Report: Optimising NYC Taxi Operations

Include your visualisations, analysis, results, insights, and outcomes. Explain your methodology and approach to the tasks. Add your conclusions to the sections.

1. Data Preparation

1.1. Loading the dataset

1.1.1. Sample the data and combine the files

The first step is to import necessary libraries for visualisations and analysis, which are - Numpy, Pandas, Matplotlib, Seaborn. Along with them, the versions that have been installed are also shown as output.

```
# Import warnings
import warnings
warnings.filterwarnings('ignore')
```

```
# Import the libraries you will be using for analysis
!pip install numpy==1.26.4 pandas==2.2.2 matplotlib==3.10.0 seaborn==0.13.2 pyarrow
```

```
Requirement already satisfied: numpy==1.26.4 in /usr/local/lib/python3.12/dist-packages (1.26.4)
Requirement already satisfied: pandas==2.2.2 in /usr/local/lib/python3.12/dist-packages (2.2.2)
Requirement already satisfied: matplotlib==3.10.0 in /usr/local/lib/python3.12/dist-packages (3.10.0)
Requirement already satisfied: seaborn==0.13.2 in /usr/local/lib/python3.12/dist-packages (0.13.2)
Requirement already satisfied: pyarrow in /usr/local/lib/python3.12/dist-packages (18.1.0)
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/python3.12/dist-packages (from pandas==2.2.2) (2.9.0.post0)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.12/dist-packages (from pandas==2.2.2) (2025.2)
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.12/dist-packages (from pandas==2.2.2) (2025.2)
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.12/dist-packages (from matplotlib==3.10.0) (1.3.3)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.12/dist-packages (from matplotlib==3.10.0) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.12/dist-packages (from matplotlib==3.10.0) (4.60.1)
Requirement already satisfied: kiwisolver>=1.3.1 in /usr/local/lib/python3.12/dist-packages (from matplotlib==3.10.0) (1.4.9)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.12/dist-packages (from matplotlib==3.10.0) (25.0)
Requirement already satisfied: pillow>=8 in /usr/local/lib/python3.12/dist-packages (from matplotlib==3.10.0) (11.3.0)
Requirement already satisfied: pyparsing>=2.3.1 in /usr/local/lib/python3.12/dist-packages (from matplotlib==3.10.0) (3.2.5)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.12/dist-packages (from python-dateutil>=2.8.2->pandas==2.2.2) (1.17.0)
```

As the project is done in Google Colab, I have loaded the dataset into the platform by mounting my Google Drive first, as shown below

```
numpy version: 1.26.4
pandas version: 2.2.2
matplotlib version: 3.10.0
seaborn version: 0.13.2
```

```
# need to mount the google drive to read the files (need to download the files into google drive)
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

I have then loaded 1 parquet file, out of 12, as a dataframe and used the *.info()* method to display a summary of it.

```
# Try loading one file

df = pd.read_parquet('/content/drive/MyDrive/Vaishu EDA Assignment/dataset/yellow_tripdata_2023-01.parquet')
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3066766 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: 444.6+ MB
```

The output above shows that there are a total of 3066766 entries with 19 columns present in that particular dataset.

Questions

a) How many rows are there?
   Answer: The dataset contains 3 million rows, spread across 19 columns

b) Do you think handling such a large number of rows is computationally feasible when we have to combine the data for all twelve months into one?
   Answer: No, this dataset will take a lot of computing power to process.

c) To handle this, we need to sample a fraction of data from each of the files. How to go about that? Think of a way to select only some portion of the data from each month's file that accurately represents the trends.
   Answer: We aim to take a 5% sample of data from each month, meaning each file.

The final step is to combine the 12 parquet files into one, sample 5% of data from each file and then convert the merged file into CSV as it is easier to store and use directly, using the code shown below.

The “sample_5_percent” function takes a DataFrame as an input, calculates 5% of the group size. The “random_state = 42” parameter is used to ensure that the same random rows are present every time.

The for loop will read each file and save it into the “df_temp’ variable. Within it, I have converted the “tpep_pickup_datetime” column from object to datetime type and then extracted both the hour and month into 2 new columns called “pickup_hour” and “pickup_month”. This will make it easier to group data during sampling.

```
# Take a small percentage of entries from each hour of every date.
# Iterating through the monthly data:
#   read a month file -> day -> hour: append sampled data -> move to next hour -> move to next day after 24 hours -> move to next month file
# Create a single dataframe for the year combining all the monthly data

# Select the folder having data files
import os
#import pandas as pd

# Update this path to the correct location of your data files in Google Drive
data_folder_path = '/content/drive/MyDrive/Vaishu EDA Assignment/dataset'

# Create a list of all the parquet files for 2023
file_list = [f'yellow_tripdata_2023-{month:02d}.parquet' for month in range(1, 13)]

# initialise an empty dataframe
df = pd.DataFrame()

# iterate through the list of files and sample one by one:
for file_name in file_list:
    try:
        # file path for the current file
        file_path = os.path.join(data_folder_path, file_name)

        # Reading the current file
        monthly_df = pd.read_parquet(file_path)

        # We will store the sampled data for the current date in this df by appending the sampled data from each hour to this
        # After completing iteration through each date, we will append this data to the final dataframe.
        sampled_data_month = pd.DataFrame()

        # Extract date and hour for sampling
        monthly_df['pickup_date'] = monthly_df['tpep_pickup_datetime'].dt.date
        monthly_df['pickup_hour'] = monthly_df['tpep_pickup_datetime'].dt.hour

        # Get unique dates in the month
        unique_dates = monthly_df['pickup_date'].unique()

        # Loop through dates and then loop through every hour of each date
        for date in unique_dates:
            daily_df = monthly_df[monthly_df['pickup_date'] == date]
            # Iterate through each hour of the selected date
            for hour in range(24):
                hour_data = daily_df[daily_df['pickup_hour'] == hour]
                if not hour_data.empty:
                    # Sample 5% of the hourly data randomly
                    sample = hour_data.sample(frac=0.05, random_state=42)
                    # add data of this hour to the dataframe
                    sampled_data_month = pd.concat([sampled_data_month, sample])

        # Concatenate the sampled data of all the dates to a single dataframe
        df = pd.concat([df, sampled_data_month])

    except FileNotFoundError:
        print(f"Error: File not found: {file_name}. Skipping.")
    except Exception as e:
        print(f"Error reading file {file_name}: {e}")

# Drop the temporary columns
if 'pickup_date' in df.columns:
    df = df.drop(columns=['pickup_date', 'pickup_hour'])

print("Finished sampling and combining data.")
print(f"Total number of rows in combined dataframe: {len(df)}")
```

```
Finished sampling and combining data.
Total number of rows in combined dataframe: 1915511
```

“sampled_df_temp” is a temporary variable that first creates a group of rows for each combination and then it applies the “sample_5_percent”

function to it. “group_keys = False” helps to prevent the keys of the group being added to the index. This is then appended to the currently empty DataFrame “sampled_dfs”.

Finally, I have combined all the sampled DataFrames by using pd.concat() to concatenate all the sampled DataFrames into one big DataFrame. The “ignore_index=True” removes the current index such that it will now run sequentially from 0.

```
# Store the df in csv/parquet
df.to_csv('yellow_taxi_trip.csv')
```

We see that there are a total of 1915511 rows present in the combined DataFrame and I have converted it to a CSV file for easy use in upcoming steps.

2. Data Cleaning

2.1. Fixing Columns

2.1.1. Fix the index

After loading the CSV file as a new DataFrame (df1) and using both the *.head()* and *.info()* methods to check the columns, we see that there is an ‘Unnamed:0’ column in the new DataFrame.

```
df1.head()
```

	Unnamed: 0	VendorID	tpep_pickup_datetime	tpep_dropoff_datetime	passenger_count	trip_distance	RatecodeID
0	0	2	2023-01-22 00:18:34	2023-01-22 00:40:57	5.0	4.70	1.0
1	1	2	2023-01-03 00:20:28	2023-01-03 00:26:08	1.0	1.30	1.0
2	2	1	2023-01-16 00:25:28	2023-01-16 00:33:30	2.0	2.20	1.0
3	3	2	2023-01-01 00:44:58	2023-01-01 00:51:32	1.0	1.53	1.0
4	4	1	2023-01-28 00:30:32	2023-01-28 00:55:03	1.0	3.90	1.0

5 rows × 23 columns

As the newly added column will affect the entire process of EDA, I removed it by using the *.drop()* method. I have observed that the number of columns have dropped from 21 to 20, indicating that the column has

been removed. Refer to the output below.

```
# Fix the index and drop any columns that are not needed
df1 = df1.drop(columns=['Unnamed: 0'])
df1.head()
```

	VendorID	tpep_pickup_datetime	tpep_dropoff_datetime	passenger_count	trip_distance	RatecodeID
0	2	2023-01-01 00:10:30	2023-01-01 00:11:49	1.0	0.49	1.0
1	2	2023-01-01 00:49:02	2023-01-01 00:55:15	1.0	0.75	1.0
2	1	2023-01-01 00:47:17	2023-01-01 01:07:01	2.0	2.90	1.0
3	2	2023-01-01 00:06:02	2023-01-01 00:31:38	1.0	2.50	1.0
4	2	2023-01-01 00:02:19	2023-01-01 00:30:49	1.0	20.37	2.0

2.1.2. Combine the two airport_fee columns

From the initial analysis, there are two airport fee columns in the dataset. I have merged both of them and filled in the values such that only one 'airport_fee' column remains.

I did this using the *.fillna()* method to fill values from the 'Airport_fee' column into the 'airport_fee' column.

After this, I have dropped the 'Airport_fee' column as the values in it are now the same as the values in the other column. Overall, it will result in the number of columns present to drop to 19.

```
# Combine the two airport fee columns
df1['airport_fee'] = df1['airport_fee'].fillna(df1['Airport_fee'])
df1 = df1.drop(columns=['Airport_fee'])
df1.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1915511 entries, 0 to 1915510
Data columns (total 19 columns):
 #   Column                 Dtype
---  ------                 -----
 0   VendorID               int64
 1   tpep_pickup_datetime   object
 2   tpep_dropoff_datetime  object
 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: float64(12), int64(4), object(3)
memory usage: 277.7+ MB
```

2.1.3. Remove negative values in RatecodeID column

Firstly, I checked for all the negative values in the fare_amount column. Using Boolean filtering, I found out where the negative values are with fare_amount being less than 0 (fare_amount < 0)

```
# check where values of fare amount are negative
print("Rows with negative 'fare_amount':")
display(df1[df1['fare_amount'] < 0].head())
```

```
Rows with negative 'fare_amount':
```

	VendorID	tpep_pickup_datetime	tpep_dropoff_datetime	passenger_count	trip_distance	RatecodeID
115	2	2023-01-07 00:33:41	2023-01-07 00:42:56	1.0	1.64	1.0
145	2	2023-01-29 00:50:07	2023-01-29 00:53:54	1.0	0.33	1.0
196	2	2023-01-22 00:06:46	2023-01-22 00:06:53	1.0	0.00	2.0
264	2	2023-01-29 00:43:38	2023-01-29 00:44:38	1.0	0.01	1.0
274	2	2023-01-30 00:48:41	2023-01-30 01:00:19	1.0	1.97	1.0

5 rows × 21 columns

Next, I have analyzed the RateCodeID column using Boolean filtering to check for negative amounts, and then the *.value_counts()* method to see how many times a certain value occurs. As shown, we can see that NaN has appeared 331 times in the column.

```
# Analyse RatecodeID for the negative fare amounts
print("\nValue counts for 'RatecodeID' in rows with negative 'fare_amount':")
print(df1[df1['fare_amount'] < 0]['RatecodeID'].value_counts(dropna=False))
```

```
Value counts for 'RatecodeID' in rows with negative 'fare_amount':
RatecodeID
1.0    16784
2.0     1321
NaN      331
5.0      318
3.0      235
4.0       90
Name: count, dtype: int64
```

Thirdly, I have checked which other columns have negative values in them. *"df1.select_dtypes(include=np.number)"* selects only the numeric columns in the dataframe and *".columns"* gets the list of numeric column names.

The columns which contain negative values after this operation are - "fare_amount", "extra", "mta_tax", "tip_amount", "tolls_amount", "improvement_surcharge", "total_amount", "congestion_surcharge", "airport_fee".

```
# Find which columns have negative values
negative_value_cols = []
for col in df1.select_dtypes(include=np.number).columns:
    if (df1[col] < 0).any():
        negative_value_cols.append(col)
print("Columns with negative values:", negative_value_cols)
```

Finally, I fixed the values in these columns by using the function below to replace with 0 such that no negative values remain. The lambda function in the code follows this logic:

- If “x” is greater than or equal to 0, leave it as it is
- If “x” is less than 0, replace it with 0

The cleaned values are then assigned back to the same column. The code *“(df1[col] < 0).sum()”* rechecks each cleaned column and counts how many negative values remain, which is 0.

```
# fix these negative values
for col in negative_value_cols:
    df1[col] = df1[col].apply(lambda x: x if x >= 0 else 0)

print("Negative values after fixing:")
for col in negative_value_cols:
    print(f"{col}: {(df1[col] < 0).sum()}")
```

```
Negative values after fixing:
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
```

2.2. Handling Missing Values

2.2.1. Find the proportion of missing values in each column

Firstly, I found the number of missing values present using the *.isnull().sum()* method. Then, to find the proportion of missing values in each column, the formula used was: *Proportion = Number of values/Total number of rows*. Based on that, we can see in the screenshot below that there are some rows - *“passenger_count”*, *“RateCodeID”*, *“store_and_fwd_flag”* with missing values with a proportion of 0.033906, which is not much in the fully merged dataset.

```
# Find the proportion of missing values in each column
missing_values = df1.isnull().sum()
total_rows = len(df1)
missing_proportion = missing_values / total_rows
print("Proportion of missing values in each column:")
print(missing_proportion)
```

```
Proportion of missing values in each column:
VendorID                 0.000000
tpep_pickup_datetime     0.000000
tpep_dropoff_datetime    0.000000
passenger_count          0.033906
trip_distance            0.000000
RatecodeID               0.033906
store_and_fwd_flag       0.033906
PULocationID             0.000000
DOLocationID             0.000000
payment_type             0.000000
fare_amount              0.000000
extra                    0.000000
mta_tax                  0.000000
tip_amount               0.000000
tolls_amount             0.000000
improvement_surcharge    0.000000
total_amount             0.000000
congestion_surcharge     0.000000
airport_fee              0.000000
pickup_hour              0.000000
pickup_month             0.000000
dtype: float64
```

2.2.2. Handling missing values in passenger_count

```
# Display the rows with null values
# Impute NaN values in 'passenger_count'
print("Rows with null values in 'passenger_count':")
display(df1[df1['passenger_count'].isnull()].head())
print("\nValue counts for 'passenger_count':")
print(df1['passenger_count'].value_counts(dropna=False))
mode_passenger_count = df1['passenger_count'].mode()[0]
df1['passenger_count'] = df1['passenger_count'].fillna(mode_passenger_count)
print("\nMissing values in 'passenger_count' after imputation:")
print(df1['passenger_count'].isnull().sum())
print("\nValue counts for 'passenger_count' AFTER IMPUTATION:")
print(df1['passenger_count'].value_counts(dropna=False))
```

Based on the code shown above, I displayed all the rows with null values in *"passenger_count"*.

Then, I checked the value counts for the column, which showed NaN values being at a sum of 65962.

```
Value counts for 'passenger_count':
passenger_count
1.0    1391536
2.0     280696
3.0      69652
NaN      65140
4.0      39250
0.0      29587
5.0      23720
6.0      15917
8.0         10
9.0          2
7.0          1
Name: count, dtype: int64
```

Based on that, I have imputed the NaN values in the column. As the value

count shows a majority of the value 1, I used the mode and filled in the NaN rows. After this, the number of missing values present are 0.

```
Missing values in 'passenger_count' after imputation:
0
```

After imputation, we can see that there are only zero values left in the “passenger_count” column.

```
Value counts for 'passenger_count' AFTER IMPUTATION
passenger_count
1.0    1456676
2.0     280696
3.0      69652
4.0      39250
0.0      29587
5.0      23720
6.0      15917
8.0         10
9.0          2
7.0          1
Name: count, dtype: int64
```

Now, we handle the zero values present in the column by using another Boolean filter to only save values of “passenger_count” being more than 0.

I have also included the number of rows left in the dataframe after removing zero passenger counts, which is now at 1886805.

The outputs are arranged in a descending order of value counts from most to least.

```
# Handle zero values in 'passenger_count'
# Remove rows where passenger_count is 0 as it's not a valid taxi trip
df1 = df1[df1['passenger_count'] > 0].copy()

print("Number of rows after removing zero passenger counts:", len(df1))
print("\nValue counts for 'passenger_count' after removing zeros:")
print(df1['passenger_count'].value_counts(dropna=False))
```

```
Number of rows after removing zero passenger counts: 1886805

Value counts for 'passenger_count' after removing zeros:
passenger_count
1.0    1456340
2.0     280743
3.0      69733
4.0      39769
5.0      24272
6.0      15931
8.0         11
9.0          4
7.0          2
Name: count, dtype: int64
```

2.2.3. Handle missing values in RatecodeID

```
# Fix missing values in 'RatecodeID'
print("Missing values in 'RatecodeID' before handling:", df1['RatecodeID'].isnull().sum())
print("\nValue counts for 'RatecodeID':")
print(df1['RatecodeID'].value_counts(dropna=False))
mode_ratecode_id = df1['RatecodeID'].mode()[0]
df1['RatecodeID'] = df1['RatecodeID'].fillna(mode_ratecode_id)
print("\nMissing values in 'RatecodeID' after imputation:")
print(df1['RatecodeID'].isnull().sum())
```

Based on the code above, I initially found the number of missing values present in the RateCodeID column, which are 65140.

Hence after the process of imputation by mode, there are no more missing values in the RateCodeID column.

```
Missing values in 'RatecodeID' before handling: 65140

Value counts for 'RatecodeID':
RatecodeID
1.0     1717406
2.0       72252
NaN       65140
5.0       10697
99.0      10351
3.0        6295
4.0        3779
6.0           4
Name: count, dtype: int64

Missing values in 'RatecodeID' after imputation:
0
```

2.2.4. Impute NaN in congestion_surcharge

```
# handle null values in congestion_surcharge
print("Missing values in 'congestion_surcharge' before handling:", df1['congestion_surcharge'].isnull().sum())
print("\nValue counts for 'congestion_surcharge':")
print(df1['congestion_surcharge'].value_counts(dropna=False))
df1['congestion_surcharge'] = df1['congestion_surcharge'].fillna(0)
print("\nMissing values in 'congestion_surcharge' after imputation:")
print(df1['congestion_surcharge'].isnull().sum())
```

As shown in the code above, I firstly checked if there were missing values in “congestion_surcharge”, which shows that there are no missing values. I have also checked the distribution of the values in the column to decide on the imputation strategy. The most frequent values are 2.5, 0.0 and 1.0 with a value count of 1662457, 223465 and 2, respectively.

An ***assumption*** made was that the missing values might correspond to the trips where surcharge was not applied. Hence with that assumption, I imputed the missing values with 0.

After imputation, we see that the missing values in the column are 0. Hence, we can say it was not necessary for imputation to occur since there were no missing values in the starting of the process. The outputs for the code are below.

```
Missing values in 'congestion_surcharge' before handling: 0

Value counts for 'congestion_surcharge':
congestion_surcharge
2.5    1662457
0.0     223465
1.0          2
Name: count, dtype: int64

Missing values in 'congestion_surcharge' after imputation:
0
```

Questions

a) Are there missing values in other columns?
Answer: Yes there are missing values in the other columns, namely, “store_and_fwd_flag”.

b) Did you find NaN values in some other set of columns?
Answer: Yes there are NaN values in other sets of columns.

After seeing that there are missing values in other columns, I have handled them using the code below

```
# Handle any remaining missing values
print("Missing values in each column after previous steps:")
print(df1.isnull().sum())
if 'store_and_fwd_flag' in df1.columns and df1['store_and_fwd_flag'].isnull().sum() > 0:
    mode_store_and_fwd_flag = df1['store_and_fwd_flag'].mode()[0]
    df1['store_and_fwd_flag'] = df1['store_and_fwd_flag'].fillna(mode_store_and_fwd_flag)
    print("\nMissing values in 'store_and_fwd_flag' after imputation:")
    print(df1['store_and_fwd_flag'].isnull().sum())
critical_cols_with_missing = ['tpep_dropoff_datetime', 'trip_distance', 'PULocationID', 'DOLocationID', 'payment_type']
initial_rows = len(df1)
df1.dropna(subset=critical_cols_with_missing, inplace=True)
rows_removed = initial_rows - len(df1)
print(f"\nRemoved {rows_removed} row(s) with missing values in critical columns.")

print("\nMissing values in each column after handling remaining NaNs:")
print(df1.isnull().sum())
```

Firstly, I checked for missing values that are present in each column using *.isnull().sum()* and the output (below) shows that only the “store_and_fwd_flag” column has 64948 missing values, while other columns do not have any.
Next, I imputed the values by calculating the mode (highest frequency of values) and filled in the missing value with this value, as this strategy is suitable for categorical variables.
Lastly, I have dropped rows of missing data in important trip columns such as “tpep_dropoff_datetime”, “trip_distance”, “PULocationID”, “DOLocationID”, “payment_type” so that the dataset will contain valid values that can be further used during EDA.

Missing values before handling NaN was removed

Missing values after handling NaNs

VendorID	0
tpep_pickup_datetime	0
tpep_dropoff_datetime	0
passenger_count	0
trip_distance	0
RatecodeID	0
store_and_fwd_flag	65140
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
dtype: int64	

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
dtype: int64	

2.3. Handling Outliers and Standardising Values

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

Firstly, I transformed the dataframe so that the rows will become columns, which makes it easier to read and understand. I also used the *.describe()* method so that I can get the descriptive statistics of each column in the dataframe.

```
# Describe the data and check if there are any potential outliers present
# Check for potential out of place values in various columns
display(df1.describe().T)
```

	count	mean	std	min	25%	50%	75%	max
VendorID	1885924.0	1.747144	0.439719	1.0	1.00	2.00	2.00	6.00
passenger_count	1885924.0	1.377697	0.868219	1.0	1.00	1.00	1.00	9.00
trip_distance	1885924.0	4.240342	266.372250	0.0	1.04	1.79	3.40	159017.60
RatecodeID	1885924.0	1.611576	7.245592	1.0	1.00	1.00	1.00	99.00
PULocationID	1885924.0	165.194961	64.023197	1.0	132.00	162.00	234.00	265.00
DOLocationID	1885924.0	163.894141	69.875249	1.0	113.00	162.00	234.00	265.00
payment_type	1885924.0	1.184269	0.557311	0.0	1.00	1.00	1.00	4.00
fare_amount	1885924.0	19.689209	18.353558	0.0	9.30	13.50	21.90	904.60
extra	1885924.0	1.545171	1.816939	0.0	0.00	1.00	2.50	20.80
mta_tax	1885924.0	0.490314	0.069290	0.0	0.50	0.50	0.50	5.75
tip_amount	1885924.0	3.520767	4.053737	0.0	1.00	2.80	4.41	411.10
tolls_amount	1885924.0	0.592340	2.178554	0.0	0.00	0.00	0.00	104.75
improvement_surcharge	1885924.0	0.989116	0.103098	0.0	1.00	1.00	1.00	1.00
total_amount	1885924.0	28.672360	22.969402	0.0	15.95	21.00	30.72	906.10
congestion_surcharge	1885924.0	2.203771	0.807973	0.0	2.50	2.50	2.50	2.50
airport_fee	1885924.0	0.137318	0.456464	0.0	0.00	0.00	0.00	1.75

Based on this, we can see that there are some outliers present in the dataset. These can be dropped/removed by dropping the rows with these types of values.

Firstly, we remove the passenger counts of 7+ as there are very few instances of them, compared to the other passenger counts. Now, some of the outliers have been removed.

```
# remove passenger_count > 6
df1 = df1[df1['passenger_count'] <= 6].copy()
print("Number of rows after removing passenger counts > 6:", len(df1))
print("\nValue counts for 'passenger_count' after removing > 6:")
print(df1['passenger_count'].value_counts(dropna=False))
```

```
Number of rows after removing passenger counts > 6: 1885911

Value counts for 'passenger_count' after removing > 6:
passenger_count
1.0    1456676
2.0     280696
3.0      69652
4.0      39250
5.0      23720
6.0      15917
Name: count, dtype: int64
```

Next, I removed rows of data where "*payment_type* = 0" as they are not considered as a valid value. Before removing, there are 1820771 rows where payment_type is 0. After removing, there are no rows with

“*payment_type* = 0”.

```
# Fix trips where payment_type is 0
df1 = df1[df1['payment_type'] != 0].copy()

print("Number of rows after removing payment_type 0:", len(df1))
print("\nValue counts for 'payment_type' after removing 0:")
print(df1['payment_type'].value_counts(dropna=False))
```

```
Number of rows after removing payment_type 0: 1820771

Value counts for 'payment_type' after removing 0:
payment_type
1    1469280
2     315162
4      24835
3      11494
Name: count, dtype: int64
```

Thirdly, I handled rows with both distance = 0 and fare = 0 but pickup and dropoff zones are different in the dataset. The *.copy()* method is used to create a shallow copy of an object such that the content is the same but the copied object does not share the same memory address. As shown below, there were 157 rows that were removed.

```
# Handle trips with 0 distance and 0 fare but different zones
initial_rows = len(df1)
df1 = df1[~((df1['trip_distance'] == 0) & (df1['fare_amount'] == 0) &
 (df1['PULocationID'] != df1['DOLocationID']))].copy()
rows_removed = initial_rows - len(df1)
print(f"""Removed {rows_removed} rows where trip_distance and fare_amount were
0 but pickup and dropoff zones were different.""")
```

```
Removed 157 rows where trip_distance and fare_amount were
0 but pickup and dropoff zones were different.
```

Finally, I handled rows with trip distance being more than 250 miles, shown below. I was able to remove 17 rows as such.

```
# Handle entries where trip_distance is more than 250 miles
initial_rows = len(df1)
df1 = df1[df1['trip_distance'] <= 250].copy()
rows_removed = initial_rows - len(df1)
print(f"Removed {rows_removed} rows where trip_distance was more than 250 miles.")
```

```
Removed 17 rows where trip_distance was more than 250 miles.
```

Question: Do any of the columns need standardising?

Answer: No, none of the columns need any standardising, as shown below, as they are all datatypes that can be used easily.

```
VendorID                 int64
tpep_pickup_datetime    object
tpep_dropoff_datetime   object
passenger_count        float64
trip_distance          float64
RatecodeID             float64
store_and_fwd_flag      object
PULocationID             int64
DOLocationID             int64
payment_type             int64
fare_amount            float64
extra                  float64
mta_tax                float64
tip_amount             float64
tolls_amount           float64
improvement_surcharge  float64
total_amount           float64
congestion_surcharge   float64
airport_fee            float64
dtype: object
```

3. Exploratory Data Analysis

3.1. General EDA: Finding Patterns and Trends

Print the columns present in the DataFrame as a list.

```
df1.columns.tolist()
```

```
['VendorID',
 'tpep_pickup_datetime',
 'tpep_dropoff_datetime',
 'passenger_count',
 'trip_distance',
 'RatecodeID',
 'store_and_fwd_flag',
 'PULocationID',
 'DOLocationID',
 'payment_type',
 'fare_amount',
 'extra',
 'mta_tax',
 'tip_amount',
 'tolls_amount',
 'improvement_surcharge',
 'total_amount',
 'congestion_surcharge',
 'airport_fee',
 'pickup_hour',
 'pickup_month']
```

3.1.1. Classify variables into categorical and numerical

Variable	Numerical/Categorical
VendorID	Categorical
tpep_pickup_datetime	Numerical (Temporal)
tpep_dropoff_datetime	Numerical (Temporal)
passenger_count	Numerical (Discrete)

trip_distance	Numerical (Continuous)
RatecodeID	Categorical
PULocationID	Categorical
DOLocationID	Categorical
payment_type	Categorical
pickup_hour	Numerical (Discrete)
trip_duration	Numerical (Continuous)

Question: The following monetary parameters belong in the same category, is it categorical or numerical?

fare_amount	Answer: Numerical
extra	
mta_tax	
tip_amount	
tolls_amount	
improvement_surcharge	
total_amount	
congestion_surcharge	
airport_fee	

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

1 - Find & show the hourly trends in taxi pickups

```
# Find and show the hourly trends in taxi pickups
df1['tpep_pickup_datetime'] = pd.to_datetime(df1['tpep_pickup_datetime'])
df1['pickup_hour'] = df1['tpep_pickup_datetime'].dt.hour
hourly_pickups = df1['pickup_hour'].value_counts().sort_index()
print("hourly_pickups"+str(hourly_pickups.value_counts))
plt.figure(figsize=(10, 6))
hourly_pickups.plot(kind='bar')
plt.title('Hourly Taxi Pickups')
plt.xlabel('Hour of Day')
plt.ylabel('Number of Pickups')
plt.xticks(rotation=0)
plt.grid(axis='y')
plt.show()
```

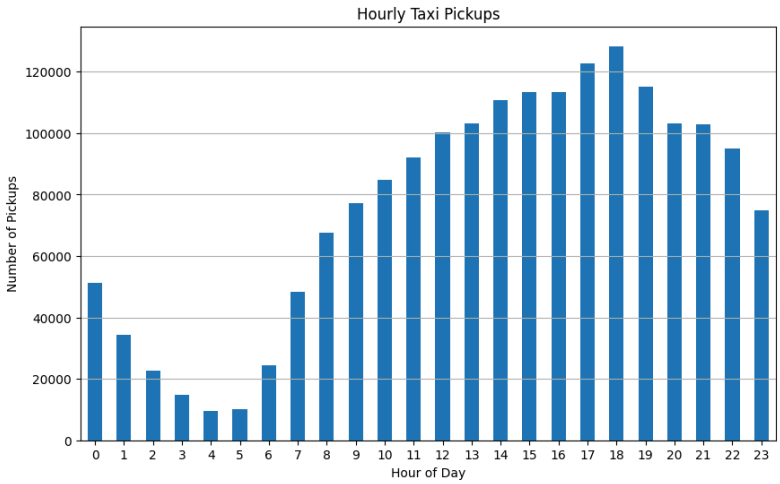
The code above is used to analyse hourly trends in taxi pickups. The steps taken are mentioned below.

1) By converting the “tpep_pickup_datetime” column into datetime format, the values are in that format for easy manipulation for the next steps.
2) Extract hour from “tpep_pickup_datetime” using the *.dt.hour* attribute and save it to a new column called “pickup_hour”. The new column will contain the extracted hours in the range of 0 - 23.
3) Count the number of pickups by the hours using the *value_counts()* method. The *sort_index()* method is used to sort the DataFrame by its index. Based on this, we see the output below arranged from hour 0 to hour 23.

```
hourly_pickups<bound method IndexOpsMixin.value_counts of pickup_hour
0      51357
1      34371
2      22686
3      14830
4       9575
5      10089
6      24346
7      48458
8      67558
9      77256
10     84870
11     92234
12    100138
13    103169
14    110699
15    113376
16    113455
17    122707
18    128339
19    115216
20    103110
21    102865
22     94987
23     74906
Name: count, dtype: int64>
```

4) Display the values above as a bar chart, shown below.

The output of the code shows the distribution of hourly taxi pickups. We can see that the data is mostly skewed to the right and the highest number of hourly pickups is 128339 at hour 18. This shows that most of the pickups happen later in the day, with a few happening at the start of the day.



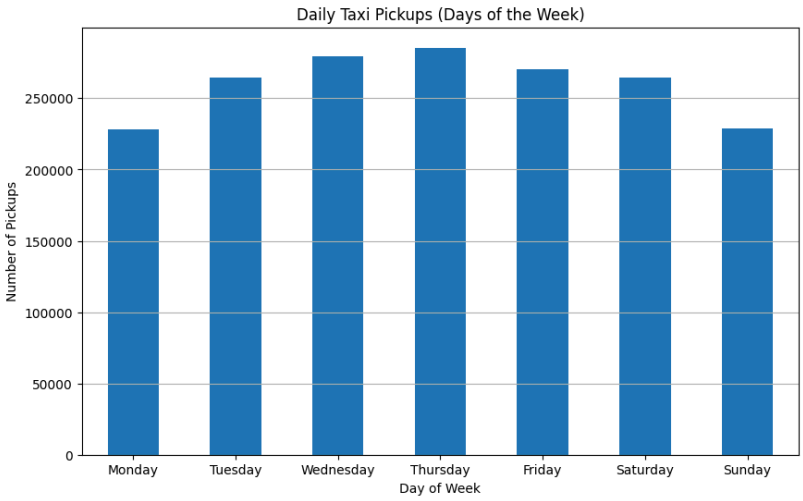
2 - Find & show the daily trends in taxi pickups (days of the week)

```
# Find and show the daily trends in taxi pickups (days of the week)
df1['pickup_day_of_week'] = df1['tpep_pickup_datetime'].dt.dayofweek
daily_pickups = df1['pickup_day_of_week'].value_counts().sort_index()
day_names = {0: 'Monday', 1: 'Tuesday', 2: 'Wednesday', 3: 'Thursday', 4: 'Friday', 5: 'Saturday', 6: 'Sunday'}
daily_pickups.index = daily_pickups.index.map(day_names)
plt.figure(figsize=(10, 6))
daily_pickups.plot(kind='bar')
plt.title('Daily Taxi Pickups (Days of the Week)')
plt.xlabel('Day of Week')
plt.ylabel('Number of Pickups')
plt.xticks(rotation=0)
plt.grid(axis='y')
plt.show()
```

The code above is used to show daily trends in taxi pickups, based on the days of the week. The steps taken are mentioned below:

1) Extract the day of the week from the “tpep_pickup_datetime” column using the *.dt.dayofweek* attribute.
2) Count the number of taxi pickups that happened on each day of the week.
3) Create a dictionary object with the key being the numeric code corresponding to the weekday and value being the day of the week. This is done to make it easier to read.
4) Create an index for the days of the week by using the *.map()* method to map the numeric index to the corresponding days in the dictionary. This helps to ensure that the bar chart will show the days instead of their equivalent numeric codes.
5) Display the values in the bar chart as shown below.

Based on the output bar chart below, we can infer that the taxi demand gradually rises from Monday, peaks midweek on Wednesday and Thursday, and gradually dips for the remainder of the week (Friday, Saturday, Sunday).



3 - Show the monthly trends in pickups

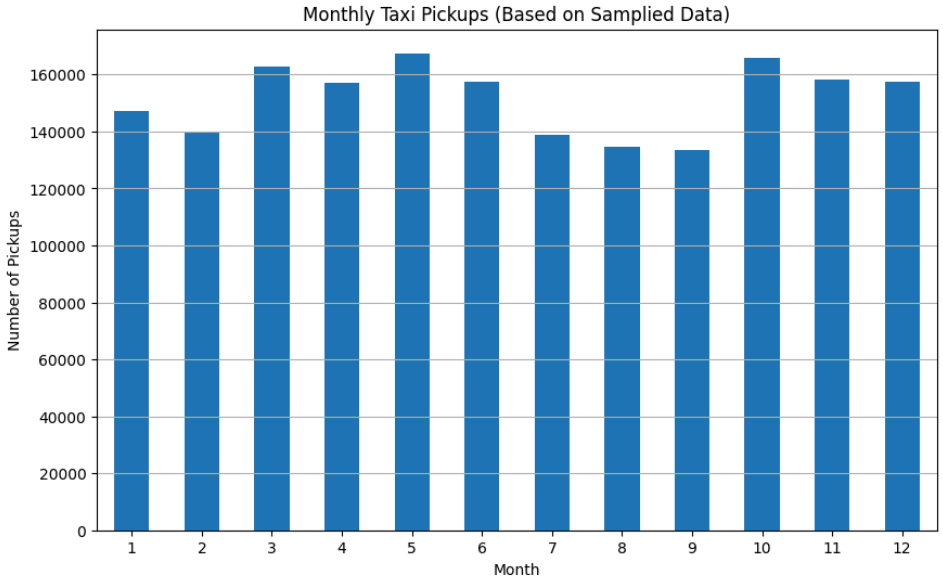
```
# Show the monthly trends in pickups
df1['pickup_month'] = df1['tpep_pickup_datetime'].dt.month
monthly_pickups = df1['pickup_month'].value_counts().sort_index()
all_months = pd.DataFrame({'pickup_month': range(1, 13)})
monthly_pickups = pd.merge(all_months, monthly_pickups, on='pickup_month', how='left').fillna(0)
monthly_pickups = monthly_pickups.set_index('pickup_month')['count']

plt.figure(figsize=(10, 6))
monthly_pickups.plot(kind='bar')
plt.title('Monthly Taxi Pickups (Based on Samplied Data)')
plt.xlabel('Month')
plt.ylabel('Number of Pickups')
plt.xticks(rotation=0)
plt.grid(axis='y')
plt.show()
```

The code above is used to show monthly trends in taxi pickups. The steps taken are mentioned below:

1) Extract the month from the “tpep_pickup_datetime” column using the *.dt.month* attribute. This creates a new column “pickup_month” which contains the numeric month values, from 1 to 12, with 1 being January, for each ride.
2) Count the number of pick ups per month using the *.value_counts()* method. This gives the number of rides that happened each month.
3) Create a dataframe for all 12 months using the *range()* function from 1 to 13 - with 1 being included and 13 being excluded. This ensures that months without data, if present, are included in the chart.
4) Merge the “monthly_pickups” data with “all_pickups” to fill any missing months with 0, using the *.fillna(0)* method.
5) Set “pickup_month” as the index for easy plotting of a visualisation
6) Plot the bar chart using *.plot()* and display it using *plt.show()*

From the output below, we can infer that the number of taxi pickups have been relatively consistent, except for sudden dips between July and September. The peak months can be interpreted as May, June and October, which show the highest number of pickups in the year.



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

Question: Take a look at the financial parameters like fare_amount, tip_amount, total_amount, and also trip_distance. Do these contain zero/negative values?

Answer: Yes they contain zero values, based on my analysis of the data as shown below.

```
# Analyse the above parameters
financial_cols = ['fare_amount', 'tip_amount', 'total_amount', 'trip_distance']

print("Number of zero values in key financial columns and trip distance:")
for col in financial_cols:
    zero_count = (df1[col] == 0).sum()
    print(f"{col}: {zero_count}")
```

```
Number of zero values in key financial columns and trip distance:
fare_amount: 19186
tip_amount: 420575
total_amount: 18980
trip_distance: 24264
```

Question: Do you think it is beneficial to create a copy of the DataFrame leaving out the zero values from these?
Answer: Yes as it removes invalid or missing data values. It helps ensure more accurate analysis and does not change any trends.

Hence, I have removed the zero values from the specified columns as shown below.

Firstly, I created a list of the columns where they contain zeros. Then, I created a Boolean DataFrame showing True where values are not zero (df1[financial_cols_and_distance] != 0). The *.all(axis=1)* function checks each row and returns True only when all selected columns in that zero are not zero. The *.copy()* function creates a new independent DataFrame so as to not modify the original DataFrame.

The original number of rows was 1820597 and after removing the rows with zero values, the new number of rows is now 1391636.

```
# Create a df with non zero entries for the selected parameters.
financial_cols_and_distance = ['fare_amount', 'tip_amount', 'total_amount', 'trip_distance']
df1_filtered = df1[(df1[financial_cols_and_distance] != 0).all(axis=1)].copy()

print(f"Original number of rows: {len(df1)}")
print(f"Number of rows after removing zero values: {len(df1_filtered)}")
```

```
Original number of rows: 1820597
Number of rows after removing zero values: 1391636
```

Question: The distance might be 0 in cases where pickup and drop is in the same zone. Do you think it is suitable to drop such cases of zero distance?

Answer: No as some rides may genuinely have started and ended in the same zone. These cases should be handled separately to see if they can represent valid rides or data errors.

3.1.4. Analyse the monthly revenue trends

The code below analyses the monthly revenue trends by grouping data by its month. The steps followed are mentioned below:

1) Extract the numeric value for month by using .dt.month for each trip. Then store it into a new column called “pickup_month”.
2) Group all the taxi records by their pickup month using the *.groupby()* method.
3) Calculate the sum of the values in the “total_amount” column to get the total revenue for each month (within the group as done in the previous step). The summation can be done by using the *.sum()* method.
4) Print the monthly revenue where the index is the numeric value of the month.

```
# Group data by month and analyse monthly revenue
df1_filtered['pickup_month'] = df1_filtered['tpep_pickup_datetime'].dt.month
monthly_revenue = df1_filtered.groupby('pickup_month')['total_amount'].sum()
print("Monthly Revenue:")
print(monthly_revenue)
```

```
Monthly Revenue:
pickup_month
1     3175817.62
2     3033279.65
3     3636879.68
4     3527199.92
5     3863039.74
6     3621873.68
7     3068441.26
8     2961840.57
9     3155540.24
10    3922610.91
11    3676866.59
12    3581192.21
Name: total_amount, dtype: float64
```

From the above output, we can infer that the month of May (month number 5) has the highest revenue of 3881024.29 and August (month number 8) has the lowest revenue of 2975118.97. The revenue fluctuates monthly which indicates the seasonal variation in taxi demand, Overall, the taxi services remain consistent based on the amount of revenue all year.

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

To calculate the proportion of each quarter's revenue, I have taken the following steps as shown in the code:

1) Create a dictionary, "quarter_map", and map the months to their respective quarters (each quarter has 3 months), using the *.map()* function. A dictionary object is easier for readability and it is more efficient for lookup compared to if-elif-else statements.
2) Create a column with quarter values based on the pickup month by applying the *.map()* function.
3) Calculate total revenue for each quarter through grouping by "pickup_quarter" and the total revenue is summed for each quarter.
4) Calculate overall annual revenue by summing all the quarters.
5) Calculate the portion of revenue contributed by each quarter - dividing the total revenue by each quarter's revenue.

```
# Calculate proportion of each quarter
quarter_map = {
    1: 'Q1', 2: 'Q1', 3: 'Q1',
    4: 'Q2', 5: 'Q2', 6: 'Q2',
    7: 'Q3', 8: 'Q3', 9: 'Q3',
    10: 'Q4', 11: 'Q4', 12: 'Q4'
}
df1_filtered['pickup_quarter'] = df1_filtered['tpep_pickup_datetime'].dt.month.map(quarter_map)
quarterly_revenue = df1_filtered.groupby('pickup_quarter')['total_amount'].sum().reindex(['Q1', 'Q2', 'Q3', 'Q4'])
total_revenue = quarterly_revenue.sum()
quarterly_revenue_proportion = quarterly_revenue / total_revenue
print("\nQuarterly Revenue Proportion:")
print(quarterly_revenue_proportion)
```

```
Quarterly Revenue Proportion:
pickup_quarter
Q1    0.238838
Q2    0.267125
Q3    0.222824
Q4    0.271214
Name: total_amount, dtype: float64
```

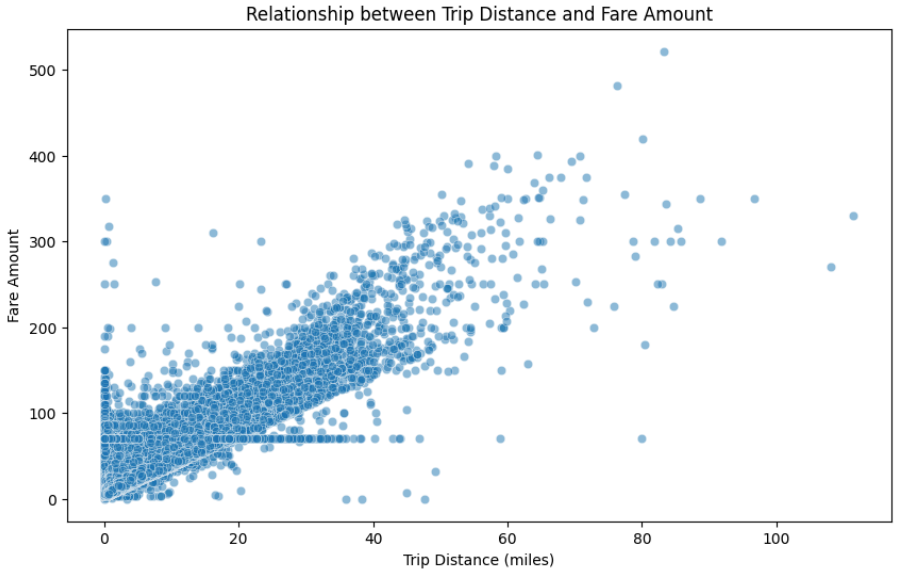
The output of the code shows the proportion of each quarter. Quarter 4, which has months October to December, has the highest proportion of revenue at 0.271214. This could be as that quarter is close to the end of the year, which has resulted in more travels happening.

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

The code below shows how the trip fare is affected by the distance. I have done the following steps to produce the output:

1) I have selected rows from “df1_filtered” where the distance column is greater than 0. Then, I have copied the results into a new DataFrame object to prevent any modifications to the original DataFrame. The purpose of this step is to remove zero distances that could possibly change the correlation and scatter plot.
2) I have then created a scatterplot using the Seaborn library with the x-axis having the trip_distance and the y-axis having the fare_amount. Here, *alpha=0.5* sets the transparency of the points to 50% such that any overlapping points will be visible.
3) I have computed the correlation between trip_distance and fare_amount using Pearson’s correlation (default), with the method *.corr()* being used here. The range of the output of this code lies between -1 and 1, where:
   -1 means that the correlation between the two columns is strongly negative,
   1 means that the correlation between the two columns is strongly positive, and
   0 means that there is little to no linear relationship between the two columns.
4) Finally, I printed the correlation result, rounded off to 2 decimal places as it is easier to read and understand.

```
# Show how trip fare is affected by distance
df_filtered_distance = df1_filtered[df1_filtered['trip_distance'] > 0].copy()
plt.figure(figsize=(10, 6))
sns.scatterplot(data=df_filtered_distance, x='trip_distance', y='fare_amount', alpha=0.5)
plt.title('Relationship between Trip Distance and Fare Amount')
plt.xlabel('Trip Distance (miles)')
plt.ylabel('Fare Amount')
plt.show()
correlation = df_filtered_distance['trip_distance'].corr(df_filtered_distance['fare_amount'])
print(f"\nCorrelation between trip_distance and fare_amount: {correlation:.2f}")
```




```
Correlation between trip_distance and fare_amount: 0.95
```

Based on the outputs above, we see that the correlation between the two columns are strongly positive (r=0.95). This means that as the trip distance increases, the fare amount also increases proportionally. The scatterplot shows a clear trend going upwards which suggests most of the fare_amount is consistent with the distance travelled.

However, there are a few outliers which could suggest incorrect data being noted or long distances travelled but having fixed fare amounts.

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

1 - Show relationship between fare and trip duration

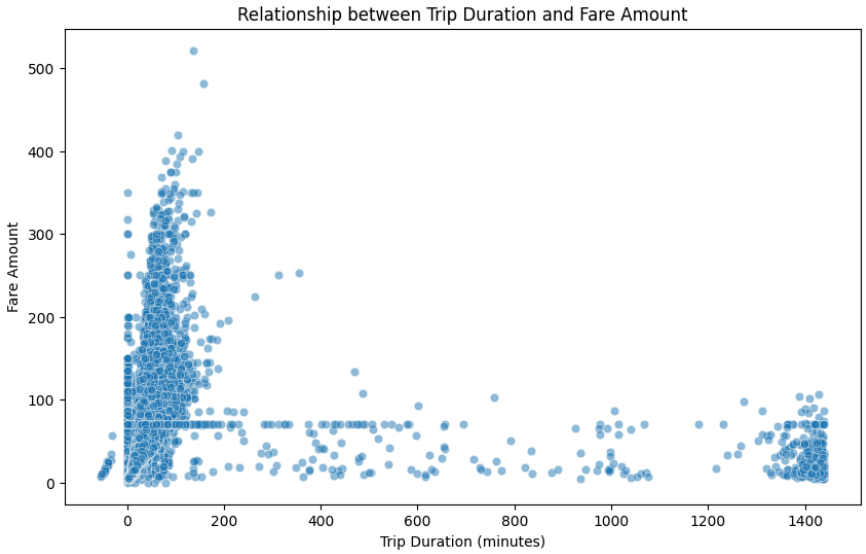
The code below shows the relationship between fare_amount and trip_duration. I have followed the steps below to produce the desired outputs.

1) I have converted the “tpep_pickup_datetime” column to datetime datatype so that time-based operations can be done easily, using *pd.to_datetime()*. Similarly, I have done the same for

"tpep_dropoff_datetime". Both of these columns are important for analysis as they provide the necessary data for calculations being done in further steps.

2) To get the duration of the trip (which is the difference in time between pickup and dropoff), I have calculated it by subtracting the two columns mentioned in the previous step. Then, I have converted the result into total seconds and then into minutes using *.dt.total_seconds()/60*. The reason for this operation was that after subtraction, the output datatype of the result will be a pandas Timedelta object (timedelat64[ns]) which is difficult to interpret. The entire result is stored in a new column called "trip_duration".
3) I have plotted the correlation using Seaborn's scatterplot with x-axis being trip_duration and y-axis being fare_amount. Alpha amount is set to 0.5 for 50% transparency on overlapping points.
4) Finally, I calculated the correlation between the two columns using Pearson's correlation as a default in the *.corr()* function and have displayed it, rounding up to 2 decimal points.

```
# Show relationship between fare and trip duration
df1_filtered['tpep_pickup_datetime'] = pd.to_datetime(df1_filtered['tpep_pickup_datetime'])
df1_filtered['tpep_dropoff_datetime'] = pd.to_datetime(df1_filtered['tpep_dropoff_datetime'])
df1_filtered['trip_duration'] = (df1_filtered['tpep_dropoff_datetime'] - df1_filtered['tpep_pickup_datetime']).dt.total_seconds() / 60
plt.figure(figsize=(10, 6))
sns.scatterplot(data=df1_filtered, x='trip_duration', y='fare_amount', alpha=0.5)
plt.title('Relationship between Trip Duration and Fare Amount')
plt.xlabel('Trip Duration (minutes)')
plt.ylabel('Fare Amount')
plt.show()
correlation_duration_fare = df1_filtered['trip_duration'].corr(df1_filtered['fare_amount'])
print(f"\nCorrelation between trip_duration and fare_amount: {correlation_duration_fare:.2f}")
```




```
Correlation between trip_duration and fare_amount: 0.32
```

The output for the code is shown above. The correlation between the two columns (trip_distance and fare_amount) is a weak positive linear

relationship (r=0.32). This suggests that even though the correlation is positive, the fare_amount does not increase proportionally when the trip_duration increases.

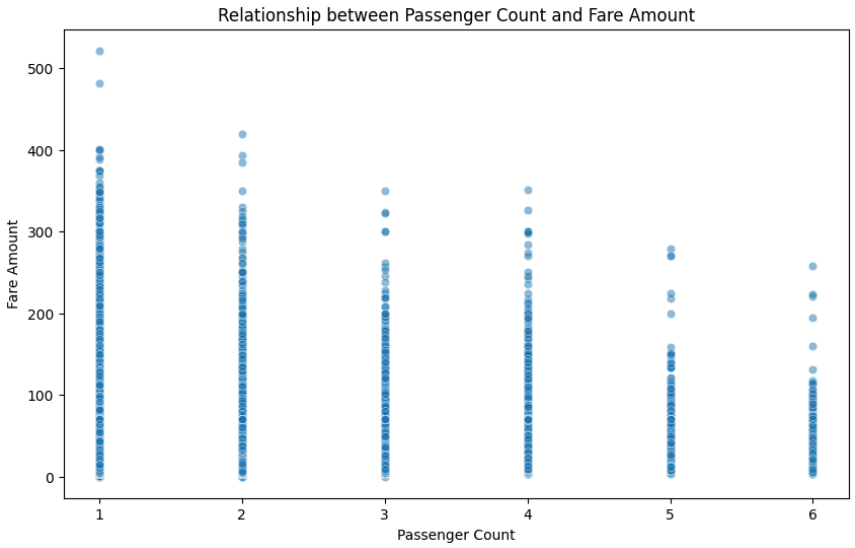
The scatterplot shows 2 dense clusters - one cluster being high fares for short trip durations (trip_duration < 200) and another cluster being for fares mostly below $100. There are horizontal plots of constant fare for different durations which could suggest that they are flat-rate fares or there is a problem with the data that was entered. There are also several outliers where the “trip_duration” exceeds 1000 minutes, which could most likely be data errors.

2 - Show relationship between fare and number of passengers

The code below shows the correlation between fare and number of passengers. I have followed the steps below to produce the desired output.

1) I did not do any manipulations to the columns as it is not necessary to do so.
2) I created a scatterplot using Seaborn with the x-axis being “passenger_count” and the y-axis being “fare_amount”. The data attribute tells the scatterplot function which DataFrame to read from.
3) I calculated Pearson’s correlation using the *.corr()* function, which uses Pearson’s correlation by default. The range of the output correlation lies between -1 and 1 where:
   “Close to 1” means the two variables have a strong positive linear relationship.
   “Close to -1” means the two variables have a strong negative linear relationship
   “Close to 0” means the two variables have little to no linear relationship.

```
# Show relationship between fare and number of passengers
plt.figure(figsize=(10, 6))
sns.scatterplot(data=df1_filtered, x='passenger_count', y='fare_amount', alpha=0.5)
plt.title('Relationship between Passenger Count and Fare Amount')
plt.xlabel('Passenger Count')
plt.ylabel('Fare Amount')
plt.show()
correlation_passenger_fare = df1_filtered['passenger_count'].corr(df1_filtered['fare_amount'])
print(f"\nCorrelation between passenger_count and fare_amount: {correlation_passenger_fare:.2f}")
```




```
Correlation between passenger_count and fare_amount: 0.04
```

Based on the two outputs above, we can see that the two columns (passenger_count and fare_amount) have almost no correlation (r=0.04). This suggests that taxi fares are independent of the number of passengers travelling.

The data points in the scatterplot form vertical clusters which show that the fare amounts vary with each category but do not increase or decrease when passenger count increases. The two columns are inversely proportional to each other.

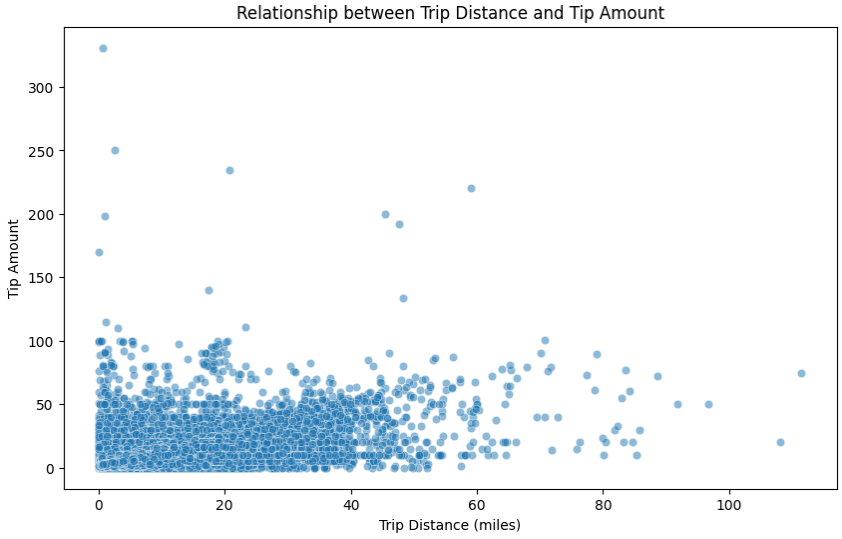
3 - Show relationship between trip and trip distance

The code below shows the correlation between trip and trip_distance. I have followed the steps below to produce the desired output.

1) I did not do any manipulations to the columns as it is not necessary to do so.
2) I created a scatterplot using Seaborn's *.scatterplot()* function, with the x-axis being "trip_distance" and the y-axis being "tip_amount".
3) Similar to the other two visualisations, I calculated Pearson's correlation using the *.corr()* function, which uses Pearson's correlation by default. The range of the output correlation lies between -1 and 1 where:
   "Close to 1" means the two variables have a strong positive linear relationship.
   "Close to -1" means the two variables have a strong negative linear relationship

"Close to 0" means the two variables have little to no linear relationship.

```
# Show relationship between trip and trip distance
plt.figure(figsize=(10, 6))
sns.scatterplot(data=df1_filtered, x='trip_distance', y='tip_amount', alpha=0.5)
plt.title('Relationship between Trip Distance and Tip Amount')
plt.xlabel('Trip Distance (miles)')
plt.ylabel('Tip Amount')
plt.show()
correlation_distance_tip = df1_filtered['trip_distance'].corr(df1_filtered['tip_amount'])
print(f"\nCorrelation between trip_distance and tip_amount: {correlation_distance_tip:.2f}")
```




```
Correlation between trip_distance and tip_amount: 0.80
```

From the outputs, we see that the data in the scatterplot shows an upward trend which means that when the trip_distance increases, the tip_amount also increases. The two columns are proportional to one another.

The two columns also have a strong positive relationship (r=0.80). This could suggest that due to longer trips, more time is spent with the driver and leading to better service.

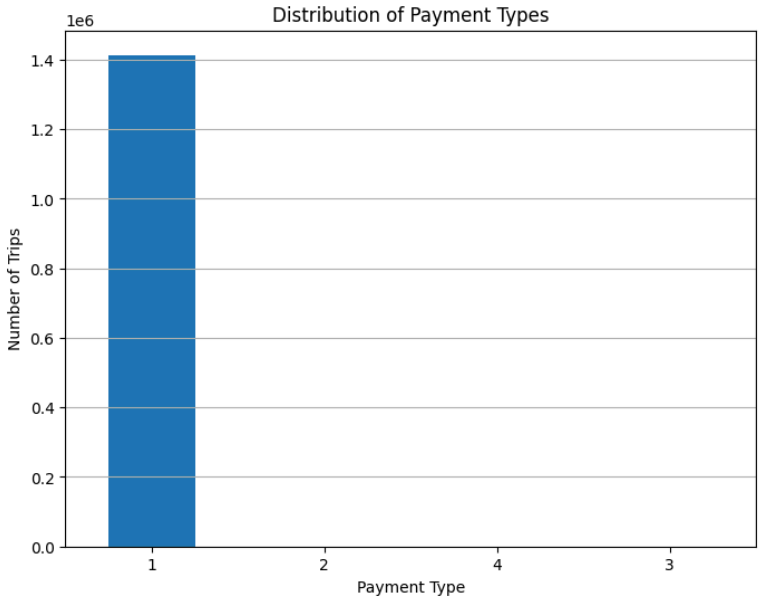
However, there are a few outliers where the trip_distance is very low but the tip_amount is very high. This could be an issue with the data entry. Most trips that are under 30 miles have the densest cluster of data points which could suggest that the tips for the distance are almost proportionate to each other.

3.1.8. Analyse the distribution of different payment types

The code below is used to analyse the different types of payments that have been done. Firstly, I have created a "payment_type_counts" variable which counts the different types of payments that have been done. Next, I

have also created a bar plot such that the distribution would be easy to visualise, as shown below.

```
# Analyse the distribution of different payment types (payment_type).
payment_type_counts = df1_filtered['payment_type'].value_counts()
print("Distribution of Payment Types:\n", payment_type_counts)
plt.figure(figsize=(8, 6))
payment_type_counts.plot(kind='bar')
plt.title('Distribution of Payment Types')
plt.xlabel('Payment Type')
plt.ylabel('Number of Trips')
plt.xticks(rotation=0)
plt.grid(axis='y')
plt.show()
```




```
Distribution of Payment Types:
 payment_type
1    1391592
2         26
4         14
3          4
Name: count, dtype: int64
```

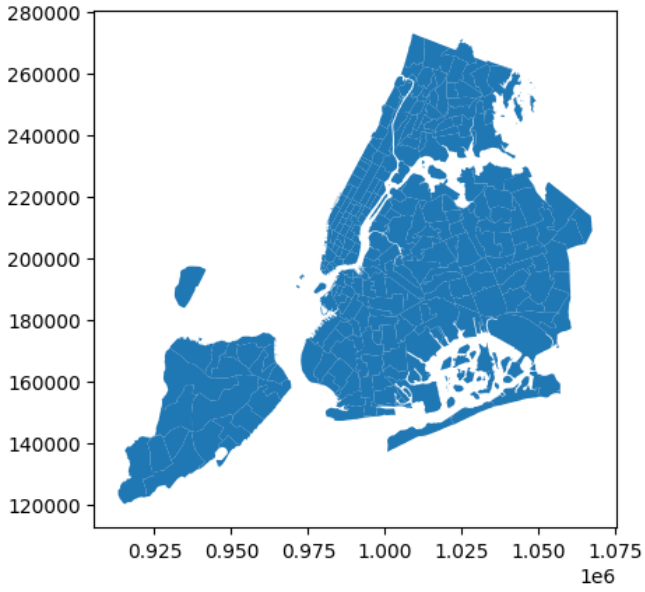
Here, “1” = Credit card, “2” = Cash, “3” = No charge, “4” = Dispute.

Based on this output, we can see that credit card payments were done for most of the trips (1391592 trips). The bar chart for the remaining payment types is barely visible due to their extremely small counts compared to credit card payment_type. This indicates a strong preference for electronic payments.

3.1.9. Load the taxi zones shapefile and display it

```
import geopandas as gpd
f_path = "/content/drive/MyDrive/Vaishu EDA Assignment/taxi_zones/taxi_zones.shp"
# Read the shapefile using geopandas
zones = gpd.read_file(f_path) # read the .shp file using gpd
zones.head()
zones.info()
```

```
<class 'geopandas.geodataframe.GeoDataFrame'>
RangeIndex: 263 entries, 0 to 262
Data columns (total 7 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   OBJECTID    263 non-null    int32
 1   Shape_Leng  263 non-null    float64
 2   Shape_Area  263 non-null    float64
 3   zone        263 non-null    object
 4   LocationID  263 non-null    int32
 5   borough     263 non-null    object
 6   geometry    263 non-null    geometry
dtypes: float64(2), geometry(1), int32(2), object(2)
memory usage: 12.5+ KB
```



Using the geometric parameters to plot the zones on a map. These parameters include shape length, shape area and geometry.

3.1.10. Merge the zone data with trips data

The *.merge()* function is similar to SQL's JOIN whereby it combines the rows from both DataFrames on columns that match. Here, I have given the attributes *"left_on"* and *"right_on"*, where *"left_on"* specifies the column used for matching while "right_on" specifies the column used to match against which is most likely a unique identifier for each zone.

I have also specified that the operation is a LEFT JOIN, with the help of the *"how"* attribute. This means that it keeps all the rows from

"df1_filtered" (left DataFrame) and matches records from "zones" (right DataFrame). The function will fill in missing values when there is no match found. The output of the code is shown below.

```
# Merge zones and trip records using locationID and PULocationID
merged_df = df1_filtered.merge(zones, left_on='PULocationID', right_on='LocationID', how='left')
print("Merged DataFrame info:")
merged_df.info()
```

```
Merged DataFrame info:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1391660 entries, 0 to 1391659
Data columns (total 31 columns):
 #   Column                 Non-Null Count    Dtype
---  ------                 --------------    -----
 0   VendorID               1391660 non-null  int64
 1   tpep_pickup_datetime   1391660 non-null  datetime64[ns]
 2   tpep_dropoff_datetime  1391660 non-null  datetime64[ns]
 3   passenger_count        1391660 non-null  float64
 4   trip_distance          1391660 non-null  float64
 5   RatecodeID             1391660 non-null  float64
 6   store_and_fwd_flag     1391660 non-null  object
 7   PULocationID           1391660 non-null  int64
 8   DOLocationID           1391660 non-null  int64
 9   payment_type           1391660 non-null  int64
 10  fare_amount            1391660 non-null  float64
 11  extra                  1391660 non-null  float64
 12  mta_tax                1391660 non-null  float64
 13  tip_amount             1391660 non-null  float64
 14  tolls_amount           1391660 non-null  float64
 15  improvement_surcharge  1391660 non-null  float64
 16  total_amount           1391660 non-null  float64
 17  congestion_surcharge   1391660 non-null  float64
 18  airport_fee            1391660 non-null  float64
 19  pickup_hour            1391660 non-null  int32
 20  pickup_day_of_week     1391660 non-null  int32
 21  pickup_month           1391660 non-null  int32
 22  pickup_quarter         1391660 non-null  object
 23  trip_duration          1391660 non-null  float64
 24  OBJECTID               1379403 non-null  float64
 25  Shape_Leng             1379403 non-null  float64
 26  Shape_Area             1379403 non-null  float64
 27  zone                   1379403 non-null  object
 28  LocationID             1379403 non-null  float64
 29  borough                1379403 non-null  object
 30  geometry               1379403 non-null  geometry
dtypes: datetime64[ns](2), float64(17), geometry(1), int32(3), int64(4), object(4)
memory usage: 313.2+ MB
```

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

I used the *.value_counts()* function to see how many times PULocationID appears in the DataFrame and returns a Series. The index of the Series is unique PULocationID values while the value is the number of trips that have happened.

Attached to it is the *.reset_index()* function which converts the Series into a DataFrame, such that the index and PULocationID become two separate columns.

I have also renamed the column names from "index" and "PULocationID" to "LocationID" and "number_of_trips" for better clarity when dealing with the data in the DataFrame. Finally, I have printed the first 5 rows of the DataFrame to show the most frequent pickup zones, as shown below in the output.

```
# Group data by location and calculate the number of trips
trips_per_zone = merged_df['PULocationID'].value_counts().reset_index()
trips_per_zone.columns = ['LocationID', 'number_of_trips']
print("Total trips per zone (first 5):")
print(trips_per_zone.head())
```

```
Total trips per zone (first 5):
   LocationID  number_of_trips
0         237            68897
1         161            66533
2         132            63167
3         236            62323
4         162            51587
```

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

The code combines geographic zone information with the number of trips per zone such that it can be used for map visualisations.
I have performed a LEFT JOIN between zones and trips_per_zone, joining on “LocationID” from the previous step.

I used LEFT JOIN so that all the zones from the DataFrame are kept even if there are no records of trips that have happened in a particular zone. If there are no values, I have replaced them with 0 instead of NaN so that it would bring a clearer meaning to the missing values in the DataFrame.

```
# Merge trip counts back to the zones GeoDataFrame
zones_with_trips = zones.merge(trips_per_zone, left_on='LocationID', right_on='LocationID', how='left')
zones_with_trips['number_of_trips'] = zones_with_trips['number_of_trips'].fillna(0)
print("Zones GeoDataFrame with trip counts (first 5):")
display(zones_with_trips.head())
```

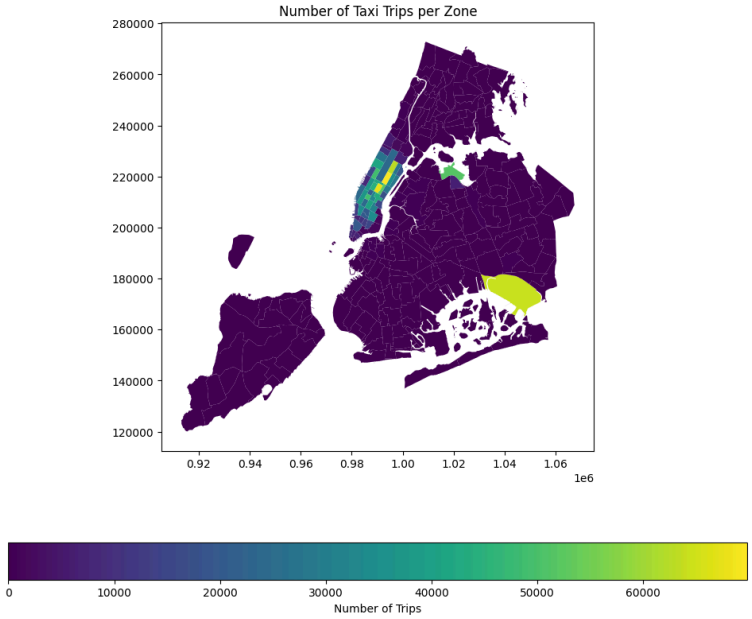
I have displayed the first 5 rows of the new DataFrame (after merging) to verify it. It shows details of a zone such as name of the zone, borough, etc., as shown in the output below.

```
Zones GeoDataFrame with trip counts (first 5):
```

	OBJECTID	Shape_Leng	Shape_Area	zone	LocationID	borough	geometry	number_of_trips
0	1	0.116357	0.000782	Newark Airport	1	EWR	POLYGON ((933100.918 192536.086, 933091.011 19...	37.0
1	2	0.433470	0.004866	Jamaica Bay	2	Queens	MULTIPOLYGON (((1033269.244 172126.008, 103343...	0.0
2	3	0.084341	0.000314	Allerton/Pelham Gardens	3	Bronx	POLYGON ((1026308.77 256767.698, 1026495.593 2...	1.0
3	4	0.043567	0.000112	Alphabet City	4	Manhattan	POLYGON ((992073.467 203714.076, 992068.667 20...	1385.0
4	5	0.092146	0.000498	Arden Heights	5	Staten Island	POLYGON ((935843.31 144283.336, 936046.565 144...	0.0

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

```
# Define figure and axis
fig, ax = plt.subplots(1, 1, figsize = (12, 10))
# Plot the map and display it
zones_with_trips.plot(column='number_of_trips', ax=ax, legend=True,
                      legend_kwds={'label': "Number of Trips", 'orientation': "horizontal"})
plt.title('Number of Taxi Trips per Zone')
plt.show()
```



The output map shows five boroughs within New York City with zones shared according to the number of taxi trips recorded there. Dark purple represents a low number of trips while light yellow represents a high number of trips.

```
# can you try displaying the zones DF sorted by the number of trips?
zones_sorted_by_trips = zones_with_trips.sort_values(by='number_of_trips', ascending=False)
print("Zones sorted by number of trips:")
display(zones_sorted_by_trips.head())
display(zones_sorted_by_trips.tail())
```

```
Zones sorted by number of trips:
```

	OBJECTID	Shape_Leng	Shape_Area	zone	LocationID	borough	geometry	number_of_trips
236	237	0.042213	0.000096	Upper East Side South	237	Manhattan	POLYGON ((993633.442 216961.016, 993507.232 21...	68897.0
160	161	0.035804	0.000072	Midtown Center	161	Manhattan	POLYGON ((991081.026 214453.698, 990952.644 21...	66533.0
131	132	0.245479	0.002038	JFK Airport	132	Queens	MULTIPOLYGON (((1032791.001 181085.006, 103283...	63167.0
235	236	0.044252	0.000103	Upper East Side North	236	Manhattan	POLYGON ((995940.048 221122.92, 995812.322 220...	62323.0
161	162	0.035270	0.000048	Midtown East	162	Manhattan	POLYGON ((992224.354 214415.293, 992096.999 21...	51587.0

	OBJECTID	Shape_Leng	Shape_Area	zone	LocationID	borough	geometry	number_of_trips
175	176	0.151995	0.000658	Oakwood	176	Staten Island	POLYGON ((950393.94 148827.195, 950393.983 148...	0.0
174	175	0.134898	0.000505	Oakland Gardens	175	Queens	POLYGON ((1051776.198 215687.328, 1051892.956 ...	0.0
21	22	0.126170	0.000472	Bensonhurst West	22	Brooklyn	POLYGON ((986318.162 166716.417, 987000.047 16...	0.0
77	78	0.093594	0.000191	East Tremont	78	Bronx	POLYGON ((1014782.254 250130.965, 1014697.776 ...	0.0
1	2	0.433470	0.004866	Jamaica Bay	2	Queens	MULTIPOLYGON (((1033269.244 172126.008, 103343...	0.0

The output above shows the zones sorted by the number of trips. The first table displays the head of the DataFrame while the second table displays the tail of the DataFrame.

3.1.14. Conclude with results

Based on all the analysis that was done and their respective outputs, these are my findings:

- The dataset has a wide range of trip distances but most trips are under 5 miles, indicating that passengers use taxis for local and short journeys.
- Trip durations are varying but most of them are between 5 and 15 minutes, which supports frequent short-distance in populated zones.
- The distribution of total fare amounts is right-skewed with most of the fares being in the acceptable range of less than $20. However, there are a few high-fares which could suggest premium surcharges/fares or longer trips being made.
- Most of the trips have one or two passengers, which implies that single-rider and duo-rider trips are popular in NYC.
- There is a strong positive correlation between trip distance, tip amount and fare amount. This suggests that longer trips earn higher fares and tips, proportionally.
- There are small values of outliers in trip distance, duration and fare variables which represent exceptionally long-distance rides or errors when recording the data.
- Most of the payments are done by credit card and very few to none with cash which suggests that there is more adoption of digital transactions in NYC.
- The average fare per mile is quite constant across all the trips which shows that fare calculation is consistent with the formula being used and has no concerning outliers/irregularities.
- Based on the analysis, the dataset shows stable fare patterns and mostly logical correlations between distance, duration and fare - which is expected for real-world taxi operations.

3.2. Detailed EDA: Insights and Strategies

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

The code below is used to calculate the trip duration in minutes. I have followed the steps below to produce the desired output.

1) Firstly, I have created a new column called “trip_duration” which contains the trip time in minutes for each row. This is done by subtracting the pickup time from the dropoff time. It will give an output as TimeDelta which will be converted to seconds using .dt.total_seconds() and then divided by 60 to get the minutes as it is a convenient form of measurement compared to seconds.
2) Next, I calculated the speed (and saved it into a new column) by using the formula speed = distance/time. I also used *.replace(0,*

np.nan) to avoid any division-by-zero errors and prevent those rows from producing infinity ("inf").

3) I then grouped the trips by their pickup zone, dropoff zone and pickup hour of the day, and calculated the mean speed for each group. This is done to get the average speed for a specific route during a specific hour. The *.reset_index()* function turns the output into a DataFrame with the four columns that were used to group in the first place.
4) To find the slowest routes, I sorted the grouped DataFrame from the previous step in ascending order such that the lowest average speed will be on top. This makes it easier to identify in which areas and at what times taxis are the slowest. I displayed the first 10 slowest routes using the *.head(10)* function, as shown below.

```
# Calculate trip duration in minutes
df1_filtered['trip_duration'] = (df1_filtered['tpep_dropoff_datetime'] - df1_filtered['tpep_pickup_datetime']).dt.total_seconds() / 60
df1_filtered['speed'] = df1_filtered['trip_distance'] / df1_filtered['trip_duration'].replace(0, np.nan)
average_speed_by_route_hour = df1_filtered.groupby(['PULocationID', 'DOLocationID', 'pickup_hour'])['speed'].mean().reset_index()
slowest_routes = average_speed_by_route_hour.sort_values(by='speed', ascending=True)
print("Slowest routes (first 10):")
display(slowest_routes.head(10))
```

```
Slowest routes (first 10):
```

	PULocationID	DOLocationID	pickup_hour	speed
33233	125	113	16	-0.275508
58589	161	125	16	-0.244926
69111	181	89	1	-0.115849
72681	209	209	14	0.000357
74654	216	216	7	0.000371
2282	13	211	0	0.000954
7244	45	211	10	0.000993
20	1	264	11	0.001453
82667	233	43	22	0.001512
62488	163	89	19	0.001844

From the output above, we can see that there are negative speed values which could suggest that dropoff time is less than pickup time, which could mean data inconsistency or recording error of values in the original dataset. Based on the remaining positive speed values, we can assume that the area/route is extremely slow-moving.

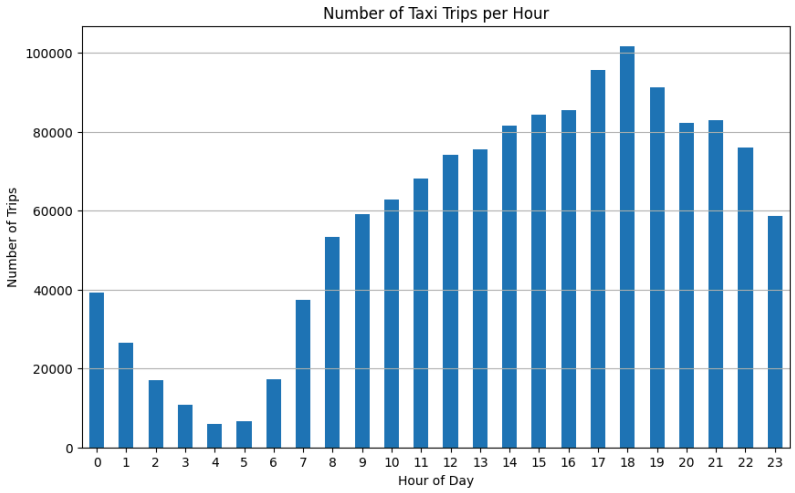
Question: How does identifying high-traffic, high-demand routes help us?

Answer: It helps in identifying potential bottlenecks and optimising routes for faster travel times. This can result in deploying more taxis to these areas during peak hours to reduce passenger waiting times and maximise the number of trips completed which in turn results in increased revenue.

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

1) The *"hourly_trips"* variable counts the number of trips that have occurred in each hour of the day, with the hours ranging from 0 to 23. The *.value_counts()* function sums occurrences of each distinct pickup_hour which in turns returns a Series of key-value pairs (key is *hour* and values are *count*). The *.sort_index()* function reorders the output to a chronologically ascending order.
2) The visualisation created is a bar chart where x-axis is the Hour of day and y-axis is the Number of trips. The *.xticks(rotation = 0)* function is used to set the labels to be at 0 degrees, instead of rotating to keep them horizontal and readable.
3) Finally, "busiest_hour" returns the index label that has the maximum value, with the help of *.idmax()* function while "busiest_hour_trips" returns the maximum number of trips that occurred during the busiest hour.

```
# Visualise the number of trips per hour and find the busiest hour
hourly_trips = df1_filtered['pickup_hour'].value_counts().sort_index()
plt.figure(figsize=(10, 6))
hourly_trips.plot(kind='bar')
plt.title('Number of Taxi Trips per Hour')
plt.xlabel('Hour of Day')
plt.ylabel('Number of Trips')
plt.xticks(rotation=0)
plt.grid(axis='y')
plt.show()
busiest_hour = hourly_trips.idxmax()
busiest_hour_trips = hourly_trips.max()
print(f"\nBusiest hour: {busiest_hour}\n Number of trips during the busiest hour: {busiest_hour_trips}")
```




```
Busiest hour: 18
 Number of trips during the busiest hour: 101552
```

As seen from the outputs above, we can observe that the busiest hour is at 6pm/18:00 with the number of trips being at the highest of 101552.

From the bar chart, we can interpret that between hours 3 and 5, the number of trips are at the lowest as this is when city traffic is usually at

minimal. From hour 6 onwards, there is a gradual rise as the taxi demand increases for commuters to start their day travels.

Based on this, the operational times of taxi availability and driver scheduling should be optimised for evening hours (4pm to 8pm), when demand is the highest while early morning hours (3am to 5 am) may require a few drivers only since demand is minimal.

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

Scaling up is a process that is done to show or estimate what a population total would look like but with a small amount of data.

An assumption here is that the data being sampled is not biased. If the sample data was biased, it could mislead the scaling process.

1) "sample_fraction = 0.05" means that 5% of the dataset will be used to create a sample such that values can be used to estimate values for an entire population.
2) Next, I have done the estimation by dividing the length of DataFrame by the sample fraction. The output may result in a float value.
3) Finally, I have also done the same process for another data which is "busiest_hour_trips".

```
# Scale up the number of trips
# Fill in the value of your sampling fraction and use that to scale up the numbers
sample_fraction = 0.05
scaled_total_trips = len(df1) / sample_fraction
print(f"Sampled total trips: {len(df1)}")
print(f"Scaled estimated total trips for the year: {scaled_total_trips:,.0f}")
scaled_busiest_hour_trips = busiest_hour_trips / sample_fraction
print(f"Scaled estimated trips during the busiest hour ({busiest_hour}): {scaled_busiest_hour_trips:,.0f}")
```

```
Sampled total trips: 1820597
Scaled estimated total trips for the year: 36,411,940
Scaled estimated trips during the busiest hour (18): 2,031,040
```

Based on the outputs shown above, we can see that 1820597 rows of data have been sampled from the entire dataset. The scaled estimated total trips are 36411940, which is an important metric for understanding the annual taxi trip volume. Similarly, the scaled estimated trips during the busiest hour (hour 18, 6PM) is 2031040.

3.2.4. Compare hourly traffic on weekdays and weekends

The code below is used to compare the number of trips during different hours of the day on both weekdays and weekends. The steps below are done to get the desired outputs.

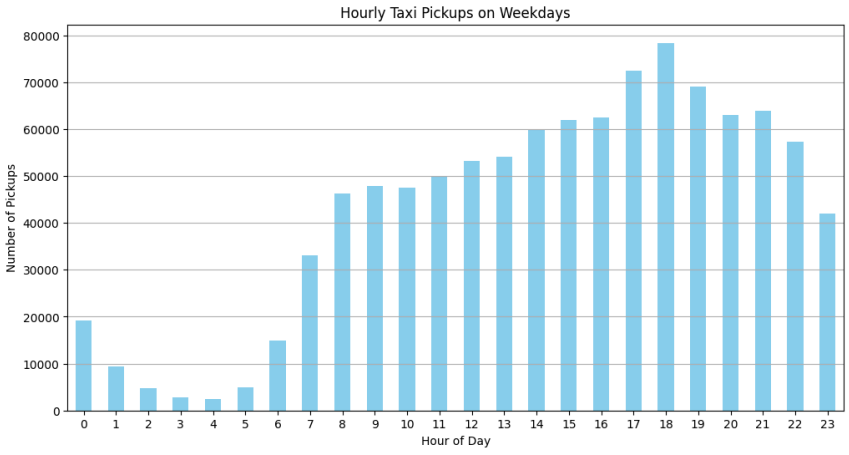
1) I have created a new column ('pickup_day_of_week') which contains the extracted day of week (0 being Monday and 6 being

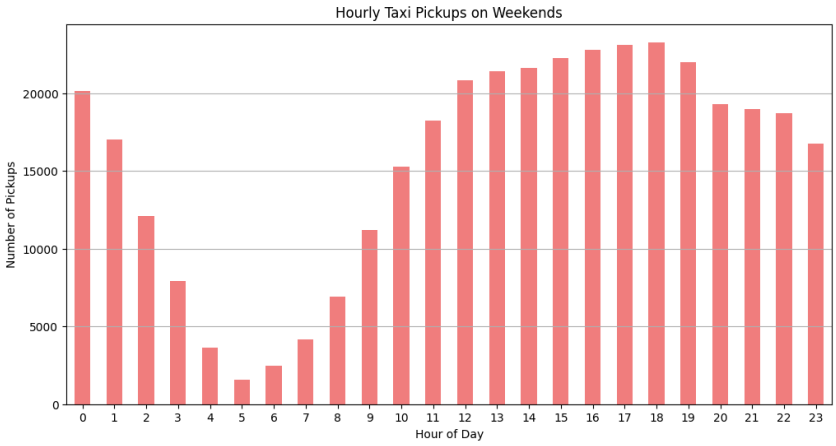
Sunday). This helps classify each trip based on which day it happens.

2) Based on what I have done in step 1, I continued to split the days into two DataFrames - weekday_df and weekend_df. If “pickup_day_of_week” is less than 5, then those days are considered as weekdays (0 being Monday and 4 being Friday). Similarly, if “pickup_day_of_week” is more than or equal to 5, those days are considered as weekends (5 being Saturday, 6 being Sunday).
3) Then, I counted the number of trips per pickup hour using the *.value_counts()* function. I have also used *.sort_index()* to order the values chronologically, which will be easier to interpret.
4) Finally, I have created two separate visualisations for the hourly taxi pickups for both weekdays and weekends.

```
# Compare traffic trends for the week days and weekends
df1_filtered['pickup_day_of_week'] = df1_filtered['tpep_pickup_datetime'].dt.dayofweek
weekday_df = df1_filtered[df1_filtered['pickup_day_of_week'] < 5].copy()
weekend_df = df1_filtered[df1_filtered['pickup_day_of_week'] >= 5].copy()
hourly_trips_weekday = weekday_df['pickup_hour'].value_counts().sort_index()
hourly_trips_weekend = weekend_df['pickup_hour'].value_counts().sort_index()
plt.figure(figsize=(12, 6))
hourly_trips_weekday.plot(kind='bar', color='skyblue')
plt.title('Hourly Taxi Pickups on Weekdays')
plt.xlabel('Hour of Day')
plt.ylabel('Number of Pickups')
plt.xticks(rotation=0)
plt.grid(axis='y')
plt.show()

plt.figure(figsize=(12, 6))
hourly_trips_weekend.plot(kind='bar', color='lightcoral')
plt.title('Hourly Taxi Pickups on Weekends')
plt.xlabel('Hour of Day')
plt.ylabel('Number of Pickups')
plt.xticks(rotation=0)
plt.grid(axis='y')
plt.show()
```





Questions:

a) **What can you infer from the above patterns?**
Weekday patterns show distinct rush hour peaks during morning and evening commutes, which represents work-related travel.

Similarly, weekend patterns show peaks later in the day (i.e. afternoon onwards), indicating more leisurely travel.

b) **How will finding busy and quiet hours for each day help us?**
It will help in deploying more taxi drivers during peak hours in high-demand areas and reduce the number of taxi drivers during off peak hours to minimise idle time. It can also help adjusting the price based on demand - surcharges during peak hours and discounts during off-peak hours.

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

To identify the top 10 zones with high hourly pickups and dropoffs, I have done the following:

1) I have selected the PULocationID column and counted the number of unique pickup zones, using *.value_counts()*. I have also added the *.head(10)* function to show the top 10 zones with high numbers of pickups and the *.reset_index()* function to convert the Series into a DataFrame and save the result to “top_10_pickup_zones”.
2) Next, I have changed the default column names that came with using the *.reset_index()* function, by using the *.columns* attribute to change to ‘PULocationID’ and ‘number_of_pickups’. Then displayed the results
3) Finally, I have followed the same process to get results for top 10 dropoff zones.

```
# Find top 10 pickup and dropoff zones
top_10_pickup_zones = df1_filtered['PULocationID'].value_counts().head(10).reset_index()
top_10_pickup_zones.columns = ['PULocationID', 'number_of_pickups']
print("Top 10 Pickup Zones:")
display(top_10_pickup_zones)
top_10_dropoff_zones = df1_filtered['DOLocationID'].value_counts().head(10).reset_index()
top_10_dropoff_zones.columns = ['DOLocationID', 'number_of_dropoffs']
print("\nTop 10 Dropoff Zones:")
display(top_10_dropoff_zones)
```

```
Top 10 Pickup Zones:
```

	PULocationID	number_of_pickups
0	237	68897
1	161	66533
2	132	63167
3	236	62323
4	162	51587
5	138	51011
6	142	47987
7	186	47825
8	230	43666
9	170	42193

```
Top 10 Dropoff Zones:
```

	DOLocationID	number_of_dropoffs
0	236	66523
1	237	61775
2	161	54798
3	239	42367
4	170	42303
5	142	40703
6	162	40262
7	141	39103
8	230	38832
9	68	35258

Based on this, we can see that there are some zones where both are popular for pickups and dropoffs, however at different ranks in the respective tables. For example, Zone ID of 237 is the most popular place for taxi pickup but is the second most popular zone for taxi dropoffs.

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

This code is used to compute the pickup/dropoff ratio per zone and shows the top 10 zones with the largest and smallest ratios.

Having a large ratio means that there were more pickups than dropoffs, and having a small ratio means the opposite. The steps I have followed are below.

1) Similar to the previous part, I have selected the pickup zone ID column and counted the values present in it.
2) Next, I have changed the column names to “LocationID” and “pickup_count” to make them consistent for merging later.
3) Similar to step 1, I have done the same for dropoff zones (DDLocationID), which produces a DataFrame of dropoff counts per zone.

4) The *pd.merge()* function is used to merge both the pickup and dropoff count DataFrames on LocationID. I have used OUTER JOIN so that the zones that appear only in pickups or only in dropoffs are still included. If there are any missing values, the value will be NaN. Doing this merge will result in the "zone_counts" DataFrame to have three columns.
5) I have created a new column "pickup_dropoff_ratio" which is used to calculate the ratio with the formula: pickups/dropoffs. The code will add 1e-6 by default to the denominator to avoid any division-by-zero errors in the case that dropoff_count has zero values.
   If Ratio >= 1, there are more pickups than dropoffs. This suggests that the zone is mostly a start point of travel.
   If Ratio ~= 1, both pickups and dropoffs are roughly balanced.
   If Ratio <= 1, then there are more dropoffs than pickups. This suggests that the zone is mostly an end point of travel.
6) Lastly, I have displayed the top 10 values for both "top_10_ratio" and "bottom_10_ratio" in ascending order.

```
# Find the top 10 and bottom 10 pickup/dropoff ratios
pickup_counts = df1_filtered['PULocationID'].value_counts().reset_index()
pickup_counts.columns = ['LocationID', 'pickup_count']
dropoff_counts = df1_filtered['DOLocationID'].value_counts().reset_index()
dropoff_counts.columns = ['LocationID', 'dropoff_count']
zone_counts = pd.merge(pickup_counts, dropoff_counts, on='LocationID', how='outer').fillna(0)
zone_counts['pickup_dropoff_ratio'] = zone_counts['pickup_count'] / (zone_counts['dropoff_count'] + 1e-6)
top_10_ratio = zone_counts.sort_values(by='pickup_dropoff_ratio', ascending=False).head(10)
bottom_10_ratio = zone_counts.sort_values(by='pickup_dropoff_ratio', ascending=True).head(10)
print("Top 10 Pickup/Dropoff Ratios:")
display(top_10_ratio)
print("\nBottom 10 Pickup/Dropoff Ratios:")
display(bottom_10_ratio)
```

```
Top 10 Pickup/Dropoff Ratios:
```

	LocationID	pickup_count	dropoff_count	pickup_dropoff_ratio
193	199	1.0	0.0	1000000.000000
69	70	6383.0	465.0	13.726882
126	132	63167.0	13433.0	4.702375
132	138	51011.0	17466.0	2.920589
180	186	47825.0	29014.0	1.648342
108	114	19531.0	14017.0	1.393379
42	43	23327.0	17008.0	1.371531
243	249	33637.0	25215.0	1.334008
156	162	51587.0	40262.0	1.281283
98	100	21144.0	17357.0	1.218183

Bottom 10 Pickup/Dropoff Ratios:

	LocationID	pickup_count	dropoff_count	pickup_dropoff_ratio
21	22	0.0	126.0	0.0
215	221	0.0	16.0	0.0
58	59	0.0	8.0	0.0
57	58	0.0	23.0	0.0
26	27	0.0	17.0	0.0
166	172	0.0	4.0	0.0
204	210	0.0	106.0	0.0
29	30	0.0	5.0	0.0
168	174	0.0	68.0	0.0
22	23	0.0	24.0	0.0

Based on the outputs above, we can see that the top 10 pickup/dropoff ratios are mostly above 1, which suggests that those zones are mostly the starting point of journeys. Likewise, we see that the bottom 10 pickup/dropoff ratios are 0, which means that those zones have higher dropoffs than pickups, suggesting that they are mostly ending points of journeys.

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

The night hours in this case are from 11pm to 5 am. The steps followed are:

1) I filtered the rows using a Boolean mask which selects rows whose pickup hour is from 23 (11pm) or up to 4 (midnight to 5am). I have used the *.copy()* function to create the new DataFrame object instead of modifying the original DataFrame. This has been saved in a new DataFrame called “night_hours _df”.
2) Next, I selected the “PULocationID” column from the new DataFrame and counted the number of values present in it using *.value_counts()*. It returns the top 10 most frequent pickup zone IDs during night hours, with the *.head(10)* function.
3) I have renamed the default column names that were created by *.reset_index()* in the previous step to meaningful names such as “PULocationID” and “number_of_night_pickups”, using the *.columns* attribute.
4) Displaying the top 10 pickup zones during night hours using the *.display()* function
5) I have repeated the same steps (1 to 4) but for the top 10 dropoff zones within the night subset.

```
# During night hours (11pm to 5am) find the top 10 pickup and dropoff zones
# Note that the top zones should be of night hours and not the overall top zones
night_hours_df = df1_filtered[
    (df1_filtered['pickup_hour'] >= 23) | (df1_filtered['pickup_hour'] < 5)
].copy()
top_10_night_pickup_zones = night_hours_df['PULocationID'].value_counts().head(10).reset_index()
top_10_night_pickup_zones.columns = ['PULocationID', 'number_of_night_pickups']
print("Top 10 Nighttime Pickup Zones (11 PM - 5 AM):")
display(top_10_night_pickup_zones)
top_10_night_dropoff_zones = night_hours_df['DOLocationID'].value_counts().head(10).reset_index()
top_10_night_dropoff_zones.columns = ['DOLocationID', 'number_of_night_dropoffs']
print("\nTop 10 Nighttime Dropoff Zones (11 PM - 5 AM):")
display(top_10_night_dropoff_zones)
```

Top 10 Nighttime Pickup Zones (11 PM - 5 AM

	PULocationID	number_of_night_pickups
0	79	12501
1	249	10326
2	132	8811
3	148	7824
4	48	7478
5	114	7230
6	230	5479
7	186	4857
8	138	4703
9	164	4674

Top 10 Nighttime Dropoff Zones (11 PM - 5 AM

	DOLocationID	number_of_night_dropoffs
0	79	6642
1	48	5100
2	170	4772
3	107	4655
4	141	4356
5	263	4232
6	68	4168
7	249	4063
8	236	3582
9	229	3567

In the output for top 10 pickup zones in the night, zone 79 has the highest pickup, suggesting that it could be a busy nightlife or transportation hub. When compared with top 10 dropoff zones in the night, zone 79 also has the highest dropoffs, suggesting that it is also a busy and important destination at night. As there is high activity in specific zones, we can infer that these zones are considered as nighttime mobility hubs.

3.2.8. Find the revenue share for nighttime and daytime hours

1) Create a new DataFrame “night_hours_df” from an existing DataFrame with certain Boolean conditions where the pickup hour is between 11pm and 5am. I also created a copy of the subset to avoid changes to the original data.
2) Calculate the nighttime revenue (“night_revenue”) using the sum of values (*.sum()*) in the “total_amount” column of the new DataFrame created in the previous step.
   An assumption made is that all the values in this column are numeric as the sum function will skip any NaN values by default.
3) Calculate total revenue from the original dataset using the *.sum()* function. The DataFrame used here is “df1_filtered”, which was created a while back.

4) Compute share of revenue earned at night by dividing "night_revenue" and "total_revenue" (night revenue/total revenue).
5) Similarly, follow the steps again to calculate daytime revenue.
6) Display the output using formatting such as:
   .2f, which formats numbers using commas and two decimal places.
   .2%, which formats a decimal as a percentage with two decimal places.

```
# Filter for night hours (11 PM to 5 AM)
night_hours_df = df1_filtered[
    (df1_filtered['pickup_hour'] >= 23) | (df1_filtered['pickup_hour'] < 5)
].copy()
night_revenue = night_hours_df['total_amount'].sum()
total_revenue = df1_filtered['total_amount'].sum()
night_revenue_share = night_revenue / total_revenue
print(f"Nighttime Revenue (11 PM - 5 AM): ${night_revenue:,.2f}")
print(f"Total Revenue: ${total_revenue:,.2f}")
print(f"Nighttime Revenue Share: {night_revenue_share:.2%}")
daytime_revenue = total_revenue - night_revenue
daytime_revenue_share = daytime_revenue / total_revenue

print(f"\nDaytime Revenue: ${daytime_revenue:,.2f}")
print(f"Daytime Revenue Share: {daytime_revenue_share:.2%}")
```

```
Nighttime Revenue (11 PM - 5 AM): $4,666,720.00
Total Revenue: $41,224,582.07
Nighttime Revenue Share: 11.32%

Daytime Revenue: $36,557,862.07
Daytime Revenue Share: 88.68%
```

From the output above, we see that the nighttime revenue is $4,666,720.00 which is 11.32% of the total revenue. The daytime revenue is $36,557,862.07 which is 88.68% of the total revenue. From this, we can infer that there is more demand and taxi activity during the day time, when people are commuting to work, running errands, etc.

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

1) Calculate the fare per miler per passenger while avoiding any divisions by 0, using the *.replace()* function, which replaces NaN values to 0.
   Formula used to calculate: fare_amount/(trip_distance * passenger_count).
2) Group the output by passenger_count and calculate the average fare per mile per passenger. This is done by using both *.groupby()* and *.mean()* functions. Display the output at the end.

```
# Analyse the fare per mile per passenger for different passenger counts
df1_filtered['fare_per_mile_per_passenger'] = df1_filtered['fare_amount'] / (df1_filtered['trip_distance'].replace(0, np.nan)
* df1_filtered['passenger_count'].replace(0, np.nan))
average_fare_per_mile_per_passenger = df1_filtered.groupby('passenger_count')['fare_per_mile_per_passenger'].mean().reset_index()
print("Average Fare per Mile per Passenger for Different Passenger Counts:")
display(average_fare_per_mile_per_passenger)
```

```
Average Fare per Mile per Passenger for Different Passenger Counts:
```

	passenger_count	fare_per_mile_per_passenger
0	1.0	9.269508
1	2.0	5.282259
2	3.0	3.490572
3	4.0	4.256864
4	5.0	1.535935
5	6.0	1.365900

From the output above, solo passengers pay $9.26 on average while a group of 5 passengers pay $1.36 each on average. This shows that the fare per mile per passenger decreases as the passenger count increases which suggests that shared rides offer better cost efficiency. There is some variation for 4 passengers which could have arisen from low sample sizes or difference in trip distances.

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

1) Create a new column “fare_per_mile” by dividing fare_amount with trip_distance. An assumption made is that there may be zeros present in the “trip_distance” column hence, to avoid any division-by-zero errors, I have replaced the 0s with NaN.
2) Next, I used the *.groupby()* function to group by the day of week and then computed the average fare using the *.mean()* function. To convert the computation back to a DataFrame from a Series, I have used the *.reset_index()* function. An assumption made here is that “pickup-day_of_week” uses integer codes (0 to 6) and will ignore NaN values, which does not generate any bias in the mean but only reduces the sample size.
3) Map numeric day codes to readable day names so that the output is easily understandable. This is done by using a dictionary to map the integer codes (keys) to the days of the week (values) where the key-value pair will look something like this: “{0: “Monday”, …}
4) Display the result of the day_of_week, which prints a DataFrame as output.
5) Similar to the previous steps, I have done the same but on the “pickup-hour” column instead.

```
# Compare the average fare per mile for different days and for different times of the day
df1_filtered['fare_per_mile'] = df1_filtered['fare_amount'] / df1_filtered['trip_distance'].replace(0, np.nan)
average_fare_per_mile_by_day = df1_filtered.groupby('pickup_day_of_week')['fare_per_mile'].mean().reset_index()
day_names = {0: 'Monday', 1: 'Tuesday', 2: 'Wednesday', 3: 'Thursday', 4: 'Friday', 5: 'Saturday', 6: 'Sunday'}
average_fare_per_mile_by_day['pickup_day_of_week'] = average_fare_per_mile_by_day['pickup_day_of_week'].map(day_names)
print("Average Fare per Mile by Day of the Week:")
display(average_fare_per_mile_by_day)
average_fare_per_mile_by_hour = df1_filtered.groupby('pickup_hour')['fare_per_mile'].mean().reset_index()
print("\nAverage Fare per Mile by Hour of the Day:")
display(average_fare_per_mile_by_hour)
```

Average Fare per Mile by Day of the W

	pickup_day_of_week	fare_per_mile
0	Monday	9.397545
1	Tuesday	9.510813
2	Wednesday	9.352389
3	Thursday	10.118252
4	Friday	9.605018
5	Saturday	9.389702
6	Sunday	9.940808

Average Fare per Mile by Hour of

	pickup_hour	fare_per_mile
0	0	10.349793
1	1	8.144467
2	2	7.974625
3	3	8.236583
4	4	11.306763
5	5	11.271275
6	6	8.234898
7	7	9.004952
8	8	9.004462
9	9	9.439142
10	10	9.434953
11	11	9.416335
12	12	9.993595
13	13	10.088884
14	14	10.397093
15	15	11.581268
16	16	11.790447
17	17	10.269342
18	18	9.424657
19	19	9.985892
20	20	8.139992
21	21	7.868809
22	22	8.704473
23	23	8.421180

Based on the outputs above, we see that the fare per mile is almost constant throughout the week, ranging between $9.37 to $10.11 per mile. However, Thursday has the highest average fare per mile ($9.89) while Monday has the lowest average fare per mile (48.97). This could suggest that there has been an increased demand towards the end of the weekdays to the weekends.

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

Group the dataset by unique vendor IDs using the *.groupby()* function and within each vendor group, find the average of the “fare_per_mile” values to show how much each vendor charges per mile of its rides on average.

```
# Compare fare per mile for different vendors
average_fare_per_mile_by_vendor = df1_filtered.groupby('VendorID')['fare_per_mile'].mean().reset_index()

print("Average Fare per Mile by VendorID:")
display(average_fare_per_mile_by_vendor)
```

```
Average Fare per Mile by VendorID:
```

	VendorID	fare_per_mile
0	1	7.899321
1	2	10.172882

Based on the output above, we see that there are only two different vendors with their IDs being 1 and 2. Each vendor is different and hence, the average fare per mile for the two vendors is $7.89 and $10.17 respectively. We can infer that vendor 2 has a higher pricing or operates under different conditions such that it leads to higher fare prices per mile. In the end, this could cause a competition between both regarding rider preferences as passengers generally prefer to hop on to taxis with lower fares per mile.

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

1) Define the distance bins and labels where the bins are a list of numeric breakpoints that partition trip distances into intervals. The last bin captures all the distances that are more than or equal to 5 miles with the help of *float("inf")* which represents infinity. The "distance_labels" variable contains human-readable and easily understandable names for each bin in the same order.
2) Create a categorical column using *pd.cut()* which assigns each "trip_distance" to one of the bins. The attribute *"right=False"* means that bins are left-inclusive but right-exclusive.
   An assumption is that "trip_distance" is numeric and non-negative.
3) Group by the "vendorID" and "distance_tier" using *.groupby()* and calculate the average of each group using the *.mean()* function.

```
# Defining distance tiers
distance_bins = [0, 2, 5, float('inf')]
distance_labels = ['0-2 miles', '2-5 miles', '> 5 miles']
df1_filtered['distance_tier'] = pd.cut(df1_filtered['trip_distance'], bins=distance_bins, labels=distance_labels, right=False)
average_fare_per_mile_by_vendor_tier = df1_filtered.groupby(['VendorID', 'distance_tier'])['fare_per_mile'].mean().reset_index()
print("Average Fare per Mile by Vendor and Distance Tier:")
display(average_fare_per_mile_by_vendor_tier)
```

```
Average Fare per Mile by Vendor and Distance Tier:
```

	VendorID	distance_tier	fare_per_mile
0	1	0-2 miles	9.588194
1	1	2-5 miles	6.435918
2	1	> 5 miles	4.484402
3	2	0-2 miles	13.899232
4	2	2-5 miles	6.552313
5	2	> 5 miles	4.510805

Based on the output above, we see that short trips have the highest fares per mile, medium trips have lower fares per mile and long trips have the lowest fare per mile. We can infer that as the distances increase, the fixed components of base fares and flat fees remain constant.
There are also vendor-level differences where vendor 2's average fare per mile for each type of distance is significantly higher than vendor 1's average fare per mile.

3.2.13. Analyse the tip percentages

1) Create a new column within "df1_filtered" called "tip_percentage" and compute it using the following formula: "tip_amount"/ "total_amount" * 100. An assumption made here is that both these columns are numeric and if there is any zero value in any of the rows, it will be replaced with NaN.
2) Analysing the average tip percentage by grouping "distance_tier" and then compute the average tip percentage for each of these groups using the *.mean()* function
3) Analyse the average tip percentage by grouping "passenger_count" and calculating the mean of each group created.
4) Analyse the average tip percentage by grouping "pickup_hour" and then calculating the average tip percentage using the *.mean()* function for each group. It shows the average tipping percentage changes over the day.
5) Analyse average tip percentage by grouping "pickup-day_of_week" and then calculating the mean of each group. This replaces the numeric day codes with names that are easily understandable.

```
#  Analyze tip percentages based on distances, passenger counts and pickup times
df1_filtered['tip_percentage'] = (df1_filtered['tip_amount'] / df1_filtered['total_amount'].replace(0, np.nan)) * 100
average_tip_percentage_by_distance = df1_filtered.groupby('distance_tier')['tip_percentage'].mean().reset_index()
print("Average Tip Percentage by Distance Tier:")
display(average_tip_percentage_by_distance)
average_tip_percentage_by_passenger = df1_filtered.groupby('passenger_count')['tip_percentage'].mean().reset_index()
print("\nAverage Tip Percentage by Passenger Count:")
display(average_tip_percentage_by_passenger)
average_tip_percentage_by_hour = df1_filtered.groupby('pickup_hour')['tip_percentage'].mean().reset_index()
print("\nAverage Tip Percentage by Pickup Hour:")
display(average_tip_percentage_by_hour)
average_tip_percentage_by_day = df1_filtered.groupby('pickup_day_of_week')['tip_percentage'].mean().reset_index()
day_names = {0: 'Monday', 1: 'Tuesday', 2: 'Wednesday', 3: 'Thursday', 4: 'Friday', 5: 'Saturday', 6: 'Sunday'}
average_tip_percentage_by_day['pickup_day_of_week'] = average_tip_percentage_by_day['pickup_day_of_week'].map(day_names)
print("\nAverage Tip Percentage by Day of the Week:")
display(average_tip_percentage_by_day)
```

Output 1

Average Tip Percentage by Distance Tier:

	distance_tier	tip_percentage
0	0-2 miles	15.630505
1	2-5 miles	15.167305
2	> 5 miles	15.113035

Output 2

Average Tip Percentage by Passenger Count:

	passenger_count	tip_percentage
0	1.0	15.400274
1	2.0	15.435323
2	3.0	15.426731
3	4.0	15.576170
4	5.0	15.426722
5	6.0	15.455855

Output 3

Average Tip Percentage by Pickup

	pickup_hour	tip_percentage
0	0	15.161724
1	1	15.206428
2	2	15.120691
3	3	15.255431
4	4	15.305470
5	5	15.270963
6	6	15.176772
7	7	15.199196
8	8	15.313074
9	9	15.519697
10	10	15.660461
11	11	15.698405
12	12	15.682505
13	13	15.708098
14	14	15.636176
15	15	15.583849
16	16	15.383479
17	17	15.269268
18	18	15.230430
19	19	15.140472
20	20	15.359729
21	21	15.441299
22	22	15.336702
23	23	15.228230

Output 4

Average Tip Percentage by Day of the Week:

	pickup_day_of_week	tip_percentage
0	Monday	15.418022
1	Tuesday	15.419743
2	Wednesday	15.434450
3	Thursday	15.407549
4	Friday	15.405119
5	Saturday	15.397820
6	Sunday	15.387273

Output 1: Average tip percentage by distance tier

By the distance groups 0-2 miles, 2-5 miles and >5 miles, the tip percentages are 15.63%, 15.17% and 15.09% respectively. This suggests that passengers roughly tip similar amounts regardless of trip distance, number of passengers and time of day.

Output 2: Average tip percentage by passenger counts

By the passenger counts, ranging from 1 to 6, the average percentage of tipping is approximately between 15.40% to 15.52%. This suggests that the number of passengers for the trip does not significantly impact how much passengers tip.

Output 3: Average tip percentage by pickup hour
By the hour of day, the value of tip percentage is around 15.10% to 15.68% with the highest percentages being around hours 10 to 13, which is almost afternoon hours, and the lower-end of percentages in early morning hours. This suggests that during midday hours, there could be higher traffic, causing tipping percentage to be towards the higher end.

Output 4: Average tip percentage by day of the week
By the days of the week (Monday to Sunday), the tipping percentage range is small, being around 15.39% to 15.43%. This suggests that the percentage is almost identical across different days and indicates that the customer tipping behaviour is quite consistent throughout the week with weekends and promotions, if any, do not have much effect on tipping patterns.

3.2.14. Analyse the trends in passenger count

1) Group the “df1_filtered” dataset by the column “pickup_hour” using the *.groupby()* function. Then, calculate the average passenger count by pickup hour groups using the *.mean()* function. I have also used *.reset_index()* to convert the grouped data back into a clean DataFrame with two columns - “pickup_hours” and “passenger_count”. Then, display the average passenger count by pickup hour using the *display()* function.
2) Similar to the previous step, group the “df1_filtered” dataset by the column “pickup_day_of_week”, which uses numeric codes where 0 means Monday and 6 refers to Sunday. Then, calculate the average passenger count and display it using the *display()* function.
3) Create a dictionary called “day_names” that maps each numeric code of the week, which are the keys, to its corresponding name, which are the values. This helps in making it easier to read and understand. Then, display the entire DataFrame.

```
# See how passenger count varies across hours and days
average_passenger_count_by_hour = df1_filtered.groupby('pickup_hour')['passenger_count'].mean().reset_index()
print("Average Passenger Count by Pickup Hour:")
display(average_passenger_count_by_hour)
average_passenger_count_by_day = df1_filtered.groupby('pickup_day_of_week')['passenger_count'].mean().reset_index()
day_names = {0: 'Monday', 1: 'Tuesday', 2: 'Wednesday', 3: 'Thursday', 4: 'Friday', 5: 'Saturday', 6: 'Sunday'}
average_passenger_count_by_day['pickup_day_of_week'] = average_passenger_count_by_day['pickup_day_of_week'].map(day_names)
print("\nAverage Passenger Count by Day of the Week:")
display(average_passenger_count_by_day)
```

Average Passenger Count by Pickup Hour

	pickup_hour	passenger_count
0	0	1.432045
1	1	1.438042
2	2	1.456034
3	3	1.443045
4	4	1.422137
5	5	1.287805
6	6	1.251560
7	7	1.282056
8	8	1.293166
9	9	1.304902
10	10	1.347974
11	11	1.354271
12	12	1.373971
13	13	1.380292
14	14	1.389933
15	15	1.400786
16	16	1.392216
17	17	1.376769
18	18	1.364793
19	19	1.381906
20	20	1.388904
21	21	1.417137
22	22	1.422343
23	23	1.418386

Average Passenger Count by Day of the Week

	pickup_day_of_week	passenger_count
0	Monday	1.347681
1	Tuesday	1.324360
2	Wednesday	1.321544
3	Thursday	1.336568
4	Friday	1.395049
5	Saturday	1.479311
6	Sunday	1.459178

Output 1: Average passenger count by pickup hour
There are small variations in passenger counts across all hours with mean being approximately 1.37 with the lowest at hour 6 being 1.25, which implies that these passengers are mostly travelling alone, and highest at hour 2 being 1.45. This implies that there is a higher average passengers per trip which could be groups returning from various activities.

Output 2: Average passenger count by day of the week
The weekends - Saturday and Sunday, have a higher average passenger count at 1.479 and 1.459 respectively. This suggests that there are more shared trips due to going out and enjoying some weekend activities.
The weekdays, specifically Monday to Thursday, have a lower average, suggesting travels to offices, schools and colleges. However, on Friday there is a slight increase in passenger count, suggesting people tend to go out in the evening or is the start of their weekend travels.

3.2.15. Analyse the variation of passenger counts across zones

1) Group the dataset by the column “PULocationID”, representing the pickup zone’s unique identifier. Then, calculate the mean of “passenger_count” using the *.mean()* function.

2) Rename the columns such that it would be easier to read and understand. “PULocationID” is changed to “LocationID” and “passenger_count” is changed to “average_passenger_count”.
3) Merge two DataFrames - “average_passenger_count_by_zone” and “zone_with_trips”, which contains geographic data for NYC taxi zones. I used LEFT JOIN to ensure that all zones from the second DataFrame are kept and if there have been no trips that occurred, those rows will have NaN values. Then, I have displayed the first 5 rows (by default) using the *.head()* function within the *display()* function.

```
# For a more detailed analysis, we can use the zones_with_trips GeoDataFrame
# Create a new column for the average passenger count in each zone.
average_passenger_count_by_zone = df1_filtered.groupby('PULocationID')['passenger_count'].mean().reset_index()
average_passenger_count_by_zone.columns = ['LocationID', 'average_passenger_count']
zones_with_trips = zones_with_trips.merge(average_passenger_count_by_zone, on='LocationID', how='left')
print("Zones GeoDataFrame with average passenger count (first 5):")
display(zones_with_trips.head())
```

Zones GeoDataFrame with average passenger count (first 5):

	OBJECTID	Shape_Leng	Shape_Area	zone	LocationID	borough	geometry	number_of_trips	average_passenger_count
0	1	0.116357	0.000782	Newark Airport	1	EWR	POLYGON ((933100.918 192536.086, 933091.011 19...	37.0	1.837838
1	2	0.433470	0.004866	Jamaica Bay	2	Queens	MULTIPOLYGON (((1033269.244 172126.008, 103343...	0.0	NaN
2	3	0.084341	0.000314	Allerton/Pelham Gardens	3	Bronx	POLYGON ((1026308.77 256767.698, 1026495.593 2...	1.0	1.000000
3	4	0.043567	0.000112	Alphabet City	4	Manhattan	POLYGON ((992073.467 203714.076, 992068.667 20...	1385.0	1.447653
4	5	0.092146	0.000498	Arden Heights	5	Staten Island	POLYGON ((935843.31 144283.336, 936046.565 144...	0.0	NaN

The output above reveals that most of the taxi rides are solo in certain zones. However, other zones such as airports have a slightly higher passenger average, while on the other end, some zones record few to no trips at all.

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

1) Create a list (“surcharger_cols”) that contains all the surcharge columns that exist in the DataFrame. Here, surcharge means additional fees. The columns within the dataset are: “extra”, “mta_tax”, “tolls_amount”, “improvement_surcharge”, “congestion_surcharge” and “airport_fee”.
2) Loop through each surcharge column using a Boolean series and marking True when there are trips that have a surcharge greater than 0. The *.sum()* function will count the total number of “True” values present.
3) Then divide the result in the previous step by “total_trips” so as to get the proportion of trips where the surcharge was applied. Finally, I have printed the result of the operation done.

```python
# How often is each surcharge applied?
surcharge_cols = ['extra', 'mta_tax', 'tolls_amount', 'improvement_surcharge', 'congestion_surcharge', 'airport_fee']
print("Frequency of Surcharges Being Applied (Number of Trips with Surcharge > 0):")
for col in surcharge_cols:
    surcharge_count = (df1_filtered[col] > 0).sum()
    print(f"{col}: {surcharge_count} trips")
print("\nProportion of Trips with Surcharges Applied:")
total_trips = len(df1_filtered)
for col in surcharge_cols:
    surcharge_proportion = (df1_filtered[col] > 0).sum() / total_trips
    print(f"{col}: {surcharge_proportion:.2%}")
```

```
Frequency of Surcharges Being Applied (Number of Trips with Surcharge > 0)
extra: 875524 trips
mta_tax: 1383578 trips
tolls_amount: 113245 trips
improvement_surcharge: 1391614 trips
congestion_surcharge: 1320147 trips
airport_fee: 114997 trips

Proportion of Trips with Surcharges Applied:
extra: 62.91%
mta_tax: 99.42%
tolls_amount: 8.14%
improvement_surcharge: 100.00%
congestion_surcharge: 94.86%
airport_fee: 8.26%
```

From the output above, we see that the frequency of surcharges that were applied were on a number of trips that have happened. Based on the proportion of trips, we see that “extra”, “mta_tax”, “improvement_surcharge” and “congestion_charge” have been applied on many of the trips, since they have a very high percentage compared to the others.

Improvement surcharges and MTA Tax are almost applied to most trips, suggesting that they are standard surcharges being applied. There are also surcharges that are dependent on location of the trip and route taken, such as “tolls_amount” and “airport_fee”. The high proportion of “congestion_charge” also suggests that more rides occur within different zones that have congestion.

4. Conclusions

4.1. Final Insights and Recommendations

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

- A recommendation would be to focus more resources during the evening peak hour (after 5pm) since that is when the highest number of trips generally occur. This can be done by deploying more taxi drivers and reducing waiting time to improve passenger satisfaction and trip completion.
- Another recommendation would be to use real-time data to route drivers away from areas with lots of congestion. This helps in

reducing delays in the journey and lets passengers reach their destination on time.

- Similarly, a dynamic repositioning system can be set up to shift idle taxis around based on the demand per zone, every 30 minutes. This will help passengers in getting a taxi as soon as possible, and increase the revenue the idle taxi drivers earn.
- With zones that have a high pickup-dropoff ratio, drivers can be pre-positioned at a place such that waiting time for passengers can be cut down significantly and also helps to reduce the number of empty trips by taxi drivers around the area.
- Adjusting driver schedules, since they are dispatched by vendors, based on trends shown on both weekdays and weekends. Since morning and evenings on weekdays show peaks, and similarly in the afternoon of weekends, more taxi drivers can be dispatched based on this to improve efficiency.

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.

- A suggestion would be to have more taxis stationed at the top pickup zones before peak hours start. Since these zones are usually active in the evenings and on weekends, they are ideal spots for more taxi rides to pick up for drivers.
- Add more taxis at certain zones, such as zone 79, with high activities late in the night. This can help boost revenue for taxi drivers from such trips.
- Create a specific group of taxis for flight arrivals and rush periods such as holiday seasons. Based on the data, airport-related trips tend to carry more passengers and have higher average fees. Hence this will help cover routes with high value.
- As 11% of the entire revenue is made from night trips even with the smaller share of total rides, it would be wise to keep a good number of taxis running across zones with more livelihood.
- The pickup-dropoff ratios can be used to balance the flow of taxis by keeping more in pickup areas and directing returning drivers to zones with high dropoffs. This helps to reduce the driving time between rides.
- Planning for trips during seasonal demand is also helpful in making the best use of taxis. The data shows busier times of the year in May, June and October. Hence the number of taxis deployed during these months and optimising routes can decrease idle time for taxi drivers.

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

- A small surge can be introduced during high-demand hours (between 4pm and 8pm) so that it increases the revenue for drivers but at the same time does not cause passenger churn.
- Offering two fare tiers that are similar to the difference between vendors 1 and 2. One of the tiers can be a budget-friendly tier so that passengers would still want to board the taxi, while the other can be a premium option to justify higher rates and include some perks such as better service or faster pickups.
- Instead of having fixed taxi fares, time-based fares can be used where pricing will be raised slightly by $2 or $3 during peak hours while keeping the original rates or adding discounts for off-peak hours. This will help improve utilisation without leaving different passengers out and also increases the profits a taxi driver receives at the end of the day.
- Encouraging shared rides during off-peak hours can help underused taxis as the fares per passenger will drop as the size of the group increases. This can help keep trips frequent and affordable at those times.