CIA III-Assignment

Python programming-CSC 351

Design and Develop a PYTHON based GUI project

THEJAS C
(2240151)

3 CME

For Academic Year 2023-24

SUBMITTED TO
Dr MANJUNATHA HIREMATH

Department of Computer Science

Christ (Deemed to be University) Bangalore

Introduction to Tkinder

Creating graphical user interfaces (GUIs) is an essential aspect of modern software development, enabling developers to design visually appealing and user-friendly applications. One of the popular tools for building GUI projects in Python is the T Kinter library. T Kinter provides a simple and efficient way to create interactive applications with graphical elements, such as buttons, menus, and windows. Whether you're a seasoned developer or a newcomer to the world of programming, T Kinter offers a user-friendly approach to building dynamic and visually engaging applications, making it an ideal choice for projects ranging from simple utilities to complex software applications.

Application of Tkinder

- **Desktop Applications:** T Kinter is ideal for building standalone desktop apps like text editors, calculators, and image viewers.
- **Data Visualization:** It's used for creating data visualization tools and interactive charts.
- **Configuration Tools:** T Kinter is common in system configuration utilities.
- **Educational Software:** It's used for educational programs and quiz applications.
- **Database Interfaces:** T Kinter is used in applications that interact with databases.
- **IoT Control Panels:** It's suitable for building control interfaces for IoT devices.
- **Games:** While not as common, T Kinter can be used for simple games.

Domain (Transportation management)

Transportation management in the realm of Python programming involves the utilization of computational tools and algorithms to optimize and streamline the movement of goods, people, or information from one location to another. This domain integrates various aspects of logistics, supply chain management, and operational efficiency, aiming to enhance the transportation process.

Python, as a versatile and powerful programming language, offers a range of libraries, frameworks, and tools that can be leveraged to address the complexities of transportation management. Through Python, developers can design and implement solutions for route optimization, vehicle scheduling, fleet management, predictive analysis, and other critical facets of transportation logistics.

The applications of Python in transportation management are diverse, catering to industries such as shipping, e-commerce, public transit, and supply chain operations. By employing Python's capabilities in data analysis, machine learning, and algorithm development, professionals can create robust systems to minimize transportation costs, reduce delivery times, and enhance overall operational efficiency.

This combination of transportation management principles with Python programming facilitates the development of sophisticated, data-driven solutions that address the evolving needs and challenges within the transportation industry. As a result, Python has become a pivotal tool in optimizing logistics, improving decision-making, and ultimately contributing to more effective and efficient transportation systems.

DatA Set

1.Route: In transportation, a route refers to the path taken from an origin to a destination. This can involve various factors like road networks, traffic conditions, distance, and time. Example functionalities related to routes in code might include:

Route Planning: Algorithms that calculate the most efficient path from one point to another, considering factors like traffic conditions, road types, and distances.

Route Optimization: Methods to optimize a route for minimal time, fuel consumption, or cost, often employing algorithms.

2.Vehicle: Vehicles are the means of transportation used to move goods or people. In code, vehicle-related functionalities might involve:

Fleet Management: Handling a fleet of vehicles, including maintenance schedules, tracking locations, and managing available capacity.

Load Optimization: Algorithms or methods to maximize the usage of a vehicle's capacity while considering weight limits and space.

3.Driver: The individual responsible for operating the vehicle. Code functionalities related to drivers might include:

Driver Assignment: Systems that assign drivers to specific vehicles or routes based on availability, skills, or regulations.

Driver Performance Tracking: Utilizing data to monitor and assess driver performance, including factors like adherence to schedules, safety, and fuel efficiency.

4.Shipment: Refers to the consignment of goods being transported from one place to another. In code, functionalities might involve tracking shipments, managing inventory, and ensuring timely delivery.

5.Traffic Data: Information related to traffic conditions, congestion, or road closures. Code functionalities could encompass collecting real-time traffic data, analyzing it, and making decisions based on these insights.

6.Scheduling: Involves planning the timing and sequence of transportation operations, including vehicle departures, arrivals, and deliveries. In code, scheduling functionalities could optimize timelines and coordinate multiple operations.

7.Geocoding and Mapping: The process of converting addresses into geographic coordinates and representing them on maps. Code functionalities may involve geocoding addresses and visualizing routes or locations on maps.

8.Transportation APIs: Application Programming Interfaces that allow interaction with transportation-related services or data sources. Code functionalities might involve integrating APIs for services like route calculation, traffic updates, or geospatial data retrieval.

These keywords and functionalities form the foundation for building software solutions in transportation management, enabling efficient, data-driven decision-making and optimization within the industry.

Modules

Modules used in the program

Pandas: Pandas is a powerful library for data manipulation and analysis. It is extensively used for handling diverse datasets in transportation management, such as logs of transportation activities, customer information, and supply chain data. Pandas allows for data cleaning, transformation, and statistical analysis.

GeoPandas: GeoPandas extends the Pandas library to work seamlessly with geospatial data. It's instrumental in managing and analyzing geographical datasets, such as information about road networks, traffic flow, and infrastructure details.

NumPy: NumPy is fundamental for numerical computations and array operations. In transportation management, it aids in performing mathematical operations on large datasets, enabling efficient handling and manipulation of numeric information.

Matplotlib and Seaborn: These libraries are used for data visualization. They help create various types of plots and charts, making it easier to visualize transportation-related data, such as traffic patterns, route optimizations, and shipment statistics.

Scikit-learn: This library offers a range of machine learning algorithms. In transportation management, Scikit-learn can be applied for tasks like predictive analysis, such as predicting traffic conditions or optimizing routes based on historical data.

Network X: For those dealing with network analysis or route optimization, Network X is valuable. It allows the creation, manipulation, and study of complex networks, making it suitable for applications involving transportation networks, such as finding shortest paths or analysing network structures.

Optimization Libraries (such as PuLP, CVXPY): These libraries are used for solving optimization problems. In transportation management, these are applied to optimize routes, schedules, or resource allocation, aiming to minimize costs or travel time while meeting specific constraints.

TensorFlow and PyTorch: For those involved in more advanced tasks like predictive modelling, deep learning, or developing AI-based transportation solutions, these deep learning frameworks are used for tasks like predicting traffic patterns or optimizing vehicle routing based on complex patterns.

These modules, among others, offer a rich ecosystem within Python for addressing the challenges in transportation management. They empower professionals to handle, analyse, and derive insights from transportation-related data, ultimately aiding in the optimization and efficient management of transportation systems.

Operations

In the realm of transportation management, various operations and programming concepts are crucial for efficient logistics and transportation optimization. Here's a description of these concepts without specific reference to box details:

Adding Items: Adding items in transportation management involves the incorporation of new packages or shipments into the system. This operation entails recording essential information such as dimensions, weight, sender, receiver details, and unique identifiers for efficient tracking and inventory management.

Deleting Items: Deleting items from the system refers to the removal of specific packages or shipments. It requires identification of the item to be removed and updating associated data such as inventory records, delivery status, or route planning.

Viewing Item Information: Viewing item information involves accessing details about existing packages within the transportation system. This operation allows users to retrieve information such as current location, delivery status, and other relevant data.

Indexing Items: Indexing is fundamental for referencing or accessing specific elements within a dataset. In transportation management, indexing allows identification and retrieval of specific items by their unique identifiers or positions within the system's database.

Loop Concepts in Transportation Management:

a. Iterating Through Items: Utilizing loops to iterate through a collection of shipment items, enabling tracking, updating statuses, or processing multiple items concurrently.

b. Status Update Loops: Continuous monitoring and updating of the status of shipment items. These loops ensure effective management of shipment tracking and real-time visibility throughout the transportation process.

c. Error-Handling Loops: Loops designed for monitoring anomalies or issues, triggering corrective actions, alerts, or rerouting strategies to ensure the smooth flow of logistics.

d. Optimization Loops: Iteratively evaluating and optimizing transportation operations by assessing routes, logistics strategies, or scheduling algorithms to improve efficiency, reduce costs, and enhance delivery times.

These operations and programming concepts are integral to transportation management systems, facilitating streamlined logistics operations, informed decision-making, and optimized transportation processes without focusing specifically on the details of individual packages or boxes.

CODE

```
import tkinter as tk
from tkinter import ttk, messagebox
import sqlite3

# Create a SQLite database
conn = sqlite3.connect('transportation_data.db')
cursor = conn.cursor()
cursor.execute('''
CREATE TABLE IF NOT EXISTS routes (
id INTEGER PRIMARY KEY,
source TEXT,
destination TEXT,
distance REAL,
cost REAL
)
''')
conn.commit()

class TransportationManagement:
    def __init__(self, root):
        self.root = root
        self.root.title("Transportation Route Management")
        self.route_list = []
        self.create_widgets()
        self.load_data()

    def create_widgets(self):
        self.label = tk.Label(self.root, text="Transportation
Routes", fg="blue")
        self.label.pack()

        self.tree = ttk.Treeview(self.root, columns=("Source",
"Destination", "Distance", "Cost"))
        self.tree.heading("#1", text="Source")
        self.tree.heading("#2", text="Destination")
        self.tree.heading("#3", text="Distance")
        self.tree.heading("#4", text="Cost")
        self.tree.pack()

        self.refresh_button = tk.Button(self.root, text="Refresh
Routes", command=self.load_data, bg="lightgray")
        self.refresh_button.pack()

        self.add_button = tk.Button(self.root, text="Add Route",
command=self.add_route, bg="green")
        self.add_button.pack()
        self.delete_button = tk.Button(self.root, text="Delete
Route", command=self.delete_route, bg="red")
        self.delete_button.pack()

        self.search_label = tk.Label(self.root, text="Search
Source/Destination:", fg="green")
        self.search_label.pack()
        self.search_entry = tk.Entry(self.root)
        self.search_entry.pack()
        self.search_button = tk.Button(self.root, text="Search",
command=self.search_route, bg="blue")
        self.search_button.pack()

    def load_data(self):
        cursor.execute("SELECT * FROM routes")
        self.route_list = cursor.fetchall()
        for item in self.tree.get_children():
            self.tree.delete(item)
        for route in self.route_list:
            self.tree.insert("", "end", values=route[1:])

    def add_route(self):
        add_window = tk.Toplevel(self.root)
        add_window.title("Add Route")
    source_label = tk.Label(add_window, text="Source:",
fg="blue")
    source_label.pack()
    source_entry = tk.Entry(add_window)
    source_entry.pack()

    dest_label = tk.Label(add_window, text="Destination:",
fg="blue")
    dest_label.pack()
    dest_entry = tk.Entry(add_window)
    dest_entry.pack()

    distance_label = tk.Label(add_window, text="Distance:",
fg="blue")
    distance_label.pack()
    distance_entry = tk.Entry(add_window)
    distance_entry.pack()

    cost_label = tk.Label(add_window, text="Cost:",
fg="blue")
    cost_label.pack()
    cost_entry = tk.Entry(add_window)
    cost_entry.pack()
```

```
        add_button = tk.Button(add_window, text="Add", command=lambda: self.insert_route(
            source_entry.get(), dest_entry.get(), distance_entry.get(), cost_entry.get()), bg="green")
        add_button.pack()

    def insert_route(self, source, dest, distance, cost):
        try:
            distance = float(distance)
            cost = float(cost)
            cursor.execute("INSERT INTO routes (source, destination, distance, cost) VALUES (?, ?, ?, ?)",
                           (source, dest, distance, cost))
            conn.commit()
            self.load_data()
        except ValueError:
            messagebox.showerror("Error", "Distance and Cost must be numeric.")

    def delete_route(self):
        selected_item = self.tree.selection()
        if selected_item:
            route_id = self.route_list[self.tree.index(selected_item[0])][0]
            cursor.execute("DELETE FROM routes WHERE
id=?", (route_id,))
            conn.commit()
            self.load_data()

    def search_route(self):
        keyword = self.search_entry.get()
        if keyword:
            filtered_routes = [route for route in self.route_list if
keyword.lower() in ' '.join(map(str, route)).lower()]
            for item in self.tree.get_children():
                self.tree.delete(item)
            for route in filtered_routes:
                self.tree.insert("", "end", values=route[1:])

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

# Close the database connection when done
conn.close()
```

Data set examples

```
# Dictionary example for transportation data
transportation_dict = {
    1: {'source': 'City A', 'destination': 'City B', 'distance': 200,
'cost': 150},
    2: {'source': 'City B', 'destination': 'City C', 'distance': 300,
'cost': 250},
    3: {'source': 'City A', 'destination': 'City C', 'distance': 400,
'cost': 300},
    # Additional entries can be added
}

# Tuple example for transportation data
transportation_tuple = (
    ('City A', 'City B', 200, 150),
    ('City B', 'City C', 300, 250),
    ('City A', 'City C', 400, 300),
    # Additional entries can be added
)
```

Object-Oriented Programming (OOP) with Inheritance

```
# Example of OOP with inheritance for vehicles
class Vehicle:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year

    def show_info(self):
        return f"Vehicle: {self.make} {self.model} ({self.year})"

class Car(Vehicle):
    def __init__(self, make, model, year, doors):
        super().__init__(make, model, year)
        self.doors = doors

    def show_info(self):
        return f"Car: {self.make} {self.model} ({self.year}) - 
{self.doors} doors"

class ElectricCar(Car):
    def __init__(self, make, model, year, doors, battery):
        super().__init__(make, model, year, doors)
        self.battery = battery
    def show_info(self):
        return f"Electric Car: {self.make} {self.model} 
({self.year}) - {self.doors} doors, {self.battery} kWh battery"
```

Regular Expression Operation

```
import re
# Target string in the transportation domain
target_string = "Route: City A to City B, Distance: 200 miles, Cost: $150"
# RE pattern for finding distance (numeric value)
pattern = r'Distance: (\d+) miles'
# Methods using regular expressions
compiled_pattern = re.compile(pattern)
match_result = compiled_pattern.match(target_string)
findall_result = compiled_pattern.findall(target_string)
group_result = match_result.group(1) if match_result else None
split_result = compiled_pattern.split(target_string)
sub_result = compiled_pattern.sub("Distance: XXX miles", target_string)
```

Importing Functions and Variables

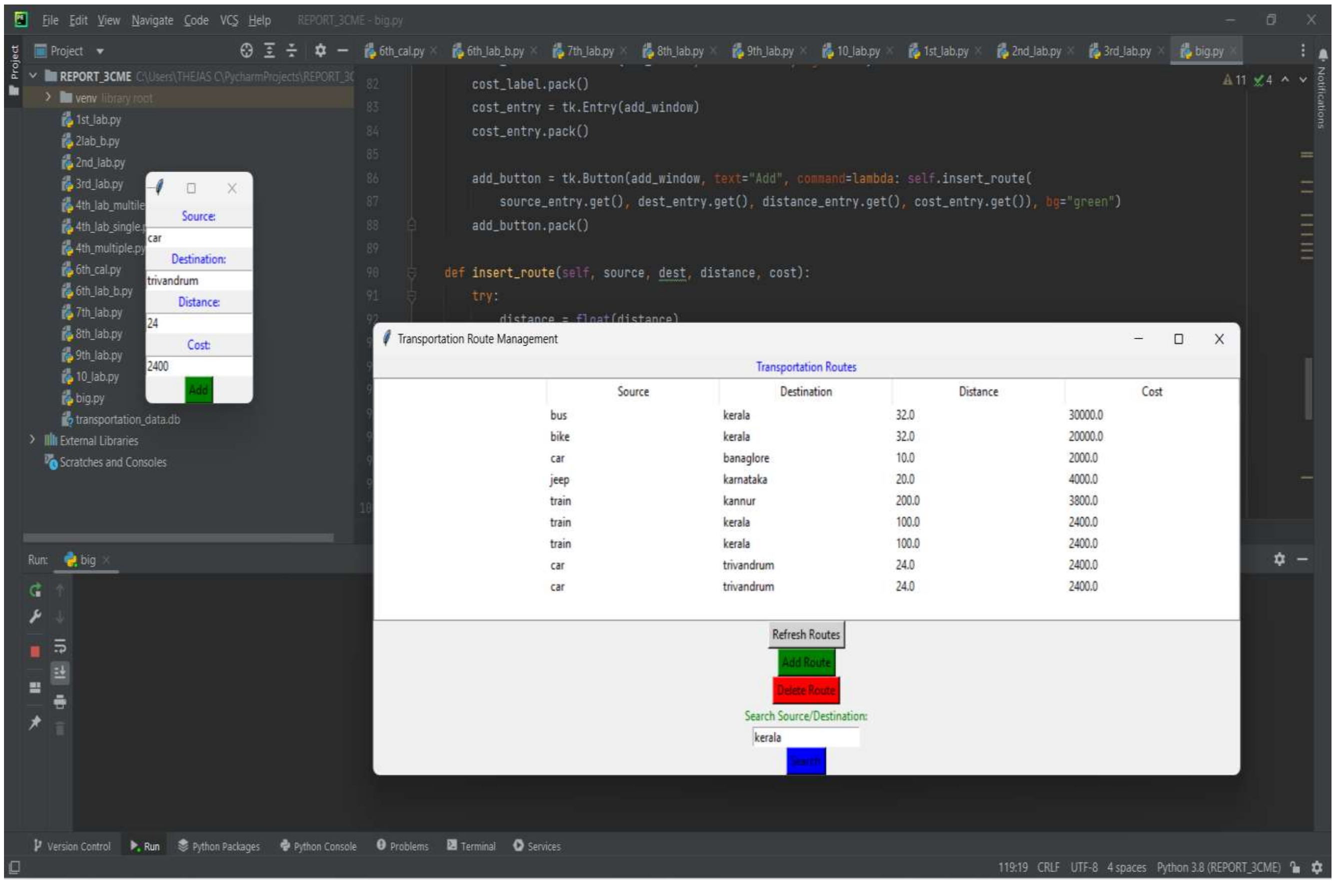
```
# Module-based approach
# Save the above OOP and regular expression code in a file, e.g., transportation_operations.py
# In the main Python script:
import transportation_operations
vehicle = transportation_operations.Vehicle("Toyota", "Camry", 2022)
```

```
car = transportation_operations.Car("Honda", "Accord", 2020,
4)
electric_car = transportation_operations.ElectricCar("Tesla",
"Model S", 2023, 4, 100)

print(vehicle.show_info())
print(car.show_info())
print(electric_car.show_info())

re_result =
transportation_operations.compiled_pattern.findall(transportat
ion_operations.target_string)
print(re_result)
```

Code Description

The provided code demonstrates a program for managing transportation routes using Python's Tkinter for the graphical interface and SQLite for database handling. The script initializes a SQLite database, creating a table named routes with columns such as id, source, destination, distance, and cost. The Transportation Management class organizes the GUI and operations related to the transportation routes. Upon initialization, it sets up the main window, creating labels, buttons, and a tree view for presenting the route data. This class features functions like load_data to fetch and display existing route information, add_route to insert new routes, insert_route to validate and save the entered route into the database, delete_route to remove selected routes, and search_route to find and display routes based on a keyword within the source or destination fields. The main execution block initializes the application using Tkinter, creating an instance of the main class and initiating the GUI event loop. Additionally, the code ensures the proper closure of the database connection when the Tkinter loop ends, offering a functional transportation management system for manipulating route data.

Output of Code



```
        cost_label.pack()
        cost_entry = tk.Entry(add_window)
        cost_entry.pack()

        add_button = tk.Button(add_window, text="Add", command=lambda: self.insert_route(
            source_entry.get(), dest_entry.get(), distance_entry.get(), cost_entry.get()), bg="green")
        add_button.pack()

    def insert_route(self, source, dest, distance, cost):
        try:
            distance = float(distance)
```

Source: car
Destination: trivandrum
Distance: 24
Cost: 2400

Transportation Route Management

Transportation Routes

Source	Destination	Distance	Cost
bus	kerala	32.0	30000.0
bike	kerala	32.0	20000.0
car	banaglore	10.0	2000.0
jeep	karnataka	20.0	4000.0
train	kannur	200.0	3800.0
train	kerala	100.0	2400.0
train	kerala	100.0	2400.0
car	trivandrum	24.0	2400.0
car	trivandrum	24.0	2400.0

Search Source/Destination: kerala

Conclusion

The provided code intricately intertwines Tkinter's intuitive interface capabilities with the database management finesse of SQLite, crafting an impressive solution for transportation route management. Through the skillful orchestration within the Transportation Management class, the program delivers a seamless, user-centric interaction with transportation routes. This encapsulation offers a range of functionalities, from the simplicity of adding and deleting routes to the dynamic capability of searching, empowering users with versatile and effortless control over their data. The Tkinter-designed interface ensures a visually engaging and user-friendly experience, allowing users to navigate with ease and interact intuitively with the system.

The marriage of Tkinter's design elegance and SQLite's data robustness guarantees not just a system, but a reliable and scalable tool for handling vast volumes of transportation data. These fusion promises not only operational efficiency but also opens avenues for sophisticated decision-making in the domain of transportation management. As this solution continues to evolve and adapt, it holds the potential to become a pivotal instrument in the real-world transportation industry. It will not only streamline the management of route information but also foster advancements in operational efficiency and strategic decision-making, contributing significantly to the transportation management landscape.