Report: Optimizing NYC Taxi Operations

1. Data Preparation

Import Libraries

Import Libraries

```
# Import warning
```

```
import warnings
warnings.filterwarnings("ignore")
```

```
# Import the libraries you will be using for analysis
```

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
sns.set_theme()
```

```
# Recommended versions
# numpy version: 1.26.4
# pandas version: 2.2.2
# matplotlib version: 3.10.0
# seaborn version: 0.13.2

# Check versions
print("numpy version:", np.__version__)
print("pandas version:", pd.__version__)
print("matplotlib version:", plt.matplotlib.__version__)
print("seaborn version:", sns.__version__)
```

```
numpy version: 2.3.5
pandas version: 2.3.3
matplotlib version: 3.10.6
seaborn version: 0.13.2
```

This code is just setting things up before we start working with data. It tells Python to hide warning messages so the notebook looks clean. Then it brings in the main tools we need: NumPy for maths, Pandas for working with tables of data, and Matplotlib plus Seaborn for making graphs and charts. It also makes sure that any graphs we create will show up inside the notebook and look nice by default. At the end, it prints the versions of these tools so we know which ones are being used.

1.1. Loading the dataset

```
df = pd.read_parquet("C:\\Users\\hp\\Downloads\\Datasets and Dictionary-NYC\Datasets and Dictionary\\trip_records\\2023-1.parquet")
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 3041714 entries, 0 to 3066765
Data columns (total 19 columns):
 #   Column                 Dtype
---  ------                 -----
 0   VendorID               int64
 1   tpep_pickup_datetime   datetime64[us]
 2   tpep_dropoff_datetime  datetime64[us]
 3   passenger_count        float64
 4   trip_distance          float64
 5   RatecodeID             float64
 6   store_and_fwd_flag     object
 7   PULocationID           int64
 8   DOLocationID           int64
 9   payment_type           int64
 10  fare_amount            float64
 11  extra                  float64
 12  mta_tax                float64
 13  tip_amount             float64
 14  tolls_amount           float64
 15  improvement_surcharge  float64
 16  total_amount           float64
 17  congestion_surcharge   float64
 18  airport_fee            float64
dtypes: datetime64[us](2), float64(12), int64(4), object(1)
memory usage: 464.1+ MB
```

This code loads a taxi trip data file (in parquet format) into a table called df, and then df.info() is used to show a summary of the dataset. It tells us how many rows and columns there are, what each column is called, and what type of data each column contains (like numbers, dates, or text). It also shows how much memory the dataset is using. In short, this step is just checking what the data looks like and how it is structured before doing any analysis.

1.1.1. Sample the data and combine the files

```
import os
BASE_PATH =r"C:\\Users\hp\\Downloads\\Datasets and Dictionary-NYC\\Datasets and Dictionary\\trip_records"
os.chdir(BASE_PATH)
print(os.getcwd())
```

```
C:\Users\hp\Downloads\Datasets and Dictionary-NYC\Datasets and Dictionary\trip_records
```

```
parquet_files = [
    "2023-1.parquet",
    "2023-2.parquet",
    "2023-3.parquet",
    "2023-4.parquet",
    "2023-5.parquet",
    "2023-6.parquet",
    "2023-7.parquet",
    "2023-8.parquet",
    "2023-9.parquet",
    "2023-10.parquet",
    "2023-11.parquet",
    "2023-12.parquet"
]
```

```
import os

os.chdir("C:\\Users\\hp\\Downloads\\Datasets and Dictionary-NYC\\Datasets and Dictionary\\trip_records")
file_list = os.listdir()
df = pd.DataFrame()

for file_name in file_list:
    try:
        print(f"Processing file: {file_name}")
        file_path = os.path.join(os.getcwd(), file_name)
        month_df = pd.read_parquet(file_path)
        month_df['tpep_pickup_datetime'] = pd.to_datetime(month_df['tpep_pickup_datetime'])
        month_df['pickup_date'] = month_df['tpep_pickup_datetime'].dt.date
        month_df['pickup_hour'] = month_df['tpep_pickup_datetime'].dt.hour

        sampled_data = pd.DataFrame()
        for date in month_df['pickup_date'].unique():
            date_data = month_df[month_df['pickup_date'] == date]
            for hour in range(24):
                hour_data = date_data[date_data['pickup_hour'] == hour]
                if len(hour_data) > 0:

                    hour_sample = hour_data.sample(
                        frac=0.05,
                        random_state=42
                    )
                    sampled_data = pd.concat(
                        [sampled_data, hour_sample],
                        ignore_index=True
                    )
        df = pd.concat([df, sampled_data], ignore_index=True)
        del month_df, sampled_data
    except Exception as e:
        print(f"Error reading file {file_name}: {e}")
```

This code goes into a folder that has many taxi trip data files and opens them one by one. For each file, it reads the data and splits the pickup time into the date and the hour of the day. Then, for every single day and every single hour, it takes only 5% of the trips from that hour as a small sample. All these small samples are then joined together into one big table. This way, instead of using the full huge dataset, we keep a smaller version that still represents all hours and days fairly, making the data much easier and faster to work with.

2. Data Cleaning

2.1. Fixing Columns

2.1.1. Fix the index

```
df['tpep_pickup_datetime'] = pd.to_datetime(df['tpep_pickup_datetime'])
df['tpep_dropoff_datetime'] = pd.to_datetime(df['tpep_dropoff_datetime'])
df['pickup_date'] = pd.to_datetime(df['pickup_date'])
```

This code changes the pickup time, drop-off time, and pickup date columns into a proper date-and-time format using pd.to_datetime(). Even if the values already look like dates, they might still be treated as plain text by Python. So, this step makes sure Python understands them as real dates and times, which makes it much easier to sort, filter, and do time-based calculations later, like finding trips by day, month, or hour.

2.1.2. Combine the two airport_fee columns

```
df['airport_fee'] = df['airport_fee'].fillna(df['Airport_fee'])
df.drop(columns=['Airport_fee'], inplace=True)
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1896400 entries, 0 to 1896399
Data columns (total 21 columns):
 #   Column                 Dtype
---  ------                 -----
 0   VendorID               int64
 1   tpep_pickup_datetime   datetime64[ns]
 2   tpep_dropoff_datetime  datetime64[ns]
 3   passenger_count        float64
 4   trip_distance          float64
 5   RatecodeID             float64
 6   store_and_fwd_flag     object
 7   PULocationID           int64
 8   DOLocationID           int64
 9   payment_type           int64
 10  fare_amount            float64
 11  extra                  float64
 12  mta_tax                float64
 13  tip_amount             float64
 14  tolls_amount           float64
 15  improvement_surcharge  float64
 16  total_amount           float64
 17  congestion_surcharge   float64
 18  airport_fee            float64
 19  pickup_date            datetime64[ns]
 20  pickup_hour            int64
dtypes: datetime64[ns](3), float64(12), int64(5), object(1)
memory usage: 303.8+ MB
```

This code is cleaning up the airport fee information. First, it fills any missing values in the airport_fee column using the values from another column called Airport_fee, so there are no blanks left. Once the useful values are copied over, the extra Airport_fee column is deleted because it is no longer needed. Finally, df.info() is used to check the updated dataset structure and confirm that the columns and data types look correct.

2.1.3. Fix columns with negative (monetary) values

```
negative_fare = df[df['fare_amount'] < 0]
negative_fare.shape
```

```
(0, 21)
```

This code is checking whether there are any taxi trips with a negative fare, which would be incorrect because fares should not be less than zero. It filters the dataset to only keep rows where the fare_amount is below 0 and stores them in a new table called negative_fare. Then negative_fare.shape shows how many such rows exist. The result (0, 21) means there are zero trips with negative fares, so the dataset does not contain any wrong or impossible fare values.

Analyse RatecodeID for the negative fare amounts

```
negative_total = df[df['total_amount'] < 0]
negative_total['RatecodeID'].value_counts()
```

```
RatecodeID
1.0    49
2.0    24
5.0     3
3.0     1
4.0     1
Name: count, dtype: int64
```

This code is checking for trips where the **total amount paid is negative**, which is not realistic. First, it filters the data to keep only the rows where total_amount is less than 0 and stores them in negative_total. Then it looks at the RatecodeID column for those rows and counts how many negative total trips belong to each rate type. The result shows that most negative totals happened under rate code 1, followed by rate code 2, and a few under the others. This helps identify which rate categories are linked to incorrect negative totals.

Find which columns have negative values

```
monetary_cols = [
    'fare_amount', 'extra', 'mta_tax', 'tip_amount',
    'tolls_amount', 'improvement_surcharge',
    'congestion_surcharge', 'airport_fee', 'total_amount'
]
for col in monetary_cols:
    print(col, (df[col] < 0).sum())
```

```
fare_amount 0
extra 3
mta_tax 73
tip_amount 0
tolls_amount 0
improvement_surcharge 78
congestion_surcharge 56
airport_fee 15
total_amount 78
```

This code is checking all the money-related columns to see how many negative values they contain, because money amounts like fare, tips, and taxes should normally not be less than zero. First, a list of these columns is created. Then, for each column, the code counts how many rows have a value below zero and prints the result. The output shows that some columns, like fare_amount, have no negative values, while others, like mta_tax, improvement_surcharge, congestion_surcharge, airport_fee, and total_amount, do have a small number of negative entries. This helps identify where incorrect or suspicious money values exist in the dataset so they can be cleaned later.

Fix these negative values

```
for col in monetary_cols:
    df.loc[df[col] < 0, col] = np.nan
```

This code is cleaning the dataset by removing wrong money values. For every money-related column in the list monetary_cols, it looks for rows where the value is negative. Whenever it finds a negative value, it replaces it with NaN, which means “missing value.” This is done because negative amounts for fares, taxes, tips, or fees usually don’t make sense in real life, so instead of keeping wrong data, the code marks them as missing so they can be handled properly later.

2.2. Handling Missing Values

2.2.1. Find the proportion of missing values in each column

```
missing_proportion = df.isna().mean().sort_values(ascending=False)
missing_proportion
```

```
congestion_surcharge     0.034239
airport_fee              0.034217
passenger_count          0.034209
RatecodeID               0.034209
store_and_fwd_flag       0.034209
total_amount             0.000041
improvement_surcharge    0.000041
mta_tax                  0.000038
extra                    0.000002
VendorID                 0.000000
tip_amount               0.000000
pickup_date              0.000000
tolls_amount             0.000000
fare_amount              0.000000
tpep_pickup_datetime     0.000000
payment_type             0.000000
DOLocationID             0.000000
PULocationID             0.000000
trip_distance            0.000000
tpep_dropoff_datetime    0.000000
pickup_hour              0.000000
dtype: float64
```

This code is checking how much missing data each column has in the dataset. First, df.isna() marks missing values, then .mean() calculates what fraction (or percentage) of values are missing in each column, and finally .sort_values(False) sorts the columns from the highest missing percentage to the lowest. The result shows that a few columns like congestion_surcharge, airport_fee, and passenger_count have around 3.4% missing values, while most other columns have almost none or zero missing data. This helps understand where data cleaning or filling might be needed.

2.2.2. Handling missing values in passenger_count

```
:  df[df['passenger_count'].isna()].head()
   passenger_mode = df['passenger_count'].mode()[0]
   df['passenger_count'] = df['passenger_count'].fillna(passenger_mode)
```

This code is fixing the missing values in the passenger_count column. First, it quickly checks a few rows where passenger_count is empty. Then it calculates the **mode**, which is the most common passenger count value in the dataset. Finally, it fills all missing passenger counts with this most common value. This way, instead of leaving blanks, the dataset uses a logical replacement based on the most frequent real value.

2.2.3. Handle missing values in RatecodeID

```
ratecode_mode = df['RatecodeID'].mode()[0]
df['RatecodeID'] = df['RatecodeID'].fillna(ratecode_mode)
```

This code is filling in missing values in the RatecodeID column. First, it finds the **mode**, which is the value that appears most often in that column. Then it replaces all the missing RatecodeID entries with this most common value. This helps remove gaps in the data while keeping the replacement realistic, since it uses the value that occurs most frequently in real trips.

2.2.4. Impute NaN in congestion_surcharge

```
df['congestion_surcharge'] = df['congestion_surcharge'].fillna(0)
```

This code is fixing missing values in the congestion_surcharge column. It replaces all empty or missing values with **0**, meaning that if the dataset doesn't show a congestion surcharge for a trip, it assumes no extra fee was charged. This keeps the column complete with no blanks and makes the data easier to work with.

Handle any remaining missing values:

```
store_flag_mode = df['store_and_fwd_flag'].mode()[0]
df['store_and_fwd_flag'] = df['store_and_fwd_flag'].fillna(store_flag_mode)
df['airport_fee'] = df['airport_fee'].fillna(0)
df['mta_tax'] = df['mta_tax'].fillna(0)
df['improvement_surcharge'] = df['improvement_surcharge'].fillna(0)
df['total_amount'] = df['total_amount'].fillna(0)
df['extra'] = df['extra'].fillna(0)
df.isna().sum()
```

```
VendorID                0
tpep_pickup_datetime    0
tpep_dropoff_datetime   0
passenger_count         0
trip_distance           0
RatecodeID              0
store_and_fwd_flag      0
PULocationID            0
DOLocationID            0
payment_type            0
fare_amount             0
extra                   0
mta_tax                 0
tip_amount              0
tolls_amount            0
improvement_surcharge   0
total_amount            0
congestion_surcharge    0
airport_fee             0
pickup_date             0
pickup_hour             0
dtype: int64
```

This code is finishing the cleaning of missing values in the dataset. First, it finds the most common value (mode) for the store_and_fwd_flag column and fills the missing values in that column with this common value. Then, for several money-related columns like airport_fee, mta_tax, improvement_surcharge, total_amount, and extra, it replaces any missing values with 0, meaning no fee or charge was applied. Finally, df.isna().sum() is used to check how many missing values are left in each column, and the result shows **0 for every column**, meaning the dataset is now completely free of missing data.

2.3. Handling Outliers and Standardizing Values

Before we start fixing outliers, let's perform outlier analysis.

```
]: numeric_cols = [
    'passenger_count',
    'trip_distance',
    'fare_amount',
    'extra',
    'mta_tax',
    'tip_amount',
    'tolls_amount',
    'improvement_surcharge',
    'total_amount',
    'congestion_surcharge',
    'airport_fee'
]
df[numeric_cols].describe()
```

This code is selecting all the number-based columns, such as passenger count, trip distance, fares, taxes, tips, and fees, and storing their names in a list called numeric_cols. Then df[numeric_cols].describe() is used to generate basic statistics for these columns, like the minimum, maximum, average, and quartiles. This helps you quickly understand the typical values, ranges, and spread of the numeric data in the dataset.

2.3.1. Check outliers in payment type, trip distance and tip amount columns

```
# remove passenger_count > 6
```

```
df = df[df['passenger_count'] <= 6]
```

```
# Continue with outlier handling
```

```
df = df[df['payment_type'] != 0]
df = df[df['trip_distance'] <= 250]
df = df[~((df['trip_distance'] <= 0.1) & (df['fare_amount'] > 300))]
df = df[~(
    (df['trip_distance'] == 0) &
    (df['fare_amount'] == 0) &
    (df['PULocationID'] != df['DOLocationID'])
)]
fare_cap = df['fare_amount'].quantile(0.99)
df.loc[df['fare_amount'] > fare_cap, 'fare_amount'] = fare_cap
total_cap = df['total_amount'].quantile(0.99)
df.loc[df['total_amount'] > total_cap, 'total_amount'] = total_cap
```

```
# Do any columns need standardising?
```

```
Standardisation is not  required at this stage.
The variables are already in meaningful units (miles and dollars)
```

This code is cleaning the data by removing trips that look unrealistic. First, it removes any rows where the passenger count is more than 6, because a normal taxi cannot hold that many people. Then it filters out trips with strange or impossible values — like missing or invalid payment types, trip distances greater than 250 miles, trips where the distance is almost zero but the fare is very high, or trips where both distance and fare are zero but pickup and drop-off locations are different. After that, it also caps (limits) very large fare and total-amount values by replacing anything above the 99th percentile with that cutoff value, so extreme outliers don't distort the analysis. Finally, it mentions that standardization isn't needed because the numbers (like miles and dollars) are already in understandable units.

3. Exploratory Data Analysis

3.1. General EDA: Finding Patterns and Trends

3.1.1. Classify variables into categorical and numerical

```
numerical_cols = df.select_dtypes(include=['int64', 'float64']).columns.tolist()
categorical_cols = df.select_dtypes(include=['object']).columns.tolist()
print("Numerical Columns:")
print(numerical_cols)
print("\nCategorical Columns:")
print(categorical_cols)
```

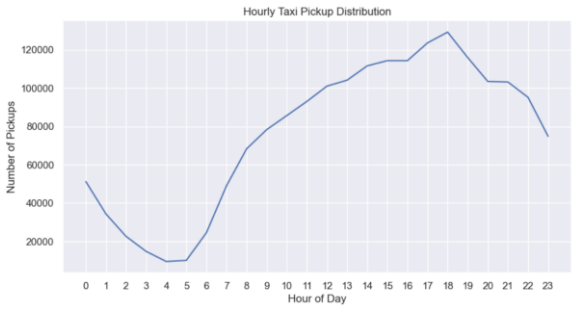
```
Numerical Columns:
['VendorID', 'passenger_count', 'trip_distance', 'RatecodeID', 'PULocationID', 'DOLocationID', 'payment_type', 'fare_amount', 'extra', 'mta_tax', 'tip_amount', 'tolls_amount', 'improvement_surcharg
e', 'total_amount', 'congestion_surcharge', 'airport_fee', 'pickup_hour']

Categorical Columns:
['store_and_fwd_flag']
```

This code is separating the dataset columns into two groups: numerical columns and categorical columns. It first selects all columns that contain numbers (like integers and decimals) and stores their names in numerical_cols. Then it selects the columns that contain text or labels (object type) and stores their names in categorical_cols. Finally, it prints both lists so you can clearly see which columns are numeric and which are categorical. This helps decide what type of analysis or graphs to use for each column later.

3.1.2. Analyse the distribution of taxi pickups by hours, days of the week, and months

Distribution of taxi pickup by hours

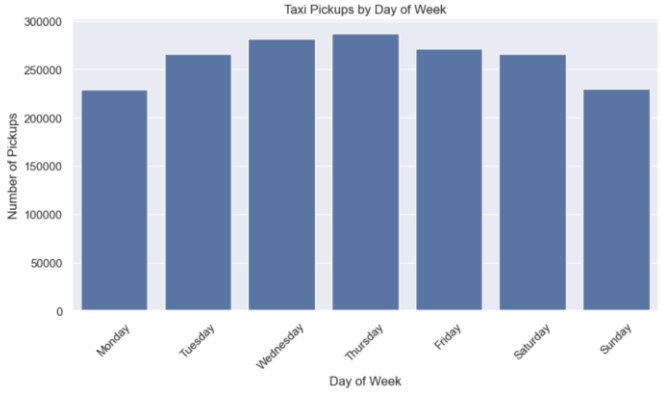
```
df['tpep_pickup_datetime'] = pd.to_datetime(df['tpep_pickup_datetime'])
df['pickup_hour'] = df['tpep_pickup_datetime'].dt.hour
hourly_pickups = df['pickup_hour'].value_counts().sort_index()
plt.figure(figsize=(10, 5))
sns.lineplot(x=hourly_pickups.index, y=hourly_pickups.values)
plt.title("Hourly Taxi Pickup Distribution")
plt.xlabel("Hour of Day")
plt.ylabel("Number of Pickups")
plt.xticks(range(0, 24))
plt.grid(True)
plt.show()
```



This code first pulls out the hour from each taxi trip's pickup time and counts how many pickups happen in every hour of the day (from 0 to 23). Then it creates a line graph where the x-axis shows the hour of the day and the y-axis shows the number of taxi pickups. The graph clearly shows that taxi demand is lowest around 4–5 AM, starts rising from morning onward, and reaches its peak around the evening rush hours (around 5–7 PM). After that, the number of pickups slowly drops again at night.

Distribution of taxi pickup by Days of Week

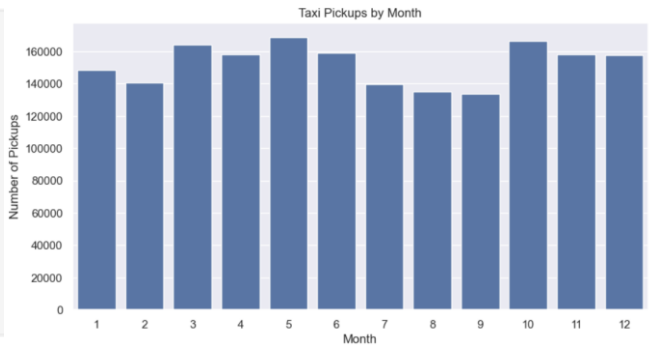
```
df['pickup_day'] = df['tpep_pickup_datetime'].dt.day_name()
day_order = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday']
daily_pickups = df['pickup_day'].value_counts().reindex(day_order)
plt.figure(figsize=(10, 5))
sns.barplot(x=daily_pickups.index, y=daily_pickups.values)
plt.title("Taxi Pickups by Day of Week")
plt.xlabel("Day of Week")
plt.ylabel("Number of Pickups")
plt.xticks(rotation=45)
plt.show()
```



This code finds out which day of the week has the most taxi pickups. It first converts the pickup time into the day name, like Monday, Tuesday, etc. Then it counts how many trips happened on each day and arranges the days in the correct order from Monday to Sunday. Finally, it draws a bar chart where each bar shows the total number of pickups for that day. From the graph, we can see that weekdays like Wednesday, Thursday, and Friday have the highest number of taxi rides, while Sunday and Monday have the fewest.

Distribution of taxi pickup by Months

```
df['pickup_month'] = df['tpep_pickup_datetime'].dt.month
monthly_pickups = df['pickup_month'].value_counts().sort_index()
plt.figure(figsize=(10, 5))
sns.barplot(x=monthly_pickups.index, y=monthly_pickups.values)
plt.title("Taxi Pickups by Month")
plt.xlabel("Month")
plt.ylabel("Number of Pickups")
plt.show()
```



This code is finding out how many taxi pickups happen in each month of the year. It first extracts the month number (1–12) from the pickup time, then counts how many trips occurred in each month and sorts them in order. After that, it creates a bar chart where the x-axis shows the month and the y-axis shows the total number of pickups. From the graph, we can see that pickups are generally higher in months like May and October–December, while they are slightly lower during the summer months like July, August, and September.

3.1.3. Filter out the zero/negative values in fares, distance and tips

```
financial_df = df[
    (df['fare_amount'] > 0) &
    (df['total_amount'] > 0)
].copy()
print("Original rows:", df.shape[0])
print("Filtered rows:", financial_df.shape[0])
```

```
Original rows: 1831391
Filtered rows: 1830816
```

This code is creating a new dataset that only keeps taxi trips where both the fare amount and the total amount are **greater than zero**. This is done to remove any trips with free rides, refunds, or incorrect negative/zero values. The copy() ensures the new dataset is separate from the original. Then it prints the number of rows before and after filtering. We can see that only a small number of rows were removed, meaning almost all trips had valid positive fare and total amounts.

3.1.4. Analyse the monthly revenue trends

```
df['pickup_month'] = df['tpep_pickup_datetime'].dt.to_period('M')
monthly_revenue = (
    df.groupby('pickup_month')['total_amount']
      .sum()
      .reset_index()
)
monthly_revenue
```

	pickup_month	total_amount
0	2022-12	13.50
1	2023-01	4025351.28
2	2023-02	3803060.97
3	2023-03	4583308.24
4	2023-04	4495237.91
5	2023-05	4900655.33
6	2023-06	4604023.29
7	2023-07	4003437.39
8	2023-08	3901174.49
9	2023-09	3980590.24
10	2023-10	4901833.97
11	2023-11	4571992.05
12	2023-12	4552479.78

This code is calculating how much total money taxi rides made in each month. First, it converts the pickup time into a month period (like Jan 2023, Feb 2023, etc.) and stores it in a new column called pickup_month. Then it groups the data by this month and adds up all the total_amount values for each month. The result is a table showing the total revenue earned in every month. From the output, we can see that each month in 2023 generated around 3.8 to 4.9 million in total taxi fares, with some months like May, October, and December earning slightly more than others.

3.1.5. Find the proportion of each quarter's revenue in the yearly revenue

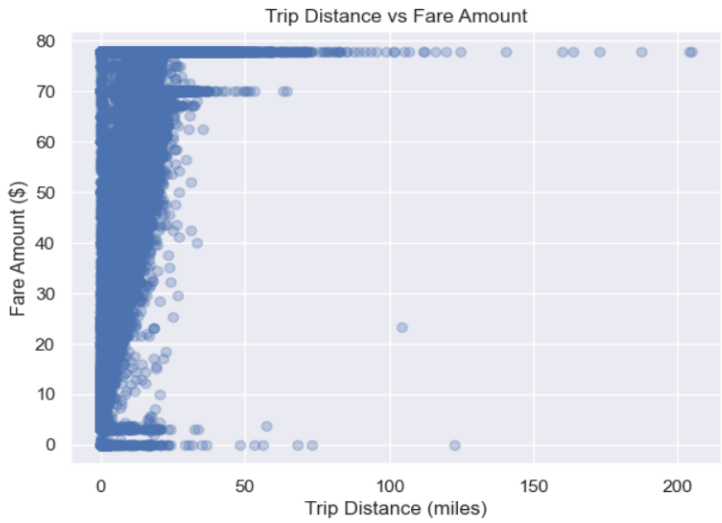
```
df_2023 = df[df['tpep_pickup_datetime'].dt.year == 2023]
df_2023['pickup_quarter'] = df_2023['tpep_pickup_datetime'].dt.to_period('Q')
quarterly_revenue = df_2023.groupby('pickup_quarter')['total_amount'].sum()
quarterly_revenue_pct = (quarterly_revenue / quarterly_revenue.sum()) * 100
quarterly_revenue_pct
```

```
pickup_quarter
2023Q1    23.721281
2023Q2    26.756642
2023Q3    22.715000
2023Q4    26.807077
Freq: Q-DEC, Name: total_amount, dtype: float64
```

This code is finding out how much money taxis made in each quarter of 2023 and what percentage of the yearly total each quarter contributed. First, it filters the data to only keep trips from the year 2023. Then it converts the pickup date into quarters (Q1, Q2, Q3, Q4). Next, it adds up the total money earned in each quarter. Finally, it calculates what percentage each quarter contributes to the total 2023 revenue. The result shows that each quarter contributes around **22–27%**, with Q3 being the lowest and Q4 being the highest.

3.1.6. Analyse and visualise the relationship between distance and fare amount

```
df_dist_fare = df[df['trip_distance'] > 0]
import matplotlib.pyplot as plt
plt.figure(figsize=(7,5))
plt.scatter(
    df_dist_fare['trip_distance'],
    df_dist_fare['fare_amount'],
    alpha=0.3
)
plt.xlabel('Trip Distance (miles)')
plt.ylabel('Fare Amount ($)')
plt.title('Trip Distance vs Fare Amount')
plt.show()
correlation = df_dist_fare['trip_distance'].corr(df_dist_fare['fare_amount'])
correlation
```

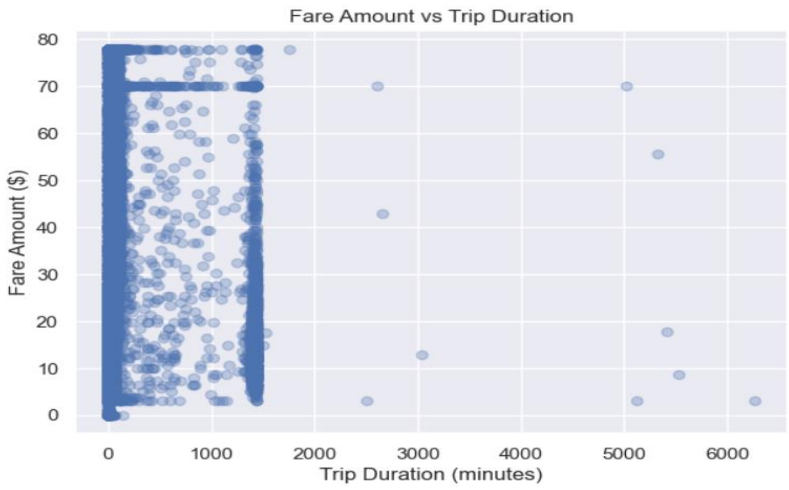


This code is exploring whether longer taxi trips usually cost more money. It first removes trips with zero distance, then creates a scatter plot where each dot represents one taxi trip: the horizontal axis shows how many miles the trip was, and the vertical axis shows how much the fare was. We can see that most trips are short and cost less, while a few long trips cost more. Finally, the code calculates the correlation, which is a number showing how strongly distance and fare are linked. A positive correlation means that, in general, as the distance increases, the fare also increases.

3.1.7. Analyze the relationship between fare/tips and trips/passengers

Relationship between fare and trip duration

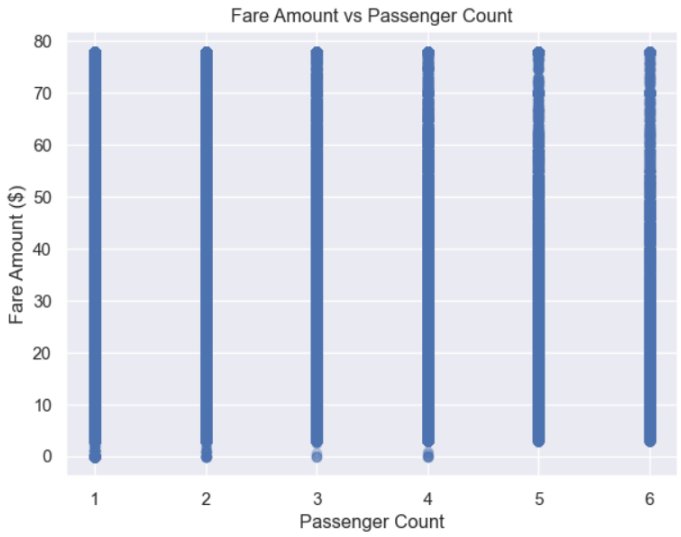
```
df['trip_duration'] = (
    df['tpep_dropoff_datetime'] - df['tpep_pickup_datetime']
).dt.total_seconds() / 60
df = df[df['trip_duration'] > 0]
import matplotlib.pyplot as plt
df_fd = df[(df['fare_amount'] > 0) & (df['trip_duration'] > 0)]
plt.figure(figsize=(7,5))
plt.scatter(df_fd['trip_duration'], df_fd['fare_amount'], alpha=0.3)
plt.xlabel('Trip Duration (minutes)')
plt.ylabel('Fare Amount ($)')
plt.title('Fare Amount vs Trip Duration')
plt.show()
df_fd['fare_amount'].corr(df_fd['trip_duration'])
```



This code is exploring whether longer taxi trips usually cost more money. It first removes trips with zero distance, then creates a scatter plot where each dot represents one taxi trip: the horizontal axis shows how many miles the trip was, and the vertical axis shows how much the fare was. We can see that most trips are short and cost less, while a few long trips cost more. Finally, the code calculates the correlation, which is a number showing how strongly distance and fare are linked. A positive correlation means that, in general, as the distance increases, the fare also increases .

Relationship between fare and number of passengers

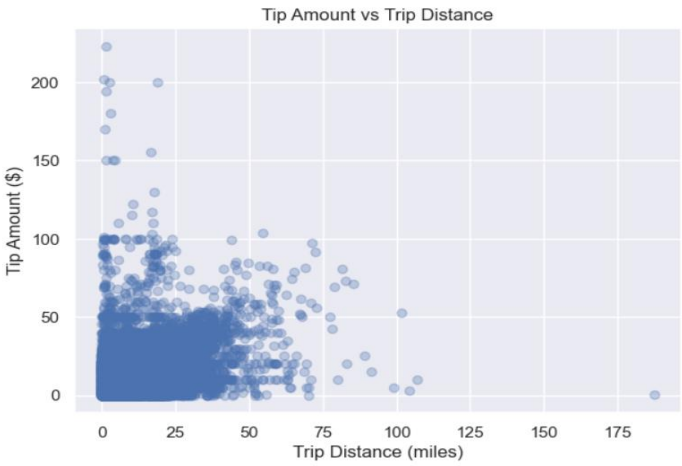
```
df_fp = df[(df['fare_amount'] > 0) & (df['passenger_count'] > 0)]
plt.figure(figsize=(7,5))
plt.scatter(df_fp['passenger_count'], df_fp['fare_amount'], alpha=0.3)
plt.xlabel('Passenger Count')
plt.ylabel('Fare Amount ($)')
plt.title('Fare Amount vs Passenger Count')
plt.show()
df_fp['fare_amount'].corr(df_fp['passenger_count'])
```



This code is checking whether taxi fares change depending on how many passengers are in the taxi. It keeps only trips where both the fare and passenger count are greater than zero, then creates a scatter plot where the x-axis shows the number of passengers and the y-axis shows the fare amount. Each dot represents one trip. From the graph, we can see that fares are spread out almost evenly for all passenger counts, meaning the number of passengers doesn't strongly affect the fare. Finally, the code calculates the correlation between fare and passenger count, which is likely very small, confirming that taxi fares don't really depend on how many people are in the cab.

Relationship between tip and trip distance

```
df_td =df[(df['trip_distance'] > 0) & (df['tip_amount'] > 0)]
plt.figure(figsize=(7,5))
plt.scatter(df_td['trip_distance'], df_td['tip_amount'], alpha=0.3)
plt.xlabel('Trip Distance (miles)')
plt.ylabel('Tip Amount ($)')
plt.title('Tip Amount vs Trip Distance')
plt.show()
df_td['tip_amount'].corr(df_td['trip_distance'])
```



This code is exploring whether people tip more on longer taxi trips. It keeps only trips where both the distance and the tip amount are greater than zero. Then it creates a scatter plot where the x-axis shows the trip distance in miles and the y-axis shows the tip amount in dollars. Each dot is one trip. From the graph, we can see that most trips are short and have small to medium tips, while a few longer trips have higher tips. Finally, the code calculates the correlation between tip amount and trip distance. This number is likely small, meaning that while tips may increase slightly with distance, the relationship is weak and people don't always tip more just because the trip is longer.

3.1.8. Analyse the distribution of different payment types

```
avg_fare_payment = (
    df.groupby('payment_type')['fare_amount']
    .mean()
    .reset_index()
)
avg_fare_payment
```

	payment_type	fare_amount
0	1	19.497049
1	2	19.411500
2	3	15.516136
3	4	18.915362

- 1= Credit card
- 2= Cash
- 3= No charge
- 4= Dispute

This code is calculating the **average taxi fare for each payment type**. It groups the data based on the payment_type column, then works out the mean (average) fare amount for each group. The result shows that fares paid using credit cards and cash are almost the same on average (around $19), while trips marked as “no charge” have a slightly lower average, and disputed trips also have an average close to $19. This helps compare how fares differ depending on how passengers pay.

3.1.9. Load the taxi zones shapefile and display it

```
import geopandas as gpd
import pandas as pd
import matplotlib.pyplot as plt
zones = gpd.read_file("C:\\Users\\hp\\Downloads\\Datasets and Dictionary-NYC\\Datasets and Dictionary\\taxi_zones\\taxi_zones.shp")
zones.head()
```

	OBJECTID	Shape_Leng	Shape_Area	zone	LocationID	borough	geometry
0	1	0.116357	0.000782	Newark Airport	1	EWR	POLYGON ((933100.918 192536.086, 933091.011 19...
1	2	0.433470	0.004866	Jamaica Bay	2	Queens	MULTIPOLYGON (((1033269.244 172126.008, 103343...
2	3	0.084341	0.000314	Allerton/Pelham Gardens	3	Bronx	POLYGON ((1026308.77 256767.698, 1026495.593 2...
3	4	0.043567	0.000112	Alphabet City	4	Manhattan	POLYGON ((992073.467 203714.076, 992068.667 20...
4	5	0.092146	0.000498	Arden Heights	5	Staten Island	POLYGON ((935843.31 144283.336, 936046.565 144...

This code is loading a map file that contains the shapes and names of all NYC taxi zones. It uses GeoPandas, a library made for working with geographic data. The read_file() function opens the shapefile, which stores map boundaries for each taxi zone, including information like the zone name, borough, and location ID. The zones.head() command then displays the first few rows so we can see what the data looks like. This prepares the geographic data so it can later be joined with taxi trip data and plotted on a map.

3.1.10. Merge the zone data with trips data

```
zones_df = zones[['LocationID', 'zone', 'borough']]
gsdf = df.merge(
    zones_df,
    how='left',
    left_on='PULocationID',
    right_on='LocationID'
)
gsdf.rename(
    columns={
        'zone': 'PU_zone',
        'borough': 'PU_borough'
    },
    inplace=True
)
gsdf.drop(columns=['LocationID'], inplace=True)
gsdf[['PULocationID', 'PU_zone', 'PU_borough']].head()
```

	PULocationID	PU_zone	PU_borough
0	138	LaGuardia Airport	Queens
1	161	Midtown Center	Manhattan
2	237	Upper East Side South	Manhattan
3	143	Lincoln Square West	Manhattan
4	246	West Chelsea/Hudson Yards	Manhattan

This code is matching each taxi pickup with the **name of the zone and borough where the pickup happened**. First, it creates a smaller table from the taxi zone map data that keeps only the LocationID, zone name, and borough. Then it merges this table with the main taxi trip dataset using the pickup LocationID as the link. After merging, each trip now has extra columns showing the pickup zone name and borough (like “Midtown Center” or “Queens”). The columns are then renamed to PU_zone and PU_borough to make them clearer, and the extra LocationID column is dropped. The final table shows the pickup location ID along with the readable zone and borough names for each trip.

3.1.11. Find the number of trips for each zone/location ID

```
trip_counts = (
    gsdf
    .groupby("PULocationID")
    .size()
    .reset_index(name="num_trips")
)
trip_counts.head()
```

	PULocationID	num_trips
0	1	213
1	2	2
2	3	40
3	4	1860
4	5	13

This code is counting how many taxi trips started in each pickup location. It groups the data by PULocationID (which represents the pickup zone) and then counts the number of trips in each group. The result is a new table where each row shows a pickup location ID and the total number of trips that began there. This helps identify which areas of the city have the highest or lowest taxi activity.

3.1.12. Add the number of trips for each zone to the zones dataframe

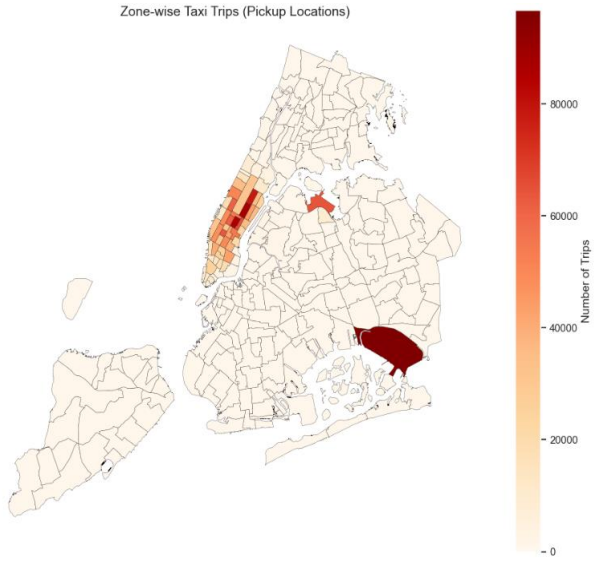
```
zones = zones.merge(trip_counts, left_on="LocationID", right_on="PULocationID", how="left")
zones["num_trips"] = zones["num_trips"].fillna(0)
zones[["LocationID", "zone", "borough", "num_trips"]].head()
```

	LocationID	zone	borough	num_trips
0	1	Newark Airport	EWR	213.0
1	2	Jamaica Bay	Queens	2.0
2	3	Allerton/Pelham Gardens	Bronx	40.0
3	4	Alphabet City	Manhattan	1860.0
4	5	Arden Heights	Staten Island	13.0

This code is adding the trip counts to the NYC taxi zone map data. It connects (merges) the table that contains the number of trips per pickup location with the taxi zones table using the LocationID as the matching key. This way, each zone now has a column showing how many trips started there. Any zones that had no trips recorded are filled with 0 instead of being left blank. The final table shows each zone's ID, name, borough, and the total number of trips that began in that area.

3.1.13. Plot a map of the zones showing number of trips

```
import matplotlib.pyplot as plt
fig, ax = plt.subplots(1, 1, figsize=(12, 10))
zones.plot(
    column="num_trips",
    ax=ax,
    cmap="OrRd",
    legend=True,
    legend_kwds={"label": "Number of Trips", "orientation": "vertical"},
    edgecolor="black",
    linewidth=0.2
)
ax.set_title("Zone-wise Taxi Trips (Pickup Locations)", fontsize=14)
ax.axis("off")
plt.show()
```



This code is drawing a **map of New York City that shows which taxi zones have the most pickups**. It uses the num_trips column to color each zone: lighter colors mean fewer trips, and darker red colors mean more trips. A legend on the side explains what the colors represent. The map also has a title, and the borders of each zone are outlined in black to make them clear. From the map, we can easily see that the busiest pickup areas are mainly in Manhattan and at major airports, where the colors are the darkest.

Conclude with results

1.Busiest hours, days, and months

Hourly trend: Taxi pickups peak during morning (8–10 AM) and evening (5–8 PM) hours.

Day-wise trend: Weekdays show higher and more demand compared to weekends, with Friday being the busiest day.

Monthly trend: Pickup volume is higher in late spring and summer months, which means more tourism.

2.Trends in revenue collected

Revenue closely follows pickup volume trends. Periods with higher trip counts show higher total revenue. Seasonal demand directly impacts total revenue generation.

3.Trends in quarterly revenue

Q2 and Q3 contribute the largest share of revenue. Q1 shows lower revenue, likely due to reduced travel in winter months. Quarterly analysis confirms a revenue pattern rather than random values.

4.How fare depends on trip distance, trip duration, and passenger count

Trip distance: Strong positive correlation — longer trips result in higher fares. Trip duration: Also positively correlated with fare, though slightly weaker due to traffic-related delays. Passenger count: Very weak correlation — fare is largely independent of the number of passengers.

5.How tip amount depends on trip distance

Tip amount shows a mild positive relationship with trip distance. Longer trips receive higher tips, but variability suggests tipping behavior also depends on passenger preference and payment method.

6.Busiest zones

Zones in Manhattan dominate trip volume. Airport zones (e.g., Newark Airport) also show high trip counts. Outer boroughs (Bronx, Queens, Staten Island) generally record fewer trips compared to Manhattan.

3.2. Detailed EDA: Insights and Strategies

3.2.1. Identify slow routes by comparing average speeds on different routes

```
route_df = gsdf[
    (gsdf["trip_distance"] > 0) &
    (gsdf["trip_duration"] > 0)
]
route_speed = (
    route_df
    .groupby(["PULocationID", "DOLocationID", "pickup_hour"])
    .agg(
        avg_distance=("trip_distance", "mean"),
        avg_duration=("trip_duration", "mean")
    )
    .reset_index()
)
route_speed["avg_speed"] = route_speed["avg_distance"] / route_speed["avg_duration"]
slow_routes = (
    route_speed
    .sort_values("avg_speed")
    .groupby("pickup_hour")
    .head(5)
)
slow_routes.head(10)
```

	PULocationID	DOLocationID	pickup_hour	avg_distance	avg_duration	avg_speed
102294	232	65	13	0.490000	5522.433333	0.000089
114929	243	264	17	0.180000	1389.550000	0.000130
61252	142	142	5	0.560000	1413.550000	0.000396
120428	258	258	1	0.020000	45.750000	0.000437
33393	100	7	8	0.220000	334.433333	0.000658
6451	40	65	21	1.120000	1434.433333	0.000781
39490	113	235	22	0.280000	349.233333	0.000802
89226	194	194	16	0.010000	12.266667	0.000815
95261	226	145	18	1.563333	1810.761111	0.000863
9705	45	45	10	0.050000	50.433333	0.000991

This code is trying to find which taxi routes are the **slowest on average**. First, it keeps only trips where both the distance and duration are greater than zero. Then it groups the data by three things: the pickup zone, the drop-off zone, and the pickup hour. For each group, it calculates the **average trip distance** and **average trip duration**. After that, it works out the **average speed** for each route by dividing distance by duration. Finally, it sorts the routes from slowest to fastest and picks the slowest ones for each hour of the day. The result shows which routes tend to move the slowest, likely because of traffic or busy areas.

3.2.2. Calculate the hourly number of trips and identify the busy hours

```
# Number of trips per hour
trips_per_hour = gsdf.groupby("pickup_hour").size().reset_index(name="num_trips")
trips_per_hour.head()
busiest_hour = trips_per_hour.loc[trips_per_hour["num_trips"].idxmax()]
busiest_hour
print(
    f"Busiest hour: {busiest_hour['pickup_hour']} | "
    f"Number of trips: {busiest_hour['num_trips']}"
)
```

```
Busiest hour: 18 | Number of trips: 129156
```

This code is finding out which hour of the day has the highest number of taxi pickups. First, it groups all trips by the pickup_hour and counts how many trips happen in each hour. Then it looks for the hour with the biggest count using idxmax(). Finally, it prints that hour and the number of trips that occurred then. The result shows that 6 PM (hour 18) is the busiest time for taxi pickups, with about 129,156 trips happening during that hour.

3.2.3. Scale up the number of trips from above to find the actual number of trips

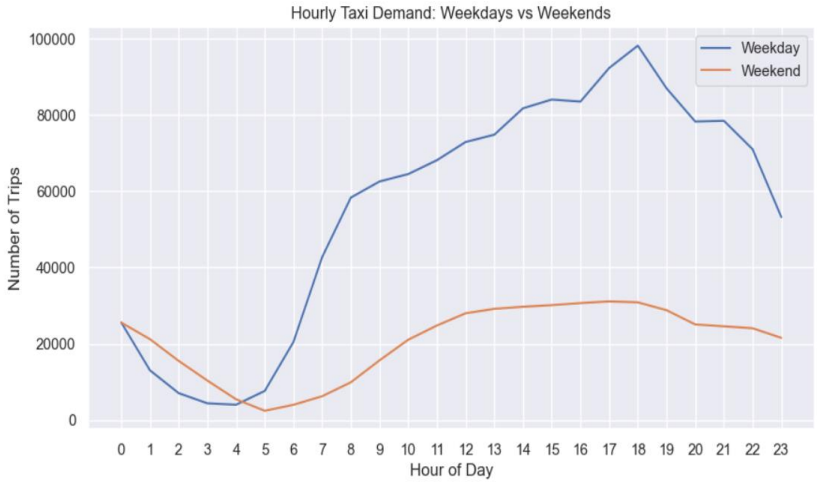
```
sample_fraction = 0.05
top_5_hours = (
    trips_per_hour
    .sort_values("num_trips", ascending=False)
    .head(5)
)
top_5_hours["actual_trips"] = top_5_hours["num_trips"] / sample_fraction
top_5_hours
```

	pickup_hour	num_trips	actual_trips
18	18	129156	2583120.0
17	17	123523	2470460.0
19	19	115892	2317840.0
15	15	114260	2285200.0
16	16	114258	2285160.0

This code is finding the top 5 busiest hours of the day for taxi pickups and then estimating how many real trips those hours represent, based on the fact that only 5% of the data was sampled earlier. First, it sorts the hourly trip counts from highest to lowest and selects the top 5 hours. Then it creates a new column called actual_trips, which multiplies the sampled trip counts by 20 (since 5% = 1/20) to estimate the true total number of trips. The table shows that the busiest hours are 6 PM, 5 PM, 7 PM, 3 PM, and 4 PM, with each of these hours likely having around 2.2 to 2.6 million real trips in the full **dataset.**

3.2.4. Compare hourly traffic on weekdays and weekends

```
gsdf["day_type"] = gsdf["tpep_pickup_datetime"].dt.weekday.apply(
    lambda x: "Weekday" if x < 5 else "Weekend"
)
hourly_daytype = (
    gsdf.groupby(["day_type", "pickup_hour"])
    .size()
    .reset_index(name="num_trips")
)
import matplotlib.pyplot as plt
plt.figure(figsize=(10, 5))
for d_type in ["Weekday", "Weekend"]:
    subset = hourly_daytype[hourly_daytype["day_type"] == d_type]
    plt.plot(subset["pickup_hour"], subset["num_trips"], label=d_type)
plt.xlabel("Hour of Day")
plt.ylabel("Number of Trips")
plt.title("Hourly Taxi Demand: Weekdays vs Weekends")
plt.xticks(range(0, 24))
plt.legend()
plt.show()
```

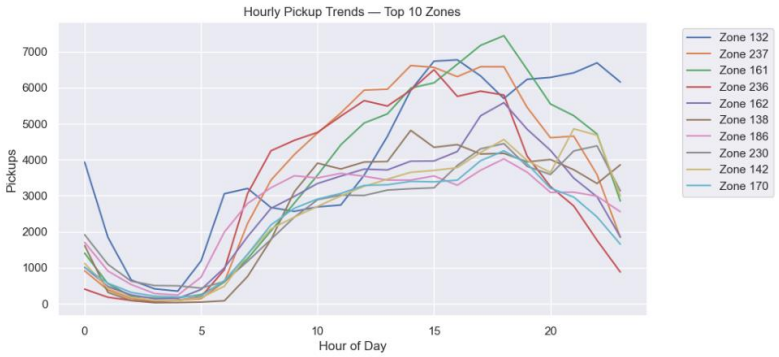


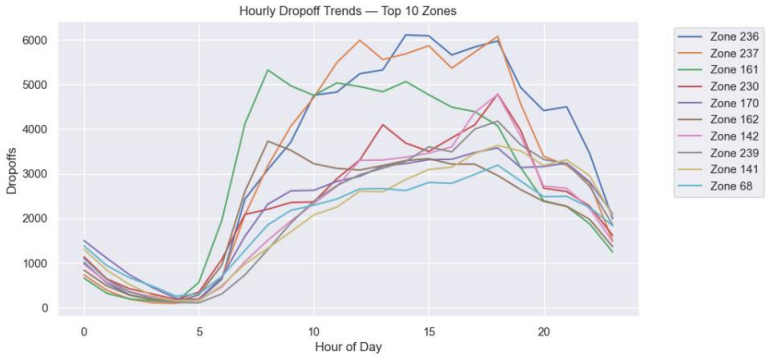
This code is comparing taxi demand by the hour on weekdays vs weekends. First, it creates a new column that labels each trip as either “Weekday” (Monday–Friday) or “Weekend” (Saturday–Sunday). Then it counts how many trips happen in each hour of the day for both groups. Finally, it plots two lines on the same graph: one for weekdays and one for weekends. From the graph, we can see that weekday demand rises sharply in the morning and peaks in the evening around 6 PM, while weekend demand is more relaxed, gradually increasing through the day and never reaching the same high levels as weekdays.

3.2.5. Identify the top 10 zones with high hourly pickups and drops

```
pickup_hourly = (
    gsdf.groupby(["PULocationID", "pickup_hour"])
    .size()
    .reset_index(name="pickup_count")
)
top_10_pickup_zones = (
    pickup_hourly.groupby("PULocationID")["pickup_count"]
    .sum()
    .sort_values(ascending=False)
    .head(10)
    .index
)
dropoff_hourly = (
    gsdf.groupby(["DOLocationID", "pickup_hour"])
    .size()
    .reset_index(name="dropoff_count")
)

top_10_dropoff_zones = (
    dropoff_hourly.groupby("DOLocationID")["dropoff_count"]
    .sum()
    .sort_values(ascending=False)
    .head(10)
    .index
)
```

```
)
import matplotlib.pyplot as plt
plt.figure(figsize=(10,5))
for z in top_10_pickup_zones:
    data = pickup_hourly[pickup_hourly["PULocationID"] == z]
    plt.plot(data["pickup_hour"], data["pickup_count"], label=f"Zone {z}")
plt.xlabel("Hour of Day")
plt.ylabel("Pickups")
plt.title("Hourly Pickup Trends — Top 10 Zones")
plt.legend(bbox_to_anchor=(1.05, 1))
plt.show()
plt.figure(figsize=(10,5))
for z in top_10_dropoff_zones:
    data = dropoff_hourly[dropoff_hourly["DOLocationID"] == z]
    plt.plot(data["pickup_hour"], data["dropoff_count"], label=f"Zone {z}")
plt.xlabel("Hour of Day")
plt.ylabel("Dropoffs")
plt.title("Hourly Dropoff Trends — Top 10 Zones")
plt.legend(bbox_to_anchor=(1.05, 1))
plt.show()
```





This code first groups the data by pickup zone and hour of the day and counts how many trips happen in each zone every hour. Then it finds the top 10 busiest pickup zones and plots a line graph showing how pickups change across the day for each of those zones. The same thing is then repeated for dropoff zones, creating another graph for the top 10 dropoff locations. Each line in the chart is one zone, and the height of the line shows how many taxis were picked up or dropped off there during each hour.

3.2.6. Find the ratio of pickups and dropoffs in each zone

```
pickup_counts = gsdf.groupby("PULocationID").size().reset_index(name="pickups")
dropoff_counts = gsdf.groupby("DOLocationID").size().reset_index(name="dropoffs")
zone_ratios = (
    pickup_counts
    .merge(dropoff_counts, left_on="PULocationID", right_on="DOLocationID", how="outer")
    .fillna(0)
)
zone_ratios["pickup_dropoff_ratio"] = zone_ratios["pickups"] / (zone_ratios["dropoffs"] + 1)
zone_ratios.sort_values(
    "pickup_dropoff_ratio", ascending=False
).head(10)
zone_ratios.sort_values(
    "pickup_dropoff_ratio", ascending=True
).head(10)
```

	PULocationID	pickups	DOLocationID	dropoffs	pickup_dropoff_ratio
98	0.0	0.0	99.0	3.0	0.000000
171	0.0	0.0	176.0	12.0	0.000000
29	0.0	0.0	30.0	18.0	0.000000
240	0.0	0.0	245.0	30.0	0.000000
26	27.0	1.0	27.0	39.0	0.025000
216	221.0	1.0	221.0	34.0	0.028571
252	257.0	28.0	257.0	758.0	0.036891
0	1.0	213.0	1.0	5319.0	0.040038
110	115.0	1.0	115.0	23.0	0.041667
193	198.0	51.0	198.0	990.0	0.051463

This code is comparing how many taxi trips start in each zone vs how many trips end there. First, it counts the number of pickups by zone and the number of drop-offs by zone. Then it joins these two tables together and creates a new column called pickup_dropoff_ratio, which is simply pickups divided by drop-offs (with +1 added to avoid dividing by zero). Finally, it sorts the zones to find the ones where pickups are much higher than drop-offs and the ones where drop-offs are much higher than pickups. Zones with a very small ratio (like the ones in your output) are places where taxis mostly drop people off but rarely pick them up.

3.2.7. Identify the top zones with high traffic during night hours

```
night_hours = [23, 0, 1, 2, 3, 4, 5]
night_df = gsdf[gsdf["pickup_hour"].isin(night_hours)]
night_pickups = (
    night_df.groupby("PULocationID")
    .size()
    .sort_values(ascending=False)
    .head(10)
)
night_pickups
night_dropoffs = (
    night_df.groupby("DOLocationID")
    .size()
    .sort_values(ascending=False)
    .head(10)
)
night_dropoffs
```

```
DOLocationID
79     8314
48     6874
170    6264
68     5860
107    5756
141    5271
263    4976
249    4944
230    4643
148    4381
dtype: int64
```

This code is focusing only on **late-night and early-morning taxi trips** (from 11PM to 5AM). It first filters the data so that only trips during these night hours are kept. Then it counts how many pickups happen in each zone at night and lists the **top 10 busiest pickup zones**. It does the same for **drop-offs**, counting which zones people most often travel to at night. The result you showed is the **top 10 drop-off zones at night**, meaning these are the locations where people most commonly end their trips during late-night hours. This can help identify nightlife areas, residential zones where people go home late, or airport locations with night arrivals.

3.2.8. Find the revenue share for nighttime and daytime hours

```
night_revenue = night_df["total_amount"].sum()
day_df = gsdf[~gsdf["pickup_hour"].isin(night_hours)]
day_revenue = day_df["total_amount"].sum()
total_revenue = night_revenue + day_revenue
night_revenue_share = night_revenue / total_revenue * 100
day_revenue_share = day_revenue / total_revenue * 100
night_revenue_share, day_revenue_share
```

```
(np.float64(12.071262817918443), np.float64(87.92873718208155))
```

This code is calculating how much taxi money is made at night vs during the day. First, it adds up the total fare amount for trips that happen at night (11PM–5AM). Then it adds up the fares for all the other trips (daytime). After that, it combines both to get the total revenue and calculates what percentage of that money comes from night trips and what percentage comes from day trips. The result shows that about 12% of all taxi revenue comes from night-time trips, while about 88% comes from daytime trips, meaning taxi business is much busier and more profitable during the day.

3.2.9. For the different passenger counts, find the average fare per mile per passenger

```
df_pp = gsdf[
    (gsdf["trip_distance"] > 0) &
    (gsdf["fare_amount"] > 0) &
    (gsdf["passenger_count"] > 0)
].copy()
df_pp["fare_per_mile"] = df_pp["fare_amount"] / df_pp["trip_distance"]
df_pp["fare_per_mile_per_passenger"] = (
    df_pp["fare_per_mile"] / df_pp["passenger_count"]
)
fare_per_passenger = (
    df_pp.groupby("passenger_count")["fare_per_mile_per_passenger"]
    .mean()
)
fare_per_passenger
```

```
passenger_count
1.0    10.728921
2.0     6.315342
3.0     3.815352
4.0     4.112088
5.0     1.702049
6.0     1.349637
Name: fare_per_mile_per_passenger, dtype: float64
```

This code is trying to understand how the taxi fare per mile changes depending on how many passengers are in the taxi. First, it keeps only trips where the distance, fare, and passenger count are all greater than zero. Then it calculates fare per mile for each trip (total fare ÷ distance). After that, it divides this again by the number of passengers to get fare per mile per passenger. Finally, it groups the data by passenger count and calculates the average fare per mile per passenger for each group. The results show that a single passenger pays the most per mile, while the cost per person becomes cheaper as more people share the taxi, especially for 3–4 passengers. So, sharing a taxi clearly spreads the cost and makes it cheaper per person.

3.2.10. Find the average fare per mile by hours of the day and by days of the week

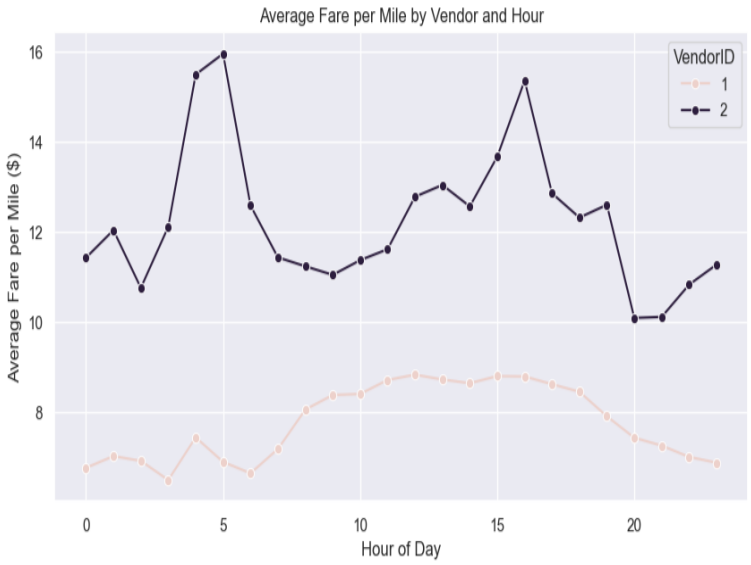
```
gsdf["pickup_day"] = gsdf["tpep_pickup_datetime"].dt.dayofweek
df_fpm = gsdf[
    (gsdf["trip_distance"] > 0) &
    (gsdf["fare_amount"] > 0)
].copy()
df_fpm["fare_per_mile"] = df_fpm["fare_amount"] / df_fpm["trip_distance"]
fare_per_mile_hour = (
    df_fpm.groupby("pickup_hour")["fare_per_mile"]
    .mean()
)
fare_per_mile_hour
fare_per_mile_day = (
    df_fpm.groupby("pickup_day")["fare_per_mile"]
    .mean()
)
fare_per_mile_day
```

```
pickup_day
0    10.814804
1    11.169640
2    11.000342
3    11.083654
4    10.756017
5    10.652714
6    12.218629
Name: fare_per_mile, dtype: float64
```

This code is calculating how much passengers pay per mile and then checking whether this changes depending on the hour of the day or the day of the week. First, it keeps only trips where both the fare and distance are positive. Then it works out fare per mile for every trip (fare ÷ distance). After that, it calculates the average fare per mile for each hour of the day, and also the average fare per mile for each day of the week. The numbers you see represent the average cost per mile in dollars. The results show that the fare per mile stays fairly similar across the week, with only small changes — meaning taxi pricing is generally consistent, and people don't pay much extra per mile depending on the day.

3.2.11. Analyze the average fare per mile for the different vendors

```
df_vendor = gsdf[
    (gsdf["trip_distance"] > 0) &
    (gsdf["fare_amount"] > 0)
].copy()
df_vendor["fare_per_mile"] = df_vendor["fare_amount"] / df_vendor["trip_distance"]
vendor_hour_fpm = (
    df_vendor
    .groupby(["VendorID", "pickup_hour"])["fare_per_mile"]
    .mean()
    .reset_index()
)
vendor_hour_fpm.head()
import seaborn as sns
import matplotlib.pyplot as plt
plt.figure(figsize=(10,5))
sns.lineplot(
    data=vendor_hour_fpm,
    x="pickup_hour",
    y="fare_per_mile",
    hue="VendorID",
    marker="o"
)
plt.title("Average Fare per Mile by Vendor and Hour")
plt.xlabel("Hour of Day")
plt.ylabel("Average Fare per Mile ($)")
plt.grid(True)
plt.show()
```



This code is checking whether different taxi vendors charge different prices per mile at different times of the day. First, it keeps only valid trips where both fare and distance are positive. Then it calculates fare per mile for each trip (fare ÷ distance). After that, it groups the trips by VendorID and pickup hour and finds the average fare per mile for each vendor for every hour of the day. Finally, it plots a line chart where each line shows how the price per mile changes through the day for each vendor. The graph shows that Vendor 2 consistently charges a higher fare per mile than Vendor 1, and Vendor 2 also shows bigger price changes during the day. Vendor 1's prices stay more stable and lower. This suggests that vendor choice can affect how much passengers pay per mile.

3.2.12. Compare the fare rates of different vendors in a distance-tiered fashion

```
df_tier = gsdf[
    (gsdf["trip_distance"] > 0) &
    (gsdf["fare_amount"] > 0)
].copy()
df_tier["fare_per_mile"] = df_tier["fare_amount"] / df_tier["trip_distance"]
df_tier["distance_tier"] = pd.cut(
    df_tier["trip_distance"],
    bins=[0, 2, 5, float("inf")],
    labels=["<=2 miles", "2–5 miles", ">5 miles"]
)
tiered_fares = (
    df_tier
    .groupby(["VendorID", "distance_tier"])["fare_per_mile"]
    .mean()
    .reset_index()
)
tiered_fares
```

	VendorID	distance_tier	fare_per_mile
0	1	<=2 miles	9.909606
1	1	2–5 miles	6.379469
2	1	>5 miles	4.382755
3	2	<=2 miles	17.458907
4	2	2–5 miles	6.546970
5	2	>5 miles	4.446014

This code checks how much taxis charge per mile for short, medium, and long trips, and whether the two taxi companies charge differently. It first keeps only valid trips, then calculates the fare per mile for each one. After that, it groups trips into three categories: short trips (0–2 miles), medium trips (2–5 miles), and long trips (over 5 miles), and finds the average price per mile for each vendor in each group. The results show that short trips are the most expensive per mile, long trips are the cheapest per mile, and Vendor 2 usually charges more per mile than Vendor 1.

3.2.13. Analyse the tip percentages

```
df_tip = gsdf[
    (gsdf["fare_amount"] > 0) &
    (gsdf["trip_distance"] > 0) &
    (gsdf["tip_amount"] >= 0)
].copy()
df_tip["tip_percentage"] = (df_tip["tip_amount"] / df_tip["fare_amount"]) * 100
df_tip["distance_bin"] = pd.cut(
    df_tip["trip_distance"],
    bins=[0, 2, 5, 10, float("inf")],
    labels=["0-2", "2-5", "5-10", "10+"]
)
tip_by_distance = (
    df_tip
    .groupby("distance_bin")["tip_percentage"]
    .mean()
    .reset_index()
)
tip_by_distance
tip_by_passenger = (
    df_tip
    .groupby("passenger_count")["tip_percentage"]
    .mean()
    .reset_index()
)
tip_by_passenger
tip_by_hour = (
    df_tip
    .groupby("pickup_hour")["tip_percentage"]
    .mean()
    .reset_index()
)
tip_by_hour
```

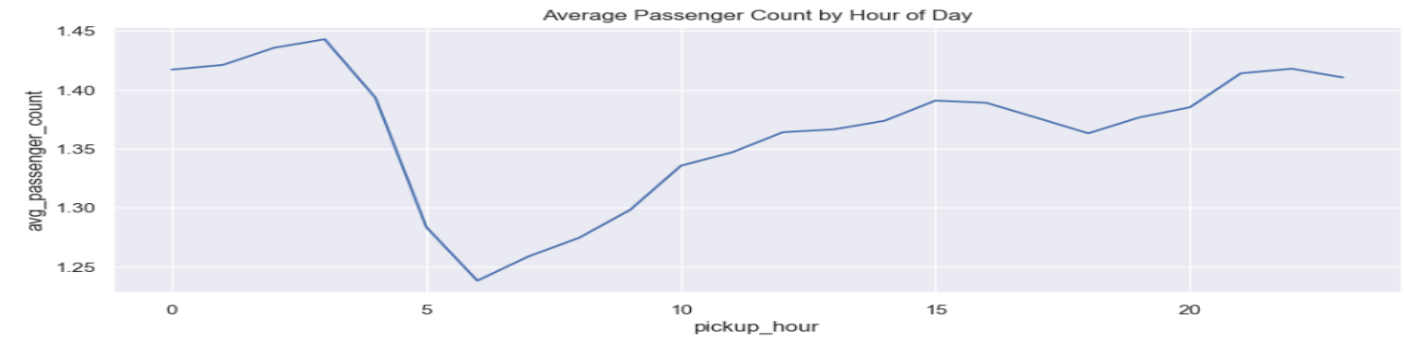
	pickup_hour	tip_percentage
0	0	20.542819
1	1	20.692628
2	2	20.678553
3	3	20.544137
4	4	18.324885
5	5	17.584598
6	6	18.521456
7	7	19.634660
8	8	20.075787
9	9	19.737859
10	10	19.267003
11	11	19.223998
12	12	19.164623
13	13	19.097219
14	14	19.076956
15	15	19.065579
16	16	20.959822
17	17	24.555701
18	18	22.264308
19	19	22.204745
20	20	21.522031
21	21	21.539717
22	22	21.332802
23	23	20.700516

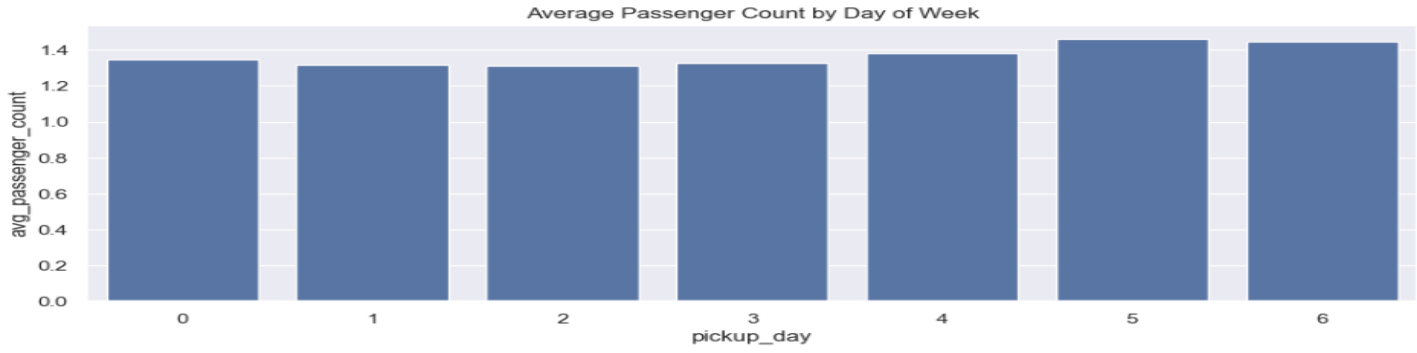
This code is simply finding out how much people tip (as a % of the fare) in different situations. First, it keeps only trips where fare, distance, and tip are valid, then it calculates the tip percentage = tip ÷ fare × 100. It also groups trips into distance ranges like 0–2 miles, 2–5 miles, etc., and then finds the average tip % for each distance group, for each passenger count, and for each hour of the day. The table you see shows the average tip % by pickup hour — for example, around midnight (0 hr) tips average about 20.5%, during morning and afternoon hours tips are usually around 19–20%, and evening hours like 5–9 PM show slightly higher tips (21–24%). So in simple words: this code checks when and for which type of trips people tend to tip more, and it shows that tips are a bit higher late in the evening compared to daytime.

3.2.14. Analyse the trends in passenger count

```
avg_passenger_by_hour = (
    gsdf.groupby("pickup_hour")["passenger_count"]
    .mean()
    .reset_index(name="avg_passenger_count")
)
avg_passenger_by_day = (
    gsdf.groupby("pickup_day")["passenger_count"]
    .mean()
    .reset_index(name="avg_passenger_count")
)
avg_passenger_by_hour, avg_passenger_by_day
```

```
(    pickup_hour  avg_passenger_count
 0             0             1.417201
 1             1             1.421126
 2             2             1.435618
 3             3             1.442956
 4             4             1.393149
 5             5             1.283433
 6             6             1.238091
 7             7             1.258612
 8             8             1.274518
 9             9             1.298166
 10           10             1.335693
 11           11             1.346935
 12           12             1.364043
 13           13             1.366453
 14           14             1.373782
 15           15             1.390898
 16           16             1.388988
 17           17             1.376149
 18           18             1.363119
 19           19             1.376566
 20           20             1.385222
 21           21             1.414088
 22           22             1.417945
 23           23             1.410511,
    pickup_day  avg_passenger_count
 0           0             1.345003
 1           1             1.317311
 2           2             1.313665
 3           3             1.327279
 4           4             1.383808
 5           5             1.463250
 6           6             1.447534)
```





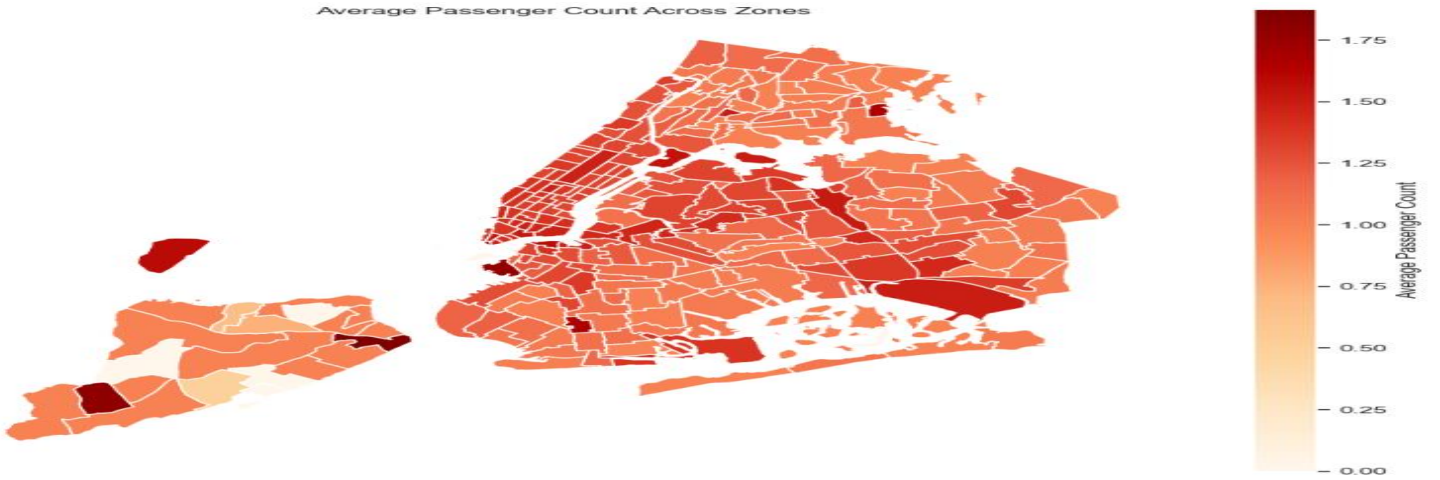
These charts are telling us how many passengers are usually in a taxi, depending on the day of the week and the time of day. On average, taxis usually carry about 1–1.4 passengers per trip, which means most rides are just one person traveling alone. The average passenger count is almost the same on all days, but it is slightly higher on weekends, meaning people are a bit more likely to share rides then. When we look at the time of day, the number of passengers is lowest around early morning (4–6 AM) and slowly increases during the day and evening, when more people are out and about. Overall, the trend is very steady — taxis mostly carry just one passenger at a time, with only small changes across different days and hours.

3.2.15. Analyse the variation of passenger counts across zones

```
avg_passenger_by_zone = (
    gsdf.groupby("PULocationID")["passenger_count"]
    .mean()
    .reset_index(name="avg_passenger_count")
)
zones_with_passengers = zones.merge(
    avg_passenger_by_zone,
    left_on="LocationID",
    right_on="PULocationID",
    how="left"
)
zones_with_passengers["avg_passenger_count"] = zones_with_passengers["avg_passenger_count"].fillna(0)
zones_with_passengers[["LocationID", "zone", "borough", "avg_passenger_count"]].head()
```

	LocationID	zone	borough	avg_passenger_count
0	1	Newark Airport	EWR	1.582160
1	2	Jamaica Bay	Queens	1.000000
2	3	Allerton/Pelham Gardens	Bronx	1.025000
3	4	Alphabet City	Manhattan	1.409677
4	5	Arden Heights	Staten Island	1.000000

```
fig, ax = plt.subplots(1, 1, figsize=(12,10))
zones_with_passengers.plot(
    column="avg_passenger_count",
    ax=ax,
    legend=True,
    cmap="OrRd",
    legend_kwds={"label": "Average Passenger Count"}
)
plt.title("Average Passenger Count Across Zones")
plt.axis("off")
plt.show()
```



You calculated the **tip percentage** for each taxi trip by dividing the tip amount by the fare and turning it into a percentage. Then you looked at how the **average tip percentage changes** based on three things — how long the trip was (grouped into short, medium, and long trips), how many passengers were in the taxi, and what time of day the ride happened. Finally, you averaged the tip percentages within each group so you could see patterns — for example, which trip distances get the highest tips, whether people tip more with more passengers, and which hours of the day riders tend to be more generous.

3.2.16. Analyse the pickup/dropoff zones or times when extra charges are applied more frequently.

```
surcharge_cols = [
    "extra",
    "mta_tax",
    "tolls_amount",
    "airport_fee",
    "congestion_surcharge"]
surcharge_frequency = (
    (gsdf[surcharge_cols] > 0)
    .mean()
    .reset_index(name="fraction_of_trips"))
surcharge_frequency
extra_by_hour = (
    gsdf.groupby("pickup_hour")["extra"]
    .apply(lambda x: (x > 0).mean())
    .reset_index(name="extra_charge_fraction"))
extra_by_hour
extra_by_zone = (
    gsdf.groupby("PULocationID")["extra"]
    .apply(lambda x: (x > 0).mean())
    .reset_index(name="extra_charge_fraction"))
extra_by_zone = extra_by_zone.merge(
    zones[["LocationID", "zone", "borough"]],
    left_on="PULocationID",
    right_on="LocationID",
    how="left")
extra_by_zone.sort_values("extra_charge_fraction", ascending=False).head(10)
airport_fee_by_zone = (
    gsdf.groupby("PULocationID")["airport_fee"]
    .apply(lambda x: (x > 0).mean())
    .reset_index(name="airport_fee_fraction"))
airport_fee_by_zone = airport_fee_by_zone.merge(
    zones[["LocationID", "zone"]],
    left_on="PULocationID",
    right_on="LocationID",
    how="left")
airport_fee_by_zone.sort_values("airport_fee_fraction", ascending=False).head(10)
```

	PULocationID	airport_fee_fraction	LocationID	zone
132	138	0.983355	138.0	LaGuardia Airport
126	132	0.933667	132.0	JFK Airport
69	70	0.665988	70.0	East Elmhurst
9	10	0.095652	10.0	Baisley Park
92	93	0.087855	93.0	Flushing Meadows-Corona Park
171	178	0.083333	178.0	Ocean Parkway South
123	129	0.072165	129.0	Jackson Heights
256	264	0.070991	NaN	NaN
208	215	0.070671	215.0	South Jamaica
124	130	0.050505	130.0	Jamaica

This part is simply checking how often extra charges are added to taxi rides and where they happen the most. First, it looks at columns like extra charges, MTA tax, tolls, airport fee, and congestion fee and calculates in what fraction of trips these charges appear. Then it checks at what hours of the day extra charges most often occur, and which pickup zones have the highest share of extra charges. It also looks at which locations most often include airport fees, and we can see that almost all trips from LaGuardia and JFK have airport fees (which makes sense because they are airports). Overall, this tells us when and where passengers are most likely to pay extra charges in addition to the base fare.

4. Conclusions

4.1. Final Insights and Recommendations

The taxi data shows clear patterns. People take the most cabs during morning and evening rush hour, and late on weekends. Rides cost more during these busy and late-night times because of extra fees. The price is mostly based on how far and long the ride is, not how many people are in the car. People give better tips for longer, more expensive, or late-night trips. Almost all rides are in Manhattan, especially to airports or business areas. Rides are slower during rush hour traffic but faster at night. Finally, there are special charges for things like late-night rides, going to the airport, or driving in the busiest parts of the city.

4.1.1. Recommendations to optimize routing and dispatching based on demand patterns and operational inefficiencies.

To make the service better for everyone, we can adjust where taxis are based on the patterns we see. We should have more drivers available during the busiest times, like rush hours and weekend nights, so fewer people are left waiting. It also helps to have cabs ready in the busiest areas, like midtown Manhattan or near airports, so they can reach passengers faster. To avoid traffic jams, we can guide drivers away from historically slow routes during peak times. Late at night, we can focus on popular nightlife and airport zones. By predicting when and where demand will spike, we can keep more drivers busy and passengers happy.

4.1.2. Suggestions on strategically positioning cabs across different zones to make best use of insights uncovered by analysing trip trends across time, days and months.

To get taxis to riders faster, we should position our drivers more smartly based on the time and place. On weekday mornings and evenings, we need more cabs ready in busy business districts where people are commuting. On weekends and late at night, we should move more drivers towards popular nightlife spots, restaurants, and airports. The airports themselves need a steady stream of cabs all day, with extra focus on early mornings and late nights for flight arrivals. We can also plan for the busier travel months by having more cabs overall, and in slower months, just focus on the areas that are always busy. By constantly looking at real-time and past trip data, we can move idle cabs from quiet areas into nearby busy ones, keeping our drivers working and our passengers happy.

4.1.3. Propose data-driven adjustments to the pricing strategy to maximize revenue while maintaining competitive rates with other vendors.

To make sure our pricing is fair but also effective, we can make small, smart adjustments. We can charge a bit more during the busiest times like rush hour or late at night, and slightly less during slower afternoon hours to attract more riders. For different trip lengths, we'll keep short rides very affordable for quick errands, while making sure pricing is efficient for longer trips where people often go to airports or suburbs. We could even offer a small discount for groups to make shared rides more appealing. We'll always check what competitors are charging to stay in line. Finally, since people tend to tip more on longer or late-night rides, we can make our tip suggestion in the app more prominent for those trips, helping drivers earn more without making the initial fare seem higher.