Group Project Final Report

Project Title: **Ride Booking System** **Course:** **Bachelor of Science in Computer Engineering**
Object Oriented Programming

Year / Section / Group Name / Number:

1 / 4 / Group 6

Team Members:

Name	Student ID	Role
Tim Kaiser L. Llegue	2024-04875-MN-0	Project Manager
Hanniel II D. Alindogan	2024-02554-MN-0	Lead Developer
Chester John F. Mendoza	2024-00796-MN-0	UI/UX Designer
Xancho Bryan G. Monreal	2024-01561-MN-O	QA/Test Engineer
Yuan Joseph B. Lantion	2024-07146-MN-0	Assistant Developer
Luiz Alfonso L. Lee	2024-04202-MN-0	QA/Test Engineer
Vince Riel Del Rosario	2024-05253-MN-0	Assistant Developer

Instructor/Adviser: **Godofredo T. Avena**

Date Submitted:

July 03, 2025

Table of Contents

Introduction

The Ride Booking System, named **PUPSeat**, is a Python-based desktop application designed to provide a modern, interactive, and playful experience for users booking transport services between locations. Unlike conventional booking platforms, PUPSeat introduces a unique twist by offering not only familiar vehicle options like sedans, SUVs, and vans, but also an imaginative catalog of fantasy vehicles such as the Elevator of Willy Wonka, a magic carpet, a Tardis, dragons, and UFOs. This novel combination of practical booking features with whimsical transport choices aims to create an enjoyable and engaging application suited for both typical commuters and users looking for themed or entertainment-focused ride experiences. The system emphasizes clear visualization of each booking's route on a dynamic map interface while managing the trip lifecycle through statuses like Active, Finished, and Cancelled, all within a user-friendly graphical environment built on modern Python GUI libraries.

Objectives

The primary objective of PUPSeat is to develop a fully functional ride booking system capable of supporting a diverse array of vehicle types, calculating travel costs based on distance, and maintaining an accessible record of user trips.

Specifically, it aims to:

1. allow users to select from a combination of ordinary and fantasy vehicles
2. to automatically retrieve and compute the geolocation and distance between any two Philippine addresses entered by the user
3. to display a clean, interactive map that marks the pickup and drop-off points with a connecting path, regardless of whether the chosen vehicle flies or drives
4. to empower users to book rides, confirm fares before finalizing, and manage the status of each booking interactively.

5. to store booking data persistently using CSV files so that records remain intact between sessions.

Scope and Limitations

The scope of this project includes all core features necessary to demonstrate a ride booking workflow from selection to confirmation, visualization, and status management. The application covers vehicle selection, input validation, fare calculation using configurable base rates and per-kilometer costs, interactive mapping, and persistent storage of booking information. However, there are notable limitations inherent in this implementation. While fantasy vehicles appear in the dropdown list and are selectable by users, they share the same distance-based fare calculation logic as real vehicles, without any unique pricing rules or map visualization differences. The mapping functionality is limited to rendering a straight line between coordinates without dynamic animation or flight path simulation. Furthermore, the geolocation service relies on the Nominatim API from the geopy library, which means the accuracy and speed of location lookup depend on the availability and responsiveness of the external service. The system also restricts searches to locations within the Philippines by appending this suffix to all queries. Finally, the application is designed for local desktop environments using Python and cannot be accessed as a web or mobile app without additional adaptation.

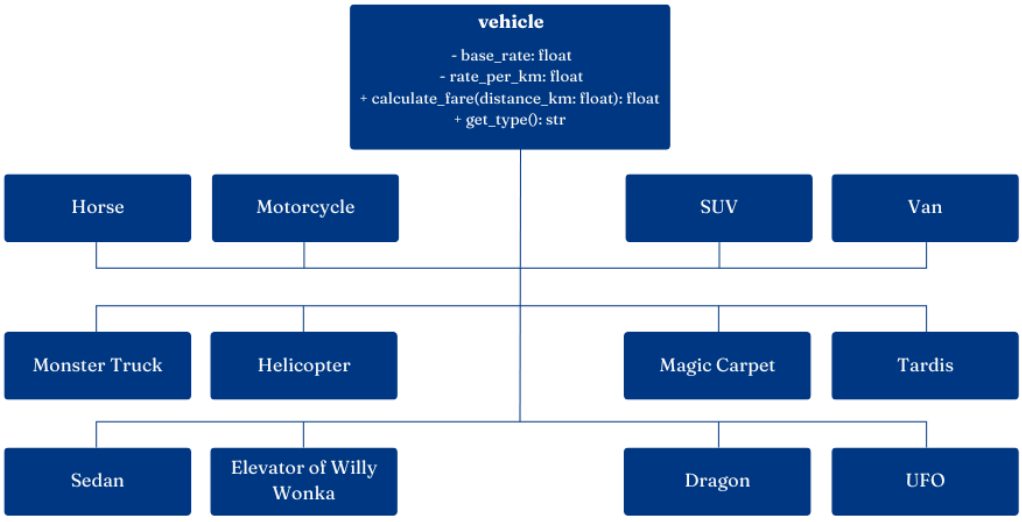
Methodology

The group followed a modular development approach with tasks divided based on roles. Our workflow encouraged iterative development and constant feedback. Each member focused on specific components as outlined below:

- Tim Llegue: Managed the project plan and ensured deliverables were on schedule. Also handled the GUI alongside Yuan.
- Hanniel Alindogan II: Developed the backend logic, including vehicle classes, fare calculations, and file operations.
- Chester John Mendoza: Designed the frontend GUI using CustomTkinter, ensuring a clean and intuitive layout.
- Yuan Joseph Lantion: Designed the GUI using CustomTkinter, ensuring a clean and intuitive layout
- Vince Del Rosario: Created test cases, performed UI and functional testing, and documented results.
- Xancho Monreal: Created test cases, performed UI and functional testing, and documented results.
- Luiz Lee: Created test cases, performed UI and functional testing, and documented results.

System Design

The system was built around an object-oriented design. At its core, the 'Vehicle' superclass defines common behavior such as fare computation, and its subclasses (e.g., Horse, Sedan, Dragon) override this to account for unique cost calculations. Encapsulation ensures that vehicle attributes remain hidden and accessed only through appropriate methods. A class diagram would illustrate the relationships between Vehicle and its subclasses, as well as the RideBookingApp class managing GUI and user interactions.



Technologies Used

PUPSeat leverages several Python libraries and tools. The graphical interface is constructed with **customtkinter** (CTk), which provides modern theming and widget customization beyond the capabilities of classic Tkinter. For mapping, the application uses **TkinterMapView**, enabling interactive, tile-based maps embedded within the GUI. Address resolution and distance measurement are handled by the **geopy** library, which connects to Nominatim's OpenStreetMap API and computes geodesic distances in kilometers. Persistent data storage is achieved through the built-in **csv** module, allowing easy inspection and portability of booking records. Additional dependencies include **os** for file path resolution, **ttk** for the styled table component displaying all bookings, and **messagebox** for pop-up dialogs and confirmations. These technologies were chosen for their balance of simplicity, community support, and suitability for desktop-based Python applications.

Implementation

app.py - serves as the primary entry point and user interface controller, the core module is the **RideBookingApp** class. This class integrates all the graphical components, event handlers, and coordination logic needed to operate the ride booking system. Inside **RideBookingApp**, the **__init__** method sets up the main application window, configuring the title, custom icon, and color themes to establish a cohesive look and feel. It also loads existing bookings by calling the **load_bookings()** function from the **booking** module, ensuring that previous trips are displayed immediately upon startup.

app.py

```
class RideBookingApp:
    def __init__(self, root):

#Initialize the main application window, set up the UI, and load bookings.

        self.root = root
        self.root.title("PUPSeat")
        ico_path = os.path.join(os.path.dirname(__file__), "app_logo.ico")
        try:
            self.root.iconbitmap(ico_path)
        except Exception:
            pass
        self.root.state('zoomed')
        self.root.configure(bg="#400000")

        self.bookings = load_bookings()
        self.active_paths = []
        self.create_widgets()
```

The **create_widgets** method in this module constructs the entire graphical interface, partitioning the window into a left panel for user input and booking tables, and a right panel for the interactive map. This method creates labeled entry fields for username, start location, and end location, as well as a vehicle selection dropdown menu that includes both conventional vehicles and whimsical options such as “Magic Carpet,” “Elevator of Willy Wonka,” and “UFO.”

```
def create_widgets(self):
    # Create and layout all widgets for the main application interface.

    self.root.grid_rowconfigure(0, weight=1)
    self.root.grid_columnconfigure(0, weight=3)
    self.root.grid_columnconfigure(1, weight=2)

    outer_frame = ctk.CTkFrame(self.root, fg_color="#400000")
    outer_frame.grid(row=0, column=0, columnspan=2, sticky="nsew")
    outer_frame.grid_rowconfigure(0, weight=1)
    outer_frame.grid_columnconfigure(0, weight=3)
    outer_frame.grid_columnconfigure(1, weight=2)

    left_frame = ctk.CTkFrame(outer_frame, fg_color="#400000")
    left_frame.grid(row=0, column=0, sticky="nsew", padx=10, pady=10)
    left_frame.grid_rowconfigure(2, weight=1)

    title_frame = ctk.CTkFrame(
        left_frame,
        fg_color="#400000",
        corner_radius=15,
        border_color="#f4bb00",
        border_width=2
    )
    title_frame.pack(pady=(5, 0), padx=130, fill="x")
    ctk.CTkLabel(
        title_frame,
        text="🚌 PUPSeat: Ride Booking System",
        font=("Lucida Console", 22, "bold"),
        text_color="#f4bb00",
    ).pack(pady=8)

    form_panel = ctk.CTkFrame(left_frame, fg_color="#400000", corner_radius=0)
    form_panel.pack(fill="x", padx=5, pady=(0,15))
    inner = ctk.CTkFrame(form_panel, fg_color="#d3d3d3")
    inner.pack(fill="x", padx=5, pady=5)
    inner.grid_columnconfigure((0,1), weight=1)

    entry_font = ctk.CTkFont(family="Courier New", size=16)
    label_font = ctk.CTkFont(family="Courier New", size=16, weight="bold")

    li = ctk.CTkFrame(inner, fg_color="#d3d3d3")
    li.grid(row=0, column=0, sticky="nsew", padx=5, pady=5)
    ctk.CTkLabel(li, text="Username:", font=label_font, text_color="#400000", fg_color="#d3d3d3").pack(anchor="w", padx=5,
pady=(5,0))
    self.username_entry = ctk.CTkEntry(li, font=entry_font, fg_color="white", text_color="black")
    self.username_entry.pack(fill="x", padx=5, pady=5)

    ctk.CTkLabel(li, text="Vehicle Type:", font=label_font, text_color="#400000", fg_color="#d3d3d3").pack(anchor="w",
padx=5, pady=(5,0))
    self.vehicle_var = ctk.StringVar(value="Sedan")
    self.vehicle_menu = ctk.CTkOptionMenu(
        li,
        variable=self.vehicle_var,
        values=[
            "Horse","Motorcycle","Sedan","SUV","Van","Monster Truck",
            "Helicopter","Elevator of Willy Wonka","Magic Carpet","Tardis","Dragon","UFO"
        ],
        fg_color="white", text_color="black",
        button_color="#f4bb00", button_hover_color="#c46b02",
        font=entry_font
    )
    self.vehicle_menu.pack(fill="x", padx=5, pady=5)

    ri = ctk.CTkFrame(inner, fg_color="#d3d3d3")
    ri.grid(row=0, column=1, sticky="nsew", padx=5, pady=5)
    ctk.CTkLabel(ri, text="Start Location:", font=label_font, text_color="#400000", fg_color="#d3d3d3").pack(anchor="w",
padx=5, pady=(5,0))
    self.start_entry = ctk.CTkEntry(ri, font=entry_font, fg_color="white", text_color="black")
    self.start_entry.pack(fill="x", padx=5, pady=5)

    ctk.CTkLabel(ri, text="End Location:", font=label_font, text_color="#400000", fg_color="#d3d3d3").pack(anchor="w",
padx=5, pady=(5,0))
    self.end_entry = ctk.CTkEntry(ri, font=entry_font, fg_color="white", text_color="black")
    self.end_entry.pack(fill="x", padx=5, pady=5)
```

The module further defines the **book_ride** method, which orchestrates the booking workflow—performing input validation, calling the **get_coords_and_distance** function from the **distance** module to retrieve coordinates and calculate the distance between two locations, generating an estimated fare via a vehicle object created by the **create_vehicle** factory function, and finally confirming the booking through a pop-up dialog. This method also appends the new booking to the in-memory list and updates the persistent storage.

app.py

```
def book_ride(self):

#Handle the booking process: validate input, calculate fare, confirm, and save booking.

        username = self.username_entry.get().strip()
        vehicle_type = self.vehicle_var.get()
        start = self.start_entry.get().strip()
        end = self.end_entry.get().strip()

        if not username or not start or not end:
            showerror("Input Error", "Please fill in all fields.")
            return

        if start.lower() == end.lower():
            showerror("Input Error", "Start and end locations cannot be the same.")
            return

        start_coords, end_coords, distance = get_coords_and_distance(start, end)
        if not start_coords or not end_coords:
            showerror("Location Error", "Invalid start or end location.")
            return

        vehicle = create_vehicle(vehicle_type)
        cost = vehicle.calculate_fare(distance)
```

In addition, **app.py** provides the **finish_booking, cancel_booking**, and **clear_finished_cancelled** methods to let users update the status of bookings and manage the display of records. The **update_status** method handles logic for changing the state of a selected booking, confirming the user's choice, and visually tagging each row in the table with color-coded status indicators such as "Active," "Finished," or "Cancelled."

app.py

```
def finish_booking(self):

#Mark the selected booking(s) as 'Finished'.

        self.update_status("Finished")
        self.tree.selection_remove(self.tree.selection())

    def cancel_booking(self):

#Mark the selected booking(s) as 'Cancelled'.

        self.update_status("Cancelled")
        self.tree.selection_remove(self.tree.selection())

    def update_status(self, new_status):

#Update the status of selected booking(s) in the tree and backend data.

        selected = self.tree.selection()
        if not selected:
            showwarning("No Selection", f"Select a booking to mark as {new_status}.")
            return

        updated = False
        for item in selected:
            values = self.tree.item(item, "values")
            booking_id, current_status = values[0], values[-1]

            if current_status in ["Finished", "Cancelled"]:
                showinfo("Action Blocked", f"Booking {booking_id} is already {current_status}.")
                continue

            confirm = askyesno(f"Confirm {new_status}", f"Are you sure you want to mark booking {booking_id} as
{new_status}?")
            if not confirm:
                continue

            for booking in self.bookings:
                if booking["id"] == booking_id:
                    booking["status"] = new_status
                    break

            self.tree.item(item, values=(*values[:-1], new_status), tags=(new_status,))
            updated = True

        if updated:
            save_bookings(self.bookings)
            self.update_map_view()
```

Finally, the **update_map_view** method clears existing map markers and draws new markers and straight-line paths for all currently active bookings, ensuring the visual representation stays synchronized with the records.

app.py

```
def update_map_view(self):

#Update the map to show active bookings with markers and paths.

        for item in self.active_paths:
            item.delete()
        self.active_paths.clear()

        self.map_widget.set_position(14.5995, 120.9842)
        self.map_widget.set_zoom(10)

        for booking in self.bookings:
            if booking["status"] ≠ "Active":
                continue
            coords = get_coords_and_distance(booking["start"], booking["end"])
            if coords[0] and coords[1]:
                m1 = self.map_widget.set_marker(*coords[0], text="Start", text_color="black")
                m2 = self.map_widget.set_marker(*coords[1], text="End", text_color="black")
                path = self.map_widget.set_path([coords[0], coords[1]])
                self.active_paths.extend([m1, m2, path])
```

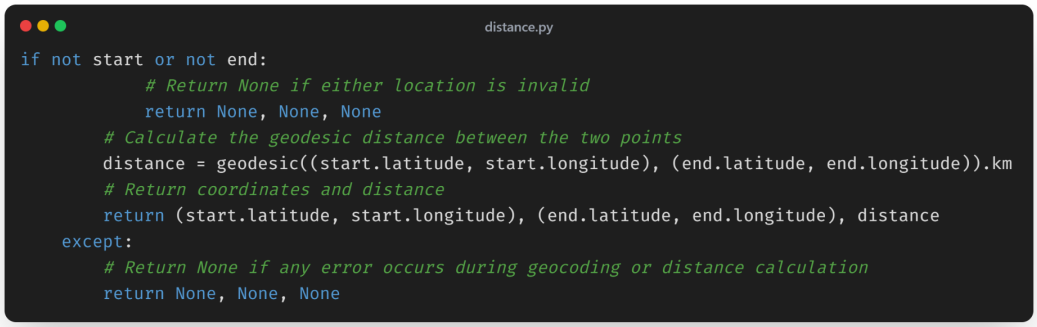
distance.py - this module contains the **get_coords_and_distance** function, which is a utility that encapsulates all logic related to geolocation and distance computation. This function relies on the **Nominatim** geocoder from the **geopy** library to translate textual addresses into latitude and longitude coordinates, explicitly appending ", Philippines" to improve accuracy and limit lookups to the intended region. It then calculates the straight-line (geodesic) distance between the start and end coordinates using the **geodesic** method from **geopy.distance.**

distance.py

```
from geopy.geocoders import Nominatim
from geopy.distance import geodesic

def get_coords_and_distance(start_location, end_location):
    # Get the coordinates (latitude, longitude) for two locations and calculate the distance between them in kilometers
    geolocator = Nominatim(user_agent="ride-booking-map")
    try:
        # Geocode the start and end locations (assume locations are in the Philippines)
        start = geolocator.geocode(start_location + ", Philippines")
        end = geolocator.geocode(end_location + ", Philippines")
```

If the address lookup fails—due to an invalid or ambiguous location—the function returns **None** values to indicate an error, which the calling code in **app.py** can detect and respond to by showing appropriate error dialogs. This module effectively abstracts away all geographic computation, providing a clean interface for the main application logic to use without worrying about low-level geocoding details.



vehicle.py - this module defines the **Vehicle** class along with several subclasses representing different vehicle types. The base **Vehicle** class encapsulates attributes such as **base_rate** and **rate_per_km** and provides the **calculate_fare** method, which computes the cost of a trip by adding the base fare to the per-kilometer charge multiplied by the calculated distance. Subclasses like **Sedan, SUV**, and **Van** specify distinct pricing configurations by initializing different base rates and per-kilometer rates through their constructors.

vehicle.py

```python
class Vehicle:
    # Base class for all vehicles, handles fare calculation and type retrieval
    def __init__(self, base_rate, rate_per_km):
        # Initialize vehicle with base rate and rate per kilometer
        self._base_rate = base_rate
        self._rate_per_km = rate_per_km

    def calculate_fare(self, distance_km):
        # Calculate fare based on distance
        return self._base_rate + (self._rate_per_km * distance_km)

    def get_type(self):
        # Return the vehicle type as a string
        return self.__class__.__name__.replace("_", " ")


class Horse(Vehicle):
    # Horse vehicle type
    def __init__(self):
        super().__init__(base_rate=30, rate_per_km=5)


class Motorcycle(Vehicle):
    # Motorcycle vehicle type
    def __init__(self):
        super().__init__(base_rate=40, rate_per_km=10)


class Sedan(Vehicle):
    # Sedan vehicle type
    def __init__(self):
        super().__init__(base_rate=50, rate_per_km=15)


class SUV(Vehicle):
    # SUV vehicle type
    def __init__(self):
        super().__init__(base_rate=60, rate_per_km=20)


class Van(Vehicle):
    # Van vehicle type
    def __init__(self):
        super().__init__(base_rate=70, rate_per_km=30)
```

vehicle.py

```python
class Monster_Truck(Vehicle):
    # Monster Truck vehicle type
    def __init__(self):
        super().__init__(base_rate=150, rate_per_km=60)


class Helicopter(Vehicle):
    # Helicopter vehicle type
    def __init__(self):
        super().__init__(base_rate=1000, rate_per_km=200)


class Elevator_of_Willy_Wonka(Vehicle):
    # Elevator of Willy Wonka vehicle type
    def __init__(self):
        super().__init__(base_rate=3000, rate_per_km=500)


class Magic_Carpet(Vehicle):
    # Magic Carpet vehicle type
    def __init__(self):
        super().__init__(base_rate=5000, rate_per_km=800)


class Tardis(Vehicle):
    # Tardis vehicle type
    def __init__(self):
        super().__init__(base_rate=8000, rate_per_km=1100)


class Dragon(Vehicle):
    # Dragon vehicle type
    def __init__(self):
        super().__init__(base_rate=50000, rate_per_km=10000)


class UFO(Vehicle):
    # UFO vehicle type
    def __init__(self):
        super().__init__(base_rate=100000, rate_per_km=25000)


def create_vehicle(vehicle_type):
    # Factory function to create a vehicle instance based on the type string
    vehicles = {
        "Horse": Horse,
        "Motorcycle": Motorcycle,
        "Sedan": Sedan,
        "SUV": SUV,
        "Van": Van,
        "Monster Truck": Monster_Truck,
        "Helicopter": Helicopter,
        "Elevator of Willy Wonka": Elevator_of_Willy_Wonka,
        "Magic Carpet": Magic_Carpet,
        "Tardis": Tardis,
        "Dragon": Dragon,
        "UFO": UFO,
    }
    return vehicles.get(vehicle_type, Sedan)()
```

booking.py - this module is dedicated for persistent storage of book records. The **load_bookings** function reads from a CSV file, parsing each row into a dictionary that contains keys such as booking ID, username, vehicle type, start and end addresses, distance, fare cost, and booking status. This function populates the bookings list used by the application upon startup, ensuring continuity across sessions. Conversely, the **save_bookings** function writes the entire current list of booking records back to the CSV file, updating it whenever the user books, finishes, or cancels a trip. These two functions encapsulate all logic for reading and writing persistent data, making the system robust and easy to maintain without requiring a more complex database backend.

booking.py

```python
import csv
import os

BOOKING_FILE = "bookings.csv"

def load_bookings():
    # Load bookings from the CSV file and return a list of booking dictionaries
    bookings = []
    if os.path.exists(BOOKING_FILE):
        with open(BOOKING_FILE, mode="r", encoding="utf-8") as f:
            reader = csv.DictReader(f)
            for row in reader:
                bookings.append({
                    "id": row["ID"],
                    "user": row["User"],
                    "vehicle": row["Vehicle"],
                    "start": row["Pick Up"],
                    "end": row["Drop Off"],
                    "distance": row["Distance"],
                    "cost": row["Cost"],
                    "status": row["Status"]
                })
    return bookings

def save_bookings(bookings):
    # Save the list of booking dictionaries to the CSV file
    with open(BOOKING_FILE, mode="w", newline="", encoding="utf-8") as f:
        writer = csv.writer(f)
        writer.writerow([
            "ID", "User", "Vehicle", "Pick Up", "Drop Off", "Distance", "Cost", "Status"
        ])  # Write header row
        for b in bookings:
            writer.writerow([
                b["id"], b["user"], b["vehicle"],
                b["start"], b["end"],
                b["distance"], b["cost"], b["status"]
            ])
```

Together, these modules—**app.py** for user interaction, **distance.py** for geolocation, **vehicle.py** for fare calculation, and **booking.py** for persistent storage- form a clear, layered architecture that separates concerns effectively. Each module is responsible for a specific domain of the system: the interface and user experience, the business logic of distance and pricing, and the management of historical booking records. This modular approach makes the codebase easier to understand, test, and extend in the future.

Testing and Evaluation

We applied the following testing strategies:

- Unit Testing – Validated fare calculations per vehicle.
- Integration Testing – Ensured file operations and map integration worked together.
- UI Testing – Checked button responsiveness and field validation.
- Error Handling – Handled invalid input, empty fields, and incorrect locations gracefully.
- Manual UI Testing (entry validation, button function).
- Functional Testing (CSV saving).
- Geolocation Testing (invalid vs. valid location names).

Results:

- All basic functions worked as expected.
- Invalid entries properly triggered error prompts.

Results and Discussion

The project met most of its goals: it successfully handled user input, fare computation, and map display. One of the challenges was handling errors from the geolocation API, especially with misspelled city names. Another was fine-tuning the UI responsiveness using the CustomTkinter framework.

The system successfully met the required features. Notable challenges and solutions included:

- Integrating geolocation – Required appending ', Philippines' to avoid ambiguous addresses.
- Dynamic GUI management – Resolved layout issues across screen sizes.
- Handling fictional vehicles – Ensured extreme fare values were processed without crashing.

Conclusion

Through this project, we gained hands-on experience in applying object-oriented programming principles such as encapsulation, inheritance, and modular design. We learned how to structure a program effectively by separating frontend interaction, backend logic, and data storage using CSV files. These technical skills were further reinforced through real-world problem solving, as we created a functional vehicle booking system. Beyond coding, the project emphasized the importance of teamwork and planning. By dividing responsibilities and communicating regularly, we were able to complete tasks more efficiently and resolve issues collaboratively. This experience not only strengthened our coding abilities but also improved our time management, coordination, and adaptability skills that are essential for future software development projects.

Future Work

For future development of this vehicle booking system, several enhancements can be implemented to improve its functionality, user experience, and scalability. One key improvement would be the integration of a graphical user interface (GUI) using tools like Tkinter or PyQt to replace or complement the command-line interface, making the system more accessible and user-friendly. Additionally, incorporating a real-time database such as SQLite or Firebase instead of CSV files would allow for better data management, concurrency, and scalability. Security features like user authentication and input validation can also be added to ensure safe and authorized use of the system. From a functional perspective, adding automated price calculations based on distance, vehicle type, and time of booking would increase realism and utility. Advanced features such as booking history tracking, cancellation policies, and email confirmations could provide a more comprehensive experience. Lastly, making the system web-based or mobile-accessible would significantly expand its reach and usability. These improvements would elevate the project from a basic educational tool to a robust, real-world application.

References

GeeksforGeeks. (2016). *Working with csv files in Python*. GeeksforGeeks. https://www.geeksforgeeks.org/python/working-csv-files-python/

LTO (2022). *What is TNVS? Transportation Network Vehicle Service*. LTO Portal PH. https://ltoportal.ph/tnvs-transportation-network-vehicle-service/#Fare_Rates_For_Transportation_Network_Vehicle_Services_TNVS

MoveIt (n.d.). *Fare Calculator*. https://moveit.com.ph/fare-calculator/

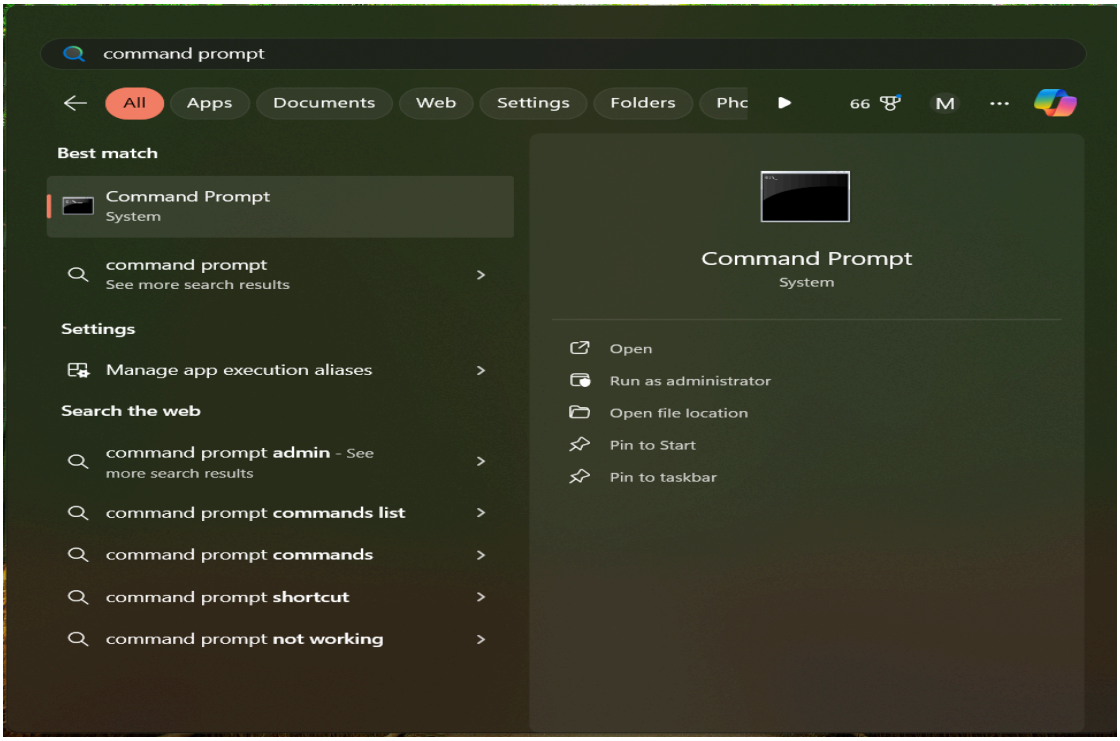
SivaraamTK (2022). *GeoPy. Guide.* futurepreneur. https://medium.com/featurepreneur/geopy-dfc65e40f4c9

The Paper Mill Blog (2016). *Mastering Color Theory: Understanding Active and Passive Colors* https://blog.thepapermillstore.com/color-theory-active-versus-passive-colors/?fbclid=IwY2xjawLQpjFleHRuA2FlbQIxMABicmlkETFBY0gwdnZsdWJsRENoTHV4AR40IDutyKGf4nOzjUDvFX5XlnTRcfFb9AOJOm2sEyEJdKj2EEmZKuSDqmY7mw_aem_jHJFEmHlrcIh4aYnkCOaew

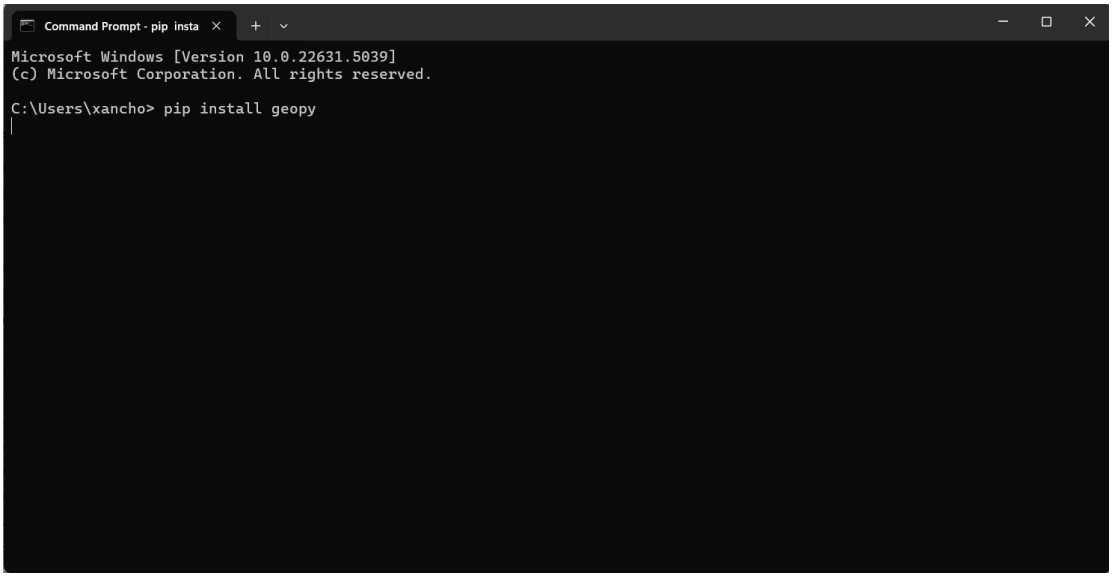
Appendices

Installation Guides

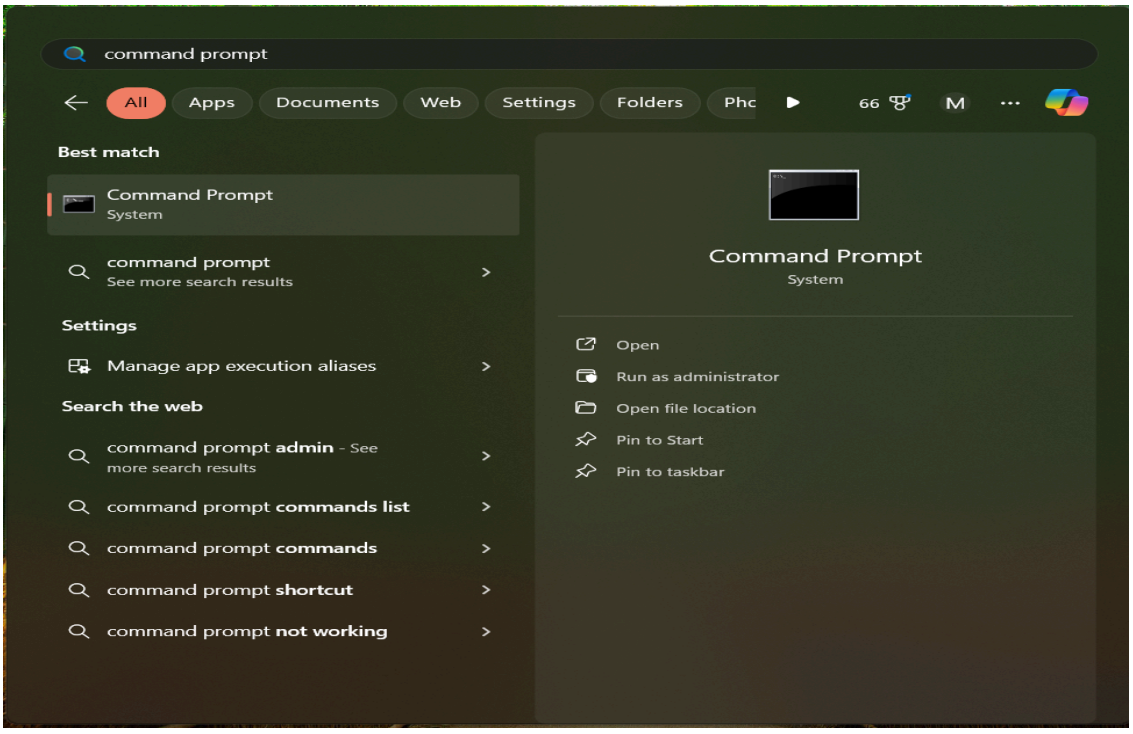
To install geopy, first you must go to your windows command prompt



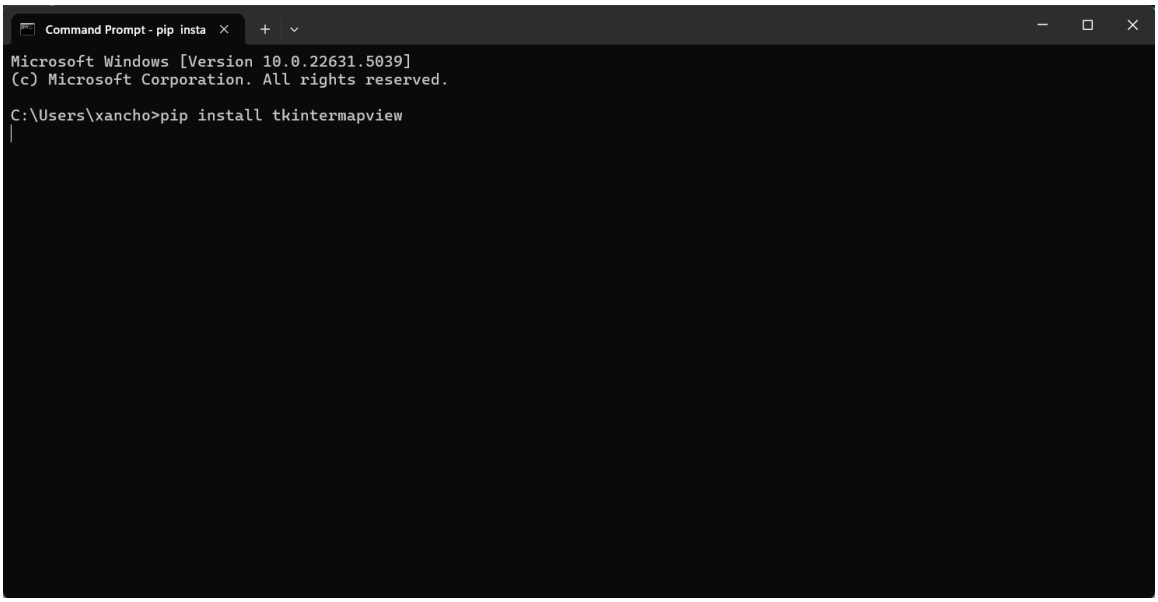
Then, type “pip install geopy” in the command prompt. And you’re done; you can now use the module geopy in your programs.



To install Tkintermapview, first you must go to your windows command prompt



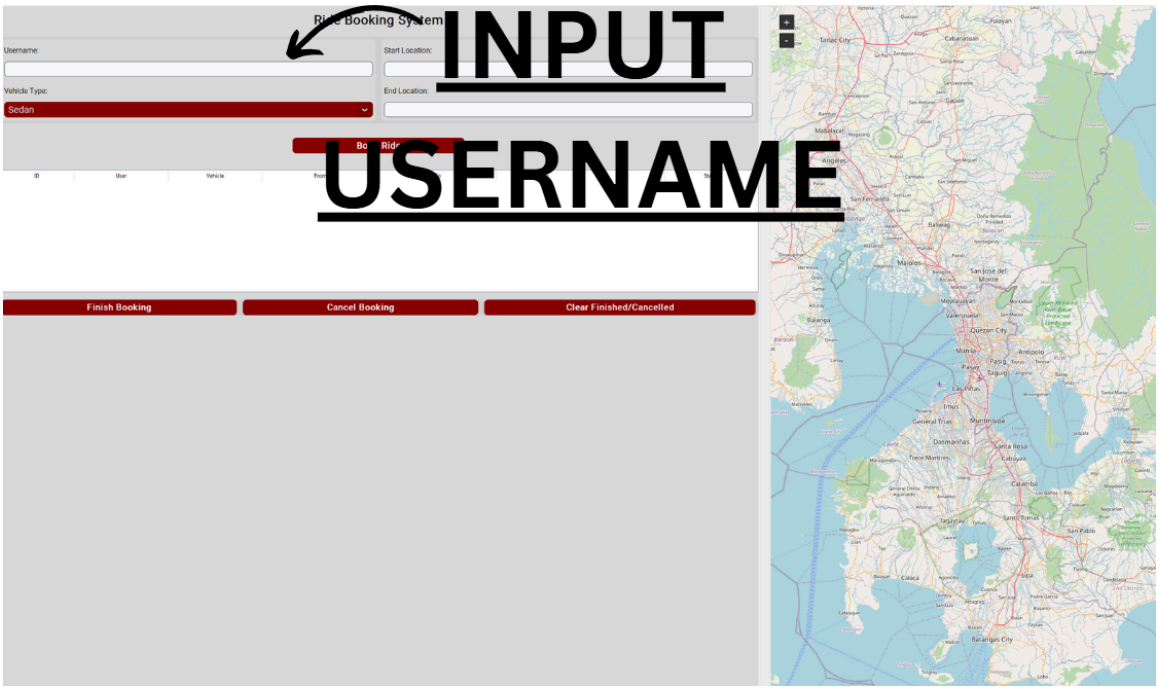
Then, type “pip install Tkintermapview” in the command prompt. And you’re done; you can now use the module Tkintermapview in your programs.



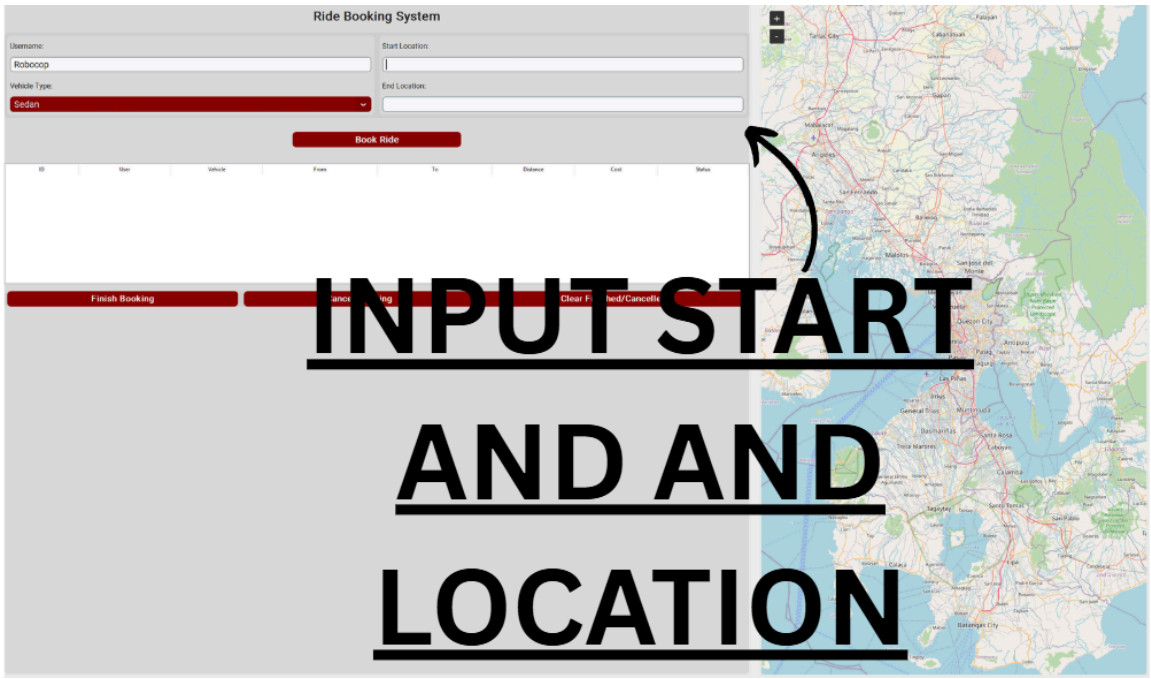
User Manual

Booking a ride:

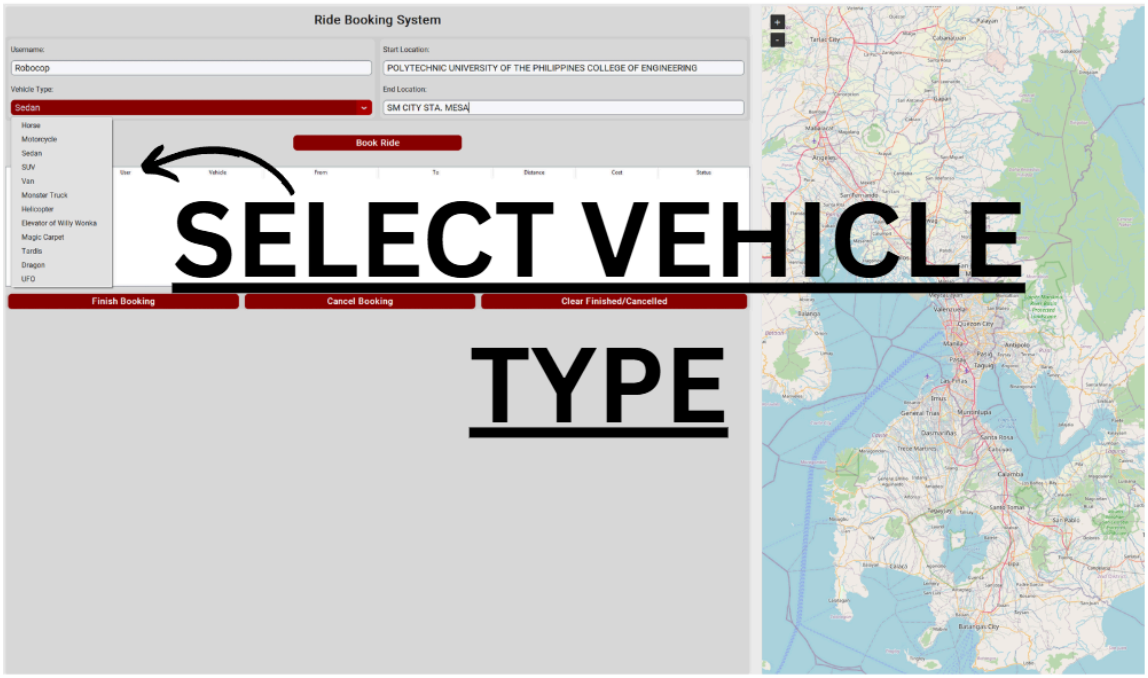
1. Input the username for tracking the identity and logging the records of the user who will book a ride.



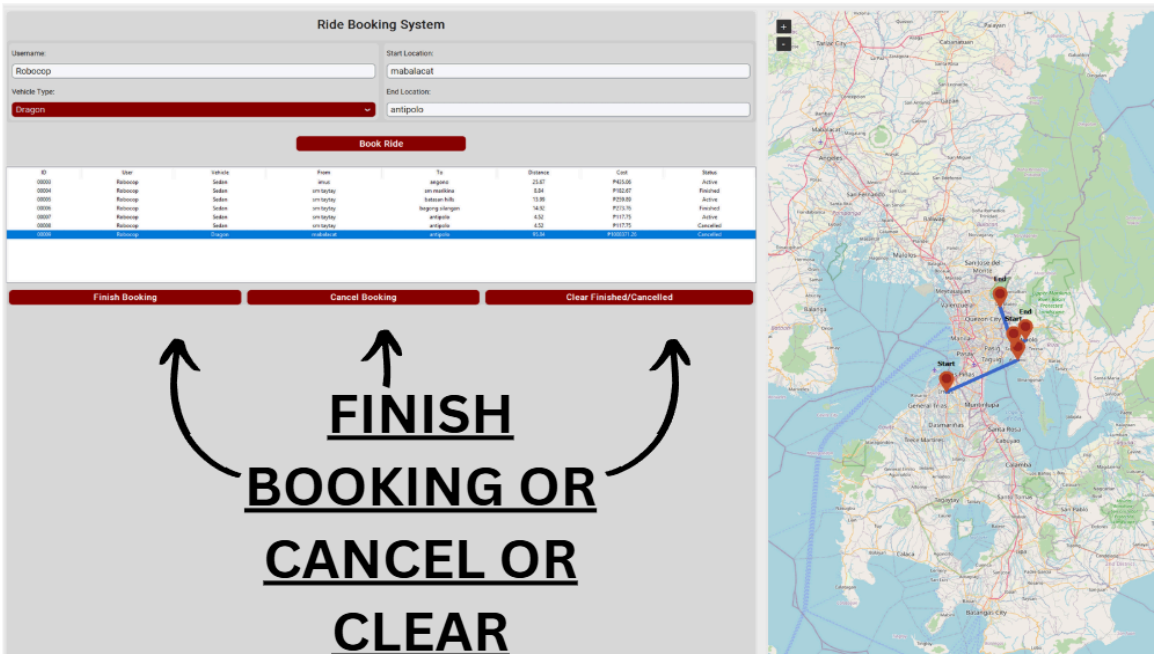
2. Enter the start location and end location names.



3. Choose your vehicle type, each vehicle has its own rate, and the fare would be larger depending on the distance with regard to the vehicle type.



4. After choosing the vehicle type, book the ride, and you may see your booking history added with the new booking you just made, and it will be tagged as active. The user may choose to cancel or mark their booking as finished. The system only allows for clearing finished and canceled bookings.



Fare Rates Table

The fares of the rides are calculated by multiplying the vehicle rate rate per kilometer by the calculated distance from start to end of the trip. Additionally, there is a base flagdown rate for each vehicle which varies from vehicle to vehicle.

TYPE OF VEHICLE	VEHICLE FLAGDOWN RATE	RATE PER KILOMETER
Horse	₱30	₱5/km
Motorcylce	₱40	₱10/km
Sedan	₱50	₱15/km
SUV	₱60	₱20/km
Van	₱70	₱30/km
Monster Truck	₱150	₱60/km
Helicopter	₱1,000	₱200/km
Elevator of Willy Wonka	₱3,000	₱500/km
Magic Carpet	₱5,000	₱800/km
Tardis	₱8,000	₱1,100/km
Dragon	₱50,000	₱10,000/km
UFO	₱100,000	₱25,000/km