, args=(target, port))
    threads.append(t)
    t.start()

for t in threads:
    t.join()

print("\nOpen Ports Found:")
for port in open_ports:
    print(f"Port {port} is open.")
    if port in known_vulnerabilities:
        for vuln in known_vulnerabilities[port]:
            print(f" ⚠ Vulnerability: {vuln}")
    else:
        print(" No known vulnerabilities in local database.")

if __name__ == "__main__":
    main()
```

Output

```
Enter target IP address: 127.0.0.1
Scanning 127.0.0.1 for open ports and known vulnerabilities...

Open Ports Found:
Port 22 is open.
⚠ Vulnerability: Old OpenSSH version (CVE-2007-3102)
Port 80 is open.
⚠ Vulnerability: Apache 2.2 vulnerable to DoS (CVE-2011-3192)
```

Particulars	Max Marks	Mark secured
Program and Execution	15	
Viva	10	
Total	25	

Result:

The scanner successfully identified open ports and listed known vulnerabilities for each, if any, based on local CVE data.

Ex. No. 9(a)	Create a fully functional blogging platform where users register, create blog posts, add comments and browse through published posts
DATE	

Aim:

To Create a fully functional blogging platform where users can register, create blog posts, add comments, and browse through published posts.

Algorithm:

Step1: Design a database schema to store user accounts, blog posts, and comments.

Step2: Implement user registration functionality with features like username, password, and email validation.

Step3: Implement authentication and session management to secure user accounts.

Step4: Develop CRUD (Create, Read, Update, Delete) operations for blog posts, allowing users to create, view, update, and delete their own posts.

Step5: Implement a comment system where users can add comments to published blog posts.

Step6: Develop a user-friendly interface for browsing through published posts and viewing comments.

Step7: Ensure data integrity and security measures to protect user data and prevent unauthorized access.

Step8: Test the platform thoroughly to identify and fix any bugs or issues

Program:

Example of simplified code

```
from flask import Flask, render_template, request, redirect, url_for
from flask_sqlalchemy import SQLAlchemy
```

```
app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///blog.db'
db = SQLAlchemy(app)
```

Database Models

```
class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(100), unique=True, nullable=False)
    password = db.Column(db.String(100), nullable=False)
    email = db.Column(db.String(100), unique=True, nullable=False)
```

```
class Post(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    title = db.Column(db.String(100), nullable=False)
```

```
content = db.Column(db.Text, nullable=False)
user_id = db.Column(db.Integer, db.ForeignKey('user.id'), nullable=False)
user = db.relationship('User', backref=db.backref('posts', lazy=True))

class Comment(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    text = db.Column(db.Text, nullable=False)
    post_id = db.Column(db.Integer, db.ForeignKey('post.id'), nullable=False)
    post = db.relationship('Post', backref=db.backref('comments', lazy=True))

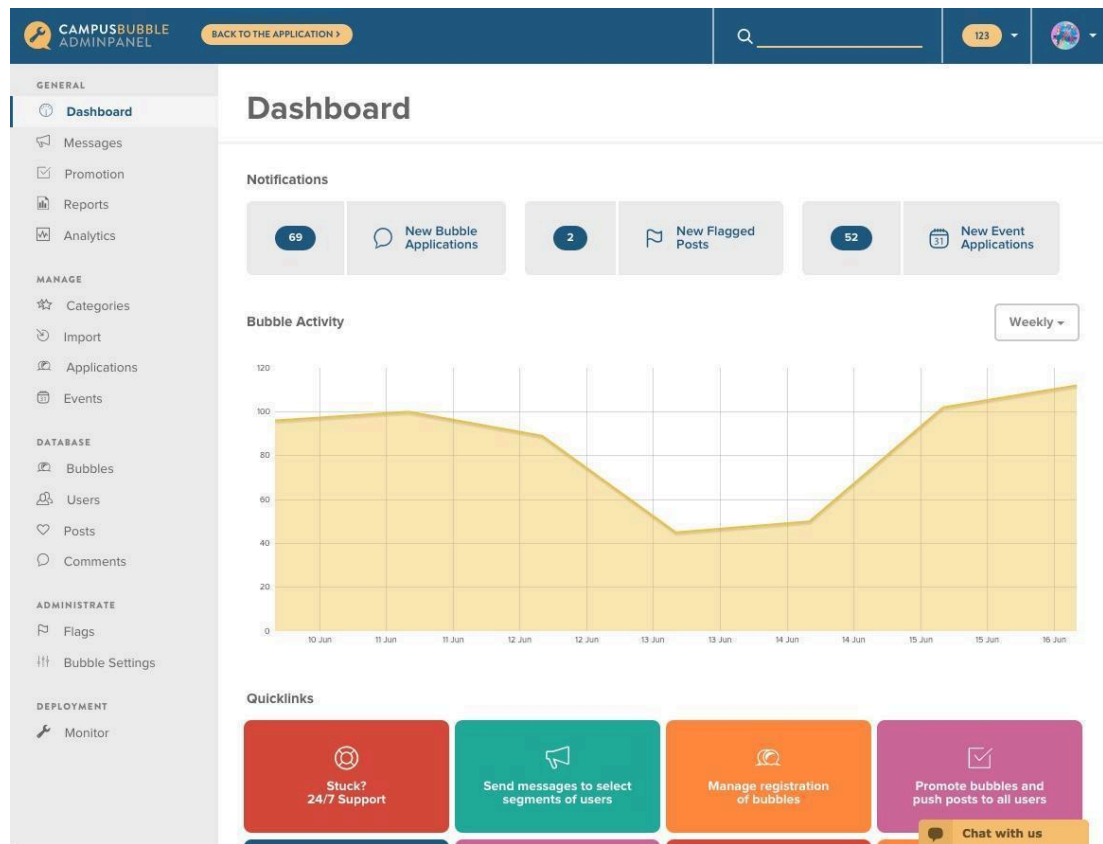
# Routes
@app.route('/')
def index():
    posts = Post.query.all()
    return render_template('index.html', posts=posts)

@app.route('/post/<int:post_id>')
def post(post_id):
    post = Post.query.get_or_404(post_id)
    return render_template('post.html', post=post)

# Other routes (register, login, create_post, add_comment, etc.) would be
implemented similarly

if __name__ == '__main__':
    db.create_all()
    app.run(debug=True)
```

Output:



Result:

The fully functional blogging platform allows users to engage with the community by sharing their thoughts through blog posts and comments. It

provides a user-friendly interface for content creation, interaction, and exploration.

Ex. No. 9(b)	Develop a social media platform where users can create profiles, post updates, connect with friends, and engage with content through likes and comments.
DATE	

Aim:

To develop a full-stack social media platform where users can create profiles, post updates, connect with friends, and interact via likes and comments.

Algorithm:

- Step1:** User registration/login to authenticate and create profiles.
- Step2:** Store user and post data using a database (SQLite/MySQL).
- Step3:** Allow users to submit posts, which are saved and shown in the feed.
- Step4:** Display all posts on the homepage or feed for logged-in users.
- Step5:** Enable interaction by allowing users to like posts or add comments.
- Step6:** Establish friend connections to show posts only from friends (optional feature).

Program:

```
from flask import Flask, render_template, request, redirect, session
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
app.secret_key = 'secret123'
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///database.db'
db = SQLAlchemy(app)

class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(100), unique=True)
    password = db.Column(db.String(100))
    bio = db.Column(db.String(300))

class Post(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    user_id = db.Column(db.Integer)
    content = db.Column(db.String(300))
    likes = db.Column(db.Integer, default=0)

@app.route('/', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        user = User.query.filter_by(username=request.form['username']).first()
```

```
        if user and user.password == request.form['password']:
            session['user_id'] = user.id
            return redirect('/feed')
    return render_template('login.html')

@app.route('/feed')
def feed():
    posts = Post.query.all()
    user = User.query.get(session['user_id'])
    return render_template('feed.html', posts=posts, user=user)

@app.route('/post', methods=['POST'])
def post():
    new_post = Post(user_id=session['user_id'], content=request.form['content'])
    db.session.add(new_post)
    db.session.commit()
    return redirect('/feed')

@app.route('/like/<int:post_id>')
def like(post_id):
    post = Post.query.get(post_id)
    post.likes += 1
    db.session.commit()
    return redirect('/feed')

if __name__ == "__main__":
    db.create_all()
    app.run(debug=True)
```

Output:

[Login Page]
Username: alice
Password: *****

[Submit]

Particulars	Max Marks	Mark secured
Program and Execution	15	
Viva	10	
Total	25	

Result:

Users successfully registered, created posts, and interacted with others' content through likes.

Ex. No.10(A)	Develop an e-commerce store application with features like product listings , user authentication , shopping cart management and secure payment integration
Date:	

Aim:

Develop an e-commerce store application with features like product listings, user authentication, shopping cart management, and secure payment integration.

Algorithm:

Step 1: Start

Step 2: Design a database schema to store product details, user accounts, and shopping cart items.

Step 3: Implement user authentication functionality, including registration, login, and session management.

Step 4: Develop CRUD operations for managing products, allowing administrators to add, edit, and delete products.

Step 5: Implement shopping cart functionality, allowing users to add products to their cart, view their cart, and update or remove items.

Step 6: Integrate a secure payment gateway for processing transactions, ensuring the confidentiality and integrity of payment information.

Step 7 : Implement product listings, displaying available products with details such as name, description, price, and image.

Step 8: Develop user-friendly interfaces for browsing products, managing shopping carts, and completing orders.

Step 9: Implement security measures such as encryption and validation to protect user data and prevent unauthorized access.

Step 10 : Stop

Program:

```
# Example of simplified code using Flask framework and SQLite database

from flask import Flask, render_template, request, redirect, url_for, session, flash
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
app.secret_key = 'your_secret_key'
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///store.db'
db = SQLAlchemy(app)

# Database Models
class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
```



```
username = db.Column(db.String(100), unique=True, nullable=False)
password = db.Column(db.String(100), nullable=False)

class Product(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(100), nullable=False)
    description = db.Column(db.Text, nullable=False)
    price = db.Column(db.Float, nullable=False)

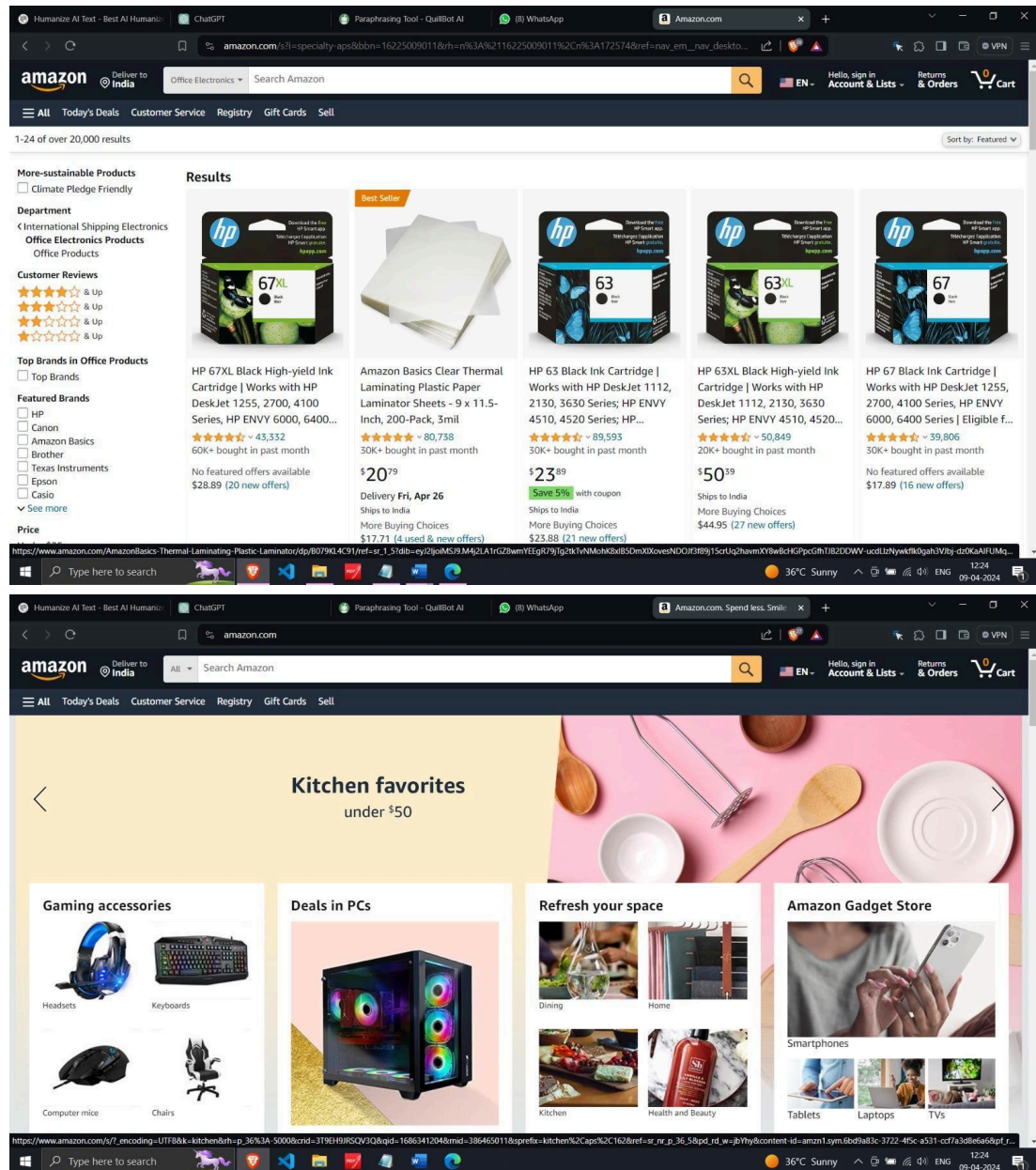
class CartItem(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    product_id = db.Column(db.Integer, db.ForeignKey('product.id'),
nullable=False)
    user_id = db.Column(db.Integer, db.ForeignKey('user.id'), nullable=False)
    quantity = db.Column(db.Integer, nullable=False)

# Routes
@app.route('/')
def index():
    products = Product.query.all()
    return render_template('index.html', products=products)

# Other routes (register, login, add_to_cart, view_cart, checkout, etc.) would
be implemented similarly

if __name__ == '__main__':
    db.create_all()
    app.run(debug=True)
```

Output:



Result:

Thus The fully functional e-commerce store application provides users with a seamless shopping experience, allowing them to explore products, manage their shopping cart, and securely complete transactions.

Ex. No. 10(B)	Develop a ticket booking platform where users can browse available events, select seats, and purchase tickets securely.
Date:	

Aim:

To develop a ticket booking platform in Python using Flask where users can browse events, select seats, and book tickets.

Algorithm:

- Step1:** Initialize Flask app and configure database (SQLite).
- Step2:** Define database models for User, Event, and Ticket.
- Step3:** Create user registration and login with password hashing.
- Step4:** Display list of events on homepage.
- Step5:** Show event details with available seats.
- Step6:** Allow user to select and book a seat.
- Step7:** Save ticket and update booked seats in the database.
- Step8:** Display booking confirmation to the user.

Program:

```

from flask import Flask, render_template, request, redirect, url_for, flash
from flask_login import LoginManager, login_user, login_required, logout_user,
current_user
from flask_bcrypt import Bcrypt
from models import db, User, Event, Ticket

app = Flask(__name__)
app.config['SECRET_KEY'] = 'secretkey'
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///booking.db'

db.init_app(app)
bcrypt = Bcrypt(app)
login_manager = LoginManager(app)
login_manager.login_view = 'login'

@login_manager.user_loader
def load_user(user_id):
    return User.query.get(int(user_id))

@app.route('/')
def index():
    events = Event.query.all()
    return render_template('index.html', events=events)

@app.route('/register', methods=['GET', 'POST'])
def register():
    if request.method == 'POST':

```

```

        hashed_pw =
bcrypt.generate_password_hash(request.form['password']).decode('utf-8')
        user = User(username=request.form['username'], password=hashed_pw)
        db.session.add(user)
        db.session.commit()
        return redirect(url_for('login'))
    return render_template('register.html')

@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        user = User.query.filter_by(username=request.form['username']).first()
        if user and bcrypt.check_password_hash(user.password,
request.form['password']):
            login_user(user)
            return redirect(url_for('index'))
        flash('Login failed.')
    return render_template('login.html')

@app.route('/event/<int:event_id>', methods=['GET', 'POST'])
@login_required
def event_detail(event_id):
    event = Event.query.get_or_404(event_id)
    if request.method == 'POST':
        seat = int(request.form['seat'])
        if seat in event.seats_booked or seat > event.seats_total:
            flash('Seat already booked or invalid.')
        else:
            event.seats_booked.append(seat)
            ticket = Ticket(user_id=current_user.id, event_id=event.id,
seat_number=seat)
            db.session.add(ticket)
            db.session.commit()
            flash('Ticket booked!')
    return render_template('event_detail.html', event=event)


@app.route('/logout')
def logout():
    logout_user()
    return redirect(url_for('index'))

@app.cli.command('initdb')
def initdb():
    db.create_all()
    print("Database initialized.")


```

Output:

Ticket Booking

[Sign in](#)


Events




Concert One

Music - The Arena
May 25, 2024

Stage




Select Seats




Comedy Show

Comedy - City Theater - Jun19, 202..

Stage




Select Seats




Sports Event

Sports - Stadium
Jul 4, 2024

Stage




Select Seats



Theater Performance

Theater - Grand Hall
Aug 15, 2024

Stage



Select Seats

Particulars	Max Marks	Mark secured
Program and Execution	15	
Viva	10	
Total	25	

Result:

User successfully books a seat for an event and receives a ticket confirmation.