EDA_Optimising_NYC_Taxis_SUBHASISH-BISWAS

February 27, 2025

1 New York City Yellow Taxi Data

1.1 Objective

In this case study you will be learning exploratory data analysis (EDA) with the help of a dataset on yellow taxi rides in New York City. This will enable you to understand why EDA is an important step in the process of data science and machine learning.

1.2 Problem Statement

As an analyst at an upcoming taxi operation in NYC, you are tasked to use the 2023 taxi trip data to uncover insights that could help optimise taxi operations. The goal is to analyse patterns in the data that can inform strategic decisions to improve service efficiency, maximise revenue, and enhance passenger experience.

1.3 Tasks

You need to perform the following steps for successfully completing this assignment: 1. Data Loading 2. Data Cleaning 3. Exploratory Analysis: Bivariate and Multivariate 4. Creating Visualisations to Support the Analysis 5. Deriving Insights and Stating Conclusions

---

NOTE: The marks given along with headings and sub-headings are cumulative marks for those particular headings/sub-headings.

The actual marks for each task are specified within the tasks themselves.

For example, marks given with heading *2* or sub-heading *2.1* are the cumulative marks, for your reference only.

The marks you will receive for completing tasks are given with the tasks.

Suppose the marks for two tasks are: 3 marks for 2.1.1 and 2 marks for 3.2.2, or * 2.1.1 [3 marks] * 3.2.2 [2 marks]

then, you will earn 3 marks for completing task 2.1.1 and 2 marks for completing task 3.2.2.

1.4 Data Understanding

The yellow taxi trip records include fields capturing pick-up and drop-off dates/times, pick-up and drop-off locations, trip distances, itemized fares, rate types, payment types, and driver-reported passenger counts.

The data is stored in Parquet format (*.parquet*). The dataset is from 2009 to 2024. However, for this assignment, we will only be using the data from 2023.

The data for each month is present in a different parquet file. You will get twelve files for each of the months in 2023.

The data was collected and provided to the NYC Taxi and Limousine Commission (TLC) by technology providers like vendors and taxi hailing apps.

You can find the link to the TLC trip records page here: https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page

1.4.1 Data Description

You can find the data description here: Data Dictionary

Trip Records

Field Name	description
VendorID	A code indicating the TPEP provider that provided the record. 1= Creative Mobile Technologies, LLC; 2= VeriFone Inc.
tpep_pickup_datetime	The date and time when the meter was engaged.
tpep_dropoff_datetime	The date and time when the meter was disengaged.
Passenger_count	The number of passengers in the vehicle. This is a driver-entered value.
Trip_distance	The elapsed trip distance in miles reported by the taximeter.
PULocationID	TLC Taxi Zone in which the taximeter was engaged
DOLocationID	TLC Taxi Zone in which the taximeter was disengaged
RateCodeID	The final rate code in effect at the end of the trip. 1 = Standard rate 2 = JFK 3 = Newark 4 = Nassau or Westchester 5 = Negotiated fare 6 = Group ride

Field Name	description
Store_and_fwd_flag	This flag indicates whether the trip record was held in vehicle memory before sending to the vendor, aka "store and forward," because the vehicle did not have a connection to the server. Y= store and forward trip N= not a store and forward trip
Payment_type	A numeric code signifying how the passenger paid for the trip. 1 = Credit card 2 = Cash 3 = No charge 4 = Dispute 5 = Unknown 6 = Voided trip
Fare_amount	The time-and-distance fare calculated by the meter. Extra Miscellaneous extras and surcharges. Currently, this only includes the 0.50 and 1 USD rush hour and overnight charges.
MTA_tax	0.50 USD MTA tax that is automatically triggered based on the metered rate in use.
Improvement_surcharge	0.30 USD improvement surcharge assessed trips at the flag drop. The improvement surcharge began being levied in 2015.
Tip_amount	Tip amount – This field is automatically populated for credit card tips. Cash tips are not included.
Tolls_amount	Total amount of all tolls paid in trip.
total_amount	The total amount charged to passengers. Does not include cash tips.
Congestion_Surcharge	Total amount collected in trip for NYS congestion surcharge.
Airport_fee	1.25 USD for pick up only at LaGuardia and John F. Kennedy Airports

Although the amounts of extra charges and taxes applied are specified in the data dictionary, you will see that some cases have different values of these charges in the actual data.

Taxi Zones

Each of the trip records contains a field corresponding to the location of the pickup or drop-off of the trip, populated by numbers ranging from 1-263.

These numbers correspond to taxi zones, which may be downloaded as a table or map/shapefile and matched to the trip records using a join.

This is covered in more detail in later sections.

1.5 1 Data Preparation

[5 marks]

1.5.1 Import Libraries

[1]:
```
# Import warnings
```

[2]:
```
# Import the libraries you will be using for analysis
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import tabulate
import os
```

[3]:
```
# Recommended versions
# numpy version: 1.26.4
# pandas version: 2.2.2
# matplotlib version: 3.10.0
# seaborn version: 0.13.2

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

```
numpy version: 1.26.4
pandas version: 2.2.3
matplotlib version: 3.10.0
seaborn version: 0.13.2
```

1.5.2 1.1 Load the dataset

[5 marks]

You will see twelve files, one for each month.

To read parquet files with Pandas, you have to follow a similar syntax as that for CSV files.

`df = pd.read_parquet('file.parquet')`

[4]:
```
# Try loading one file

# df = pd.read_parquet('2023-1.parquet')
# df.info()
```

How many rows are there? 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?

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.

Sampling the Data

> One way is to take a small percentage of entries for pickup in every hour of a date. So, for all the days in a month, we can iterate through the hours and select 5% values randomly from those. Use `tpep_pickup_datetime` for this. Separate date and hour from the datetime values and then for each date, select some fraction of trips for each of the 24 hours.

To sample data, you can use the `sample()` method. Follow this syntax:

```
# sampled_data is an empty DF to keep appending sampled data of each hour
# hour_data is the DF of entries for an hour 'X' on a date 'Y'

sample = hour_data.sample(frac = 0.05, random_state = 42)
# sample 0.05 of the hour_data
# random_state is just a seed for sampling, you can define it yourself

sampled_data = pd.concat([sampled_data, sample]) # adding data for this hour to the DF
```

This *sampled_data* will contain 5% values selected at random from each hour.

Note that the code given above is only the part that will be used for sampling and not the complete code required for sampling and combining the data files.

Keep in mind that you sample by date AND hour, not just hour. (Why?)

1.1.1 [5 marks] Figure out how to sample and combine the files.

Note: It is not mandatory to use the method specified above. While sampling, you only need to make sure that your sampled data represents the overall data of all the months accurately.

[5]:
```
# Sample the data
# It is recommmended to not load all the files at once to avoid memory overload
```

[6]:
```
# from google.colab import drive
# drive.mount('/content/drive')
```

[7]:
```
import os
# Get the base directory (current working directory)
base_dir = '/Users/subhasishbiswas/GIT/Interstellar/UpGrad/Code/Courses/C1-SQL␣
↪and Statistics Essentials/M7-NYC Taxi Records Analysis/SUBHASISH BISWAS/EDA␣
↪NYC Taxi/' #os.getcwd()
```

```
# Append the required path
trip_records_path = os.path.join(base_dir, "Datasets and Dictionary",␣
↪"trip_records")

print(trip_records_path)
```

/Users/subhasishbiswas/GIT/Interstellar/UpGrad/Code/Courses/C1-SQL and Statistics Essentials/M7-NYC Taxi Records Analysis/SUBHASISH BISWAS/EDA NYC Taxi/Datasets and Dictionary/trip_records

[8]:

```
# Select the folder having data files

os.chdir(trip_records_path)
# Create a list of all the twelve files to read

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

file_list = os.listdir()
print(file_list)
```

['2023-12.parquet', '2023-6.parquet', '2023-7.parquet', '.DS_Store', '2023-5.parquet', '2023-11.parquet', '2023-10.parquet', '2023-4.parquet', '2023-1.parquet', '2023-8.parquet', '2023-9.parquet', '2023-2.parquet', '2023-3.parquet']

[9]:

```
# 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




# 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(os.getcwd(), file_name)
        print(f"Reading file: {file_name}")
        # Reading the current file

        # 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 = pd.DataFrame()
```

```
        df_month = pd.read_parquet(file_path)
        df_month['date'] = df_month['tpep_pickup_datetime'].dt.date
        df_month['hour'] = df_month['tpep_pickup_datetime'].dt.hour

        # Loop through dates and then loop through every hour of each date
        # Sample 5% of the hourly data randomly
        # add data of this hour to the dataframe
        for date in df_month['date'].unique():
            for hour in range(24):
                # Filter data for the current date and hour
                hour_data = df_month[(df_month['date'] == date) &␣
↪(df_month['hour'] == hour)]
                # Sample 5% of the hourly data randomly
                if len(hour_data) > 0:
                    sample = hour_data.sample(frac=0.05, random_state=42)
                    sampled_data = pd.concat([sampled_data, sample])

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

    except Exception as e:
        print(f"Error reading file {file_name}: {e}")

# Store the df in csv/parquet
df.to_parquet('Sampled_NYC_Taxi_Data.parquet')
df
```

```
Reading file: 2023-12.parquet
Reading file: 2023-6.parquet
Reading file: 2023-7.parquet
Reading file: .DS_Store
Error reading file .DS_Store: Could not open Parquet input source '<Buffer>':
Parquet magic bytes not found in footer. Either the file is corrupted or this is
not a parquet file.
Reading file: 2023-5.parquet
Reading file: 2023-11.parquet
Reading file: 2023-10.parquet
Reading file: 2023-4.parquet
Reading file: 2023-1.parquet
Reading file: 2023-8.parquet
Reading file: 2023-9.parquet
Reading file: 2023-2.parquet
Reading file: 2023-3.parquet
```

[9]:
```
         VendorID tpep_pickup_datetime tpep_dropoff_datetime  passenger_count  \
1788            2  2023-12-01 00:27:51   2023-12-01 00:50:12              1.0
3196699         2  2023-12-01 00:38:48   2023-12-01 01:01:55              NaN
```

```
1408           2  2023-12-01 00:06:19   2023-12-01 00:16:57              1.0
3196663        2  2023-12-01 00:00:50   2023-12-01 00:14:37              NaN
3613           2  2023-12-01 00:16:07   2023-12-01 00:19:17              1.0
…          …                …                    …              …
3203004        2  2023-06-30 23:53:10   2023-07-01 00:05:55              1.0
3203122        1  2023-06-30 23:22:42   2023-06-30 23:39:06              1.0
3206515        1  2023-06-30 23:50:42   2023-07-01 00:20:00              2.0
3206491        1  2023-06-30 23:05:31   2023-06-30 23:15:52              1.0
3202916        2  2023-07-01 00:00:51   2023-07-01 00:24:19              1.0

         trip_distance  RatecodeID store_and_fwd_flag  PULocationID  \
1788              3.99         1.0                  N           148
3196699           4.79         NaN               None           231
1408              1.05         1.0                  N           161
3196663           2.08         NaN               None           137
3613              0.40         1.0                  N            68
…                  …         …                 …           …
3203004           2.63         1.0                  N           170
3203122           0.00        99.0                  N            90
3206515           5.40         1.0                  N            87
3206491           1.00         1.0                  N            87
3202916           5.04         1.0                  N           209

         DOLocationID  payment_type  …  mta_tax  tip_amount  tolls_amount  \
1788               50             1  …      0.5        5.66           0.0
3196699            61             0  …      0.5        3.00           0.0
1408              161             1  …      0.5        3.14           0.0
3196663           144             0  …      0.5        0.00           0.0
3613               68             1  …      0.5        0.00           0.0
…                 …            …  …      …        …           …
3203004           143             1  …      0.5        4.80           0.0
3203122           232             1  …      0.5        0.00           0.0
3206515           161             1  …      0.5        2.00           0.0
3206491           231             2  …      0.5        0.00           0.0
3202916           225             1  …      0.5        4.56           0.0

         improvement_surcharge  total_amount  congestion_surcharge  \
1788                       1.0         33.96                   2.5
3196699                    1.0         29.43                   NaN
1408                       1.0         18.84                   2.5
3196663                    1.0         21.22                   NaN
3613                       1.0         10.10                   2.5
…                        …           …                   …
3203004                    1.0         24.00                   2.5
3203122                    1.0         19.70                   0.0
3206515                    1.0         39.40                   2.5
3206491                    1.0         15.70                   2.5
```

```
3202916                    1.0         34.96                   2.5

         Airport_fee        date  hour airport_fee
1788             0.0  2023-12-01     0         NaN
3196699          NaN  2023-12-01     0         NaN
1408             0.0  2023-12-01     0         NaN
3196663          NaN  2023-12-01     0         NaN
3613             0.0  2023-12-01     0         NaN
…               …          …   …          …
3203004          0.0  2023-06-30    23         NaN
3203122          0.0  2023-06-30    23         NaN
3206515          0.0  2023-06-30    23         NaN
3206491          0.0  2023-06-30    23         NaN
3202916          0.0  2023-07-01     0         NaN

[1896400 rows x 22 columns]
```

After combining the data files into one DataFrame, convert the new DataFrame to a CSV or parquet file and store it to use directly.

Ideally, you can try keeping the total entries to around 250,000 to 300,000.

1.6 2 Data Cleaning

[30 marks]

Now we can load the new data directly.

[10]:
```python
# Load the new data file

try:
    df = pd.read_parquet('Sampled_NYC_Taxi_Data.parquet')
except FileNotFoundError:
    print("Error: 'Sampled_NYC_Taxi_Data.parquet' DataFrame not found or saved␣
 ↪file not found. Please make sure you have sampled and saved the data first.")
print(df.count().sum())
```

```
39500030
```

[11]:
```python
df.head()
```

[11]:
```
         VendorID tpep_pickup_datetime tpep_dropoff_datetime  passenger_count  \
1788            2  2023-12-01 00:27:51   2023-12-01 00:50:12              1.0
3196699         2  2023-12-01 00:38:48   2023-12-01 01:01:55              NaN
1408            2  2023-12-01 00:06:19   2023-12-01 00:16:57              1.0
3196663         2  2023-12-01 00:00:50   2023-12-01 00:14:37              NaN
3613            2  2023-12-01 00:16:07   2023-12-01 00:19:17              1.0

         trip_distance  RatecodeID store_and_fwd_flag  PULocationID  \
```

```
1788             3.99        1.0                 N          148
3196699          4.79        NaN              None          231
1408             1.05        1.0                 N          161
3196663          2.08        NaN              None          137
3613             0.40        1.0                 N           68

         DOLocationID  payment_type  …  mta_tax  tip_amount  tolls_amount  \
1788               50             1  …      0.5        5.66           0.0
3196699            61             0  …      0.5        3.00           0.0
1408              161             1  …      0.5        3.14           0.0
3196663           144             0  …      0.5        0.00           0.0
3613               68             1  …      0.5        0.00           0.0

         improvement_surcharge  total_amount  congestion_surcharge  \
1788                       1.0         33.96                   2.5
3196699                    1.0         29.43                   NaN
1408                       1.0         18.84                   2.5
3196663                    1.0         21.22                   NaN
3613                       1.0         10.10                   2.5

         Airport_fee        date  hour airport_fee
1788             0.0  2023-12-01     0         NaN
3196699          NaN  2023-12-01     0         NaN
1408             0.0  2023-12-01     0         NaN
3196663          NaN  2023-12-01     0         NaN
3613             0.0  2023-12-01     0         NaN

[5 rows x 22 columns]
```

[12]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 1896400 entries, 1788 to 3202916
Data columns (total 22 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
 19  date                   object
 20  hour                   int32
 21  airport_fee            float64
dtypes: datetime64[us](2), float64(13), int32(1), int64(4), object(2)
memory usage: 325.5+ MB
```

2.1 Fixing Columns [10 marks]

Fix/drop any columns as you seem necessary in the below sections

2.1.1 [2 marks]

Fix the index and drop unnecessary columns

[13]:
```
# Reset the index
df = df.reset_index(drop=True)

'''
I'm dropping the columns VendorID, store_and_fwd_flag, payment_ type,
tpep_pickup_datetime, and tpep_dropoff _datetime because they are not directly␣
↪relevant to
the analysis and can be dropped. The goal of the analysis is to uncover␣
↪insights that could help
optimize taxi operations, and these columns do not provide any direct␣
↪information about taxi
operations. For example, the Vendor ID column indicates the provider that␣
↪provided the record,
which is not relevant to the analysis. Similarly, the store_and_fwd_flag column␣
↪indicates
whether the trip record was held in vehicle memory before sending to the␣
↪vendor, which is also
not relevant to the analysis.
'''
# Drop unnecessary columns
df = df.drop(columns=['store_and_fwd_flag'])

df.describe()
```

[13]:
```
           VendorID        tpep_pickup_datetime       tpep_dropoff_datetime  \
count  1.896400e+06                     1896400                     1896400
mean   1.733026e+00  2023-07-02 19:59:52.930795  2023-07-02 20:17:18.919564
```

```
min    1.000000e+00         2022-12-31 23:51:30         2022-12-31 23:56:06
25%    1.000000e+00  2023-04-02 16:10:08.750000  2023-04-02 16:27:43.500000
50%    2.000000e+00  2023-06-27 15:44:22.500000         2023-06-27 16:01:15
75%    2.000000e+00         2023-10-06 19:37:45         2023-10-06 19:53:39
max    6.000000e+00         2023-12-31 23:57:51         2024-01-01 20:50:55
std    4.476401e-01                         NaN                         NaN

       passenger_count  trip_distance    RatecodeID  PULocationID  \
count     1.831526e+06   1.896400e+06  1.831526e+06  1.896400e+06
mean      1.369215e+00   3.858293e+00  1.634694e+00  1.652814e+02
min       0.000000e+00   0.000000e+00  1.000000e+00  1.000000e+00
25%       1.000000e+00   1.050000e+00  1.000000e+00  1.320000e+02
50%       1.000000e+00   1.790000e+00  1.000000e+00  1.620000e+02
75%       1.000000e+00   3.400000e+00  1.000000e+00  2.340000e+02
max       9.000000e+00   1.263605e+05  9.900000e+01  2.650000e+02
std       8.927560e-01   1.294085e+02  7.393915e+00  6.400038e+01

       DOLocationID  payment_type   fare_amount         extra       mta_tax  \
count  1.896400e+06  1.896400e+06  1.896400e+06  1.896400e+06  1.896400e+06
mean   1.640515e+02  1.163817e+00  1.991935e+01  1.588018e+00  4.952796e-01
min    1.000000e+00  0.000000e+00  0.000000e+00 -2.500000e+00 -5.000000e-01
25%    1.140000e+02  1.000000e+00  9.300000e+00  0.000000e+00  5.000000e-01
50%    1.620000e+02  1.000000e+00  1.350000e+01  1.000000e+00  5.000000e-01
75%    2.340000e+02  1.000000e+00  2.190000e+01  2.500000e+00  5.000000e-01
max    2.650000e+02  4.000000e+00  1.431635e+05  2.080000e+01  4.000000e+00
std    6.980207e+01  5.081384e-01  1.055371e+02  1.829200e+00  4.885128e-02

         tip_amount  tolls_amount  improvement_surcharge  total_amount  \
count  1.896400e+06  1.896400e+06           1.896400e+06  1.896400e+06
mean   3.547011e+00  5.965338e-01           9.989706e-01  2.898186e+01
min    0.000000e+00  0.000000e+00          -1.000000e+00 -5.750000e+00
25%    1.000000e+00  0.000000e+00           1.000000e+00  1.596000e+01
50%    2.850000e+00  0.000000e+00           1.000000e+00  2.100000e+01
75%    4.420000e+00  0.000000e+00           1.000000e+00  3.094000e+01
max    2.230800e+02  1.430000e+02           1.000000e+00  1.431675e+05
std    4.054882e+00  2.187878e+00           3.112072e-02  1.064162e+02

       congestion_surcharge   Airport_fee          hour    airport_fee
count          1.831526e+06  1.683043e+06  1.896400e+06  148483.000000
mean           2.307524e+00  1.458850e-01  1.426504e+01       0.109036
min           -2.500000e+00 -1.750000e+00  0.000000e+00      -1.250000
25%            2.500000e+00  0.000000e+00  1.100000e+01       0.000000
50%            2.500000e+00  0.000000e+00  1.500000e+01       0.000000
75%            2.500000e+00  0.000000e+00  1.900000e+01       0.000000
max            2.500000e+00  1.750000e+00  2.300000e+01       1.250000
std            6.667267e-01  4.733757e-01  5.807381e+00       0.352744
```

2.1.2 [3 marks] There are two airport fee columns. This is possibly an error in naming columns.

Let's see whether these can be combined into a single column.

[14]:
```
# Combine the two airport fee columns

print(df[['airport_fee', 'Airport_fee']].head())
# Rename the columns
df.rename(columns={'airport_fee': 'airport_fee1', 'Airport_fee':␣
↪'airport_fee2'}, inplace=True)

# Fill null values with 0
df['airport_fee1'] = df['airport_fee1'].fillna(0)
df['airport_fee2'] = df['airport_fee2'].fillna(0)

# Combine the two columns
df['airport_fee'] = df['airport_fee1'] + df['airport_fee2']

# Drop the original columns
df = df.drop(columns=['airport_fee1', 'airport_fee2'])

# Save the updated DataFrame
df.to_csv('1_Cleaned_Sampled_NYC_Taxi_Data.csv', index=False)
```

```
   airport_fee  Airport_fee
0          NaN          0.0
1          NaN          NaN
2          NaN          0.0
3          NaN          NaN
4          NaN          0.0
```

2.1.4 [5 marks] Fix columns with negative (monetary) values

[15]:
```
# check where values of fare amount are negative
# Filter the DataFrame to show only rows where `fare_amount` is negative
#negative_fare_amount = df[df['fare_amount'] < 0]
#num_negative_fares = len(negative_fare_amount)  # Get the count of rows
#print(f"Number of negative fare_amount values: {num_negative_fares}")

# 2. Remove negative values from specified columns
columns_to_check = ['fare_amount', 'tip_amount', 'total_amount',␣
↪'trip_distance']

for col in columns_to_check:
    # Count negative values before removal
    num_negatives_before = (df[col] < 0).sum()

    # Remove negative values
    df = df[df[col] >= 0]
```

```
    # Count negative values after removal (should be 0)
    num_negatives_after = (df[col] < 0).sum()

    print(f"\nColumn '{col}':")
    print(f"  - Number of negative values before removal:␣
↪{num_negatives_before}")
    print(f"  - Number of negative values after removal: {num_negatives_after}")

df.to_csv("2_Cleaned_Sampled_NYC_Taxi_Data.csv", index=False)
print("Cleaned data saved to '2_Cleaned_Sampled_NYC_Taxi_Data.csv'")
```

```
Column 'fare_amount':
  - Number of negative values before removal: 0
  - Number of negative values after removal: 0

Column 'tip_amount':
  - Number of negative values before removal: 0
  - Number of negative values after removal: 0

Column 'total_amount':
  - Number of negative values before removal: 78
  - Number of negative values after removal: 0

Column 'trip_distance':
  - Number of negative values before removal: 0
  - Number of negative values after removal: 0
Cleaned data saved to '2_Cleaned_Sampled_NYC_Taxi_Data.csv'
```

[16]:
```
# Analyse the above parameters
columns_to_check = ['fare_amount', 'tip_amount', 'total_amount',␣
↪'trip_distance']

for col in columns_to_check:
    num_zeros = (df[col] == 0).sum()
    num_negatives = (df[col] < 0).sum()
    print(f"\nColumn '{col}':")
    print(f"  - Number of zero values: {num_zeros}")
    print(f"  - Number of negative values: {num_negatives}")
```

```
Column 'fare_amount':
  - Number of zero values: 573
  - Number of negative values: 0

Column 'tip_amount':
  - Number of zero values: 435880
  - Number of negative values: 0
```

```
Column 'total_amount':
  - Number of zero values: 310
  - Number of negative values: 0

Column 'trip_distance':
  - Number of zero values: 37712
  - Number of negative values: 0
```

Did you notice something different in the `RatecodeID` column for above records?

[17]:
```python
# Analyse RatecodeID for the negative fare amounts
'''
Looking at the data dictionary, the RateCodeID column has values ranging from 1␣
 ↪to 6,
with each number representing a specific rate type.

However, in the records where fare_amount
is negative, there are instances of RateCodeID being 99, which is not a defined␣
 ↪code in the data
dictionary.

This discrepancy suggests that there might be errors or inconsistencies in the␣
 ↪data, specifically
related to the RateCodeID column. It's possible that the code 99 was used to␣
 ↪represent a special
type of fare or that it was an error during data entry.
'''

try:
    df = pd.read_csv('2_Cleaned_Sampled_NYC_Taxi_Data.csv')
except FileNotFoundError:
    print("Error: 'Sampled_NYC_Taxi_Data.parquet' DataFrame not found or saved␣
 ↪file not found. Please make sure you have sampled and saved the data first.")
print(df.count().sum())

# Count the frequency of each unique value in `RateCodeID`
ratecode_counts = df['RatecodeID'].value_counts()
# Display the counts
print(ratecode_counts.to_markdown(numalign="left", stralign="left"))

# Display rows with negative `fare_amount` and `RateCodeID` other than 99
other_ratecodes = df[df['RatecodeID']!= 99]
other_ratecodes.head()
```

```
37731818
| RatecodeID   | count       |
|:-------------|:------------|
```

```
| 1             | 1.72921e+06 |
| 2             | 71646       |
| 99            | 10472       |
| 5             | 10272       |
| 3             | 6123        |
| 4             | 3722        |
| 6             | 3           |
```

[17]:
```
   VendorID tpep_pickup_datetime tpep_dropoff_datetime  passenger_count  \
0         2  2023-12-01 00:27:51   2023-12-01 00:50:12              1.0
1         2  2023-12-01 00:38:48   2023-12-01 01:01:55              NaN
2         2  2023-12-01 00:06:19   2023-12-01 00:16:57              1.0
3         2  2023-12-01 00:00:50   2023-12-01 00:14:37              NaN
4         2  2023-12-01 00:16:07   2023-12-01 00:19:17              1.0

   trip_distance  RatecodeID  PULocationID  DOLocationID  payment_type  \
0           3.99         1.0           148            50             1
1           4.79         NaN           231            61             0
2           1.05         1.0           161           161             1
3           2.08         NaN           137           144             0
4           0.40         1.0            68            68             1

   fare_amount  extra  mta_tax  tip_amount  tolls_amount  \
0        23.30    1.0      0.5        5.66           0.0
1        22.43    0.0      0.5        3.00           0.0
2        10.70    1.0      0.5        3.14           0.0
3        17.22    0.0      0.5        0.00           0.0
4         5.10    1.0      0.5        0.00           0.0

   improvement_surcharge  total_amount  congestion_surcharge        date  \
0                    1.0         33.96                   2.5  2023-12-01
1                    1.0         29.43                   NaN  2023-12-01
2                    1.0         18.84                   2.5  2023-12-01
3                    1.0         21.22                   NaN  2023-12-01
4                    1.0         10.10                   2.5  2023-12-01

   hour  airport_fee
0     0          0.0
1     0          0.0
2     0          0.0
3     0          0.0
4     0          0.0
```

[18]:
```python
# Find which columns have negative values
for col in df.columns:
    if pd.api.types.is_numeric_dtype(df[col]):
        if (df[col] < 0).any():
```

```
            print(f"Column '{col}' has {len(df[df[col] < 0])} negative values")
```

```
Column 'extra' has 1 negative values
```

[19]:
```
# fix these negative values

# Convert negative values to positive values
for col in df.columns:
    if pd.api.types.is_numeric_dtype(df[col]):
        df[col] = df[col].abs()

# Save the updated DataFrame
df.to_csv('2_Cleaned_Sampled_NYC_Taxi_Data.csv', index=False)
```

[20]:
```
try:
   df=pd.read_csv('2_Cleaned_Sampled_NYC_Taxi_Data.csv')
except FileNotFoundError:
    print("Error: DataFrame not found or saved file not found. Please make sure␣
↪you have sampled and saved the data first.")
```

1.6.1 2.2 Handling Missing Values

[10 marks]

2.2.1 [2 marks] Find the proportion of missing values in each column

[21]:
```
# Find the proportion of missing values in each column
missing_prop = df.isnull().mean()
missing_prop
```

[21]:
```
VendorID                 0.00000
tpep_pickup_datetime     0.00000
tpep_dropoff_datetime    0.00000
passenger_count          0.03421
trip_distance            0.00000
RatecodeID               0.03421
PULocationID             0.00000
DOLocationID             0.00000
payment_type             0.00000
fare_amount              0.00000
extra                    0.00000
mta_tax                  0.00000
tip_amount               0.00000
tolls_amount             0.00000
improvement_surcharge    0.00000
total_amount             0.00000
congestion_surcharge     0.03421
date                     0.00000
hour                     0.00000
```

```
airport_fee              0.00000
dtype: float64
```

2.2.2 [3 marks] Handling missing values in `passenger_count`

[22]:
```python
# Display the rows with null values
null_rows = df[df.isnull().any(axis=1)]
null_rows
```

[22]:
```
         VendorID tpep_pickup_datetime tpep_dropoff_datetime  passenger_count  \
1               2  2023-12-01 00:38:48   2023-12-01 01:01:55              NaN
3               2  2023-12-01 00:00:50   2023-12-01 00:14:37              NaN
27              2  2023-12-01 00:01:11   2023-12-01 00:15:53              NaN
122             2  2023-12-01 00:02:18   2023-12-01 00:12:25              NaN
127             1  2023-12-01 00:04:14   2023-12-01 00:25:16              NaN
…             …                    …                     …              …
1896215         1  2023-06-30 23:14:07   2023-06-30 23:25:45              NaN
1896231         2  2023-06-30 23:40:46   2023-07-01 00:04:37              NaN
1896274         2  2023-06-30 23:57:33   2023-07-01 00:09:15              NaN
1896295         2  2023-06-30 23:36:40   2023-06-30 23:53:20              NaN
1896305         1  2023-06-30 23:34:22   2023-07-01 00:32:59              NaN

         trip_distance  RatecodeID  PULocationID  DOLocationID  payment_type  \
1                 4.79         NaN           231            61             0
3                 2.08         NaN           137           144             0
27                3.49         NaN           164           262             0
122               1.79         NaN           142           239             0
127               0.00         NaN           186            74             0
…                  …          …            …            …            …
1896215           0.70         NaN           230           186             0
1896231           4.46         NaN           143            79             0
1896274           2.75         NaN           166           142             0
1896295           5.18         NaN           148           237             0
1896305          20.20         NaN           132            74             0

         fare_amount  extra  mta_tax  tip_amount  tolls_amount  \
1              22.43   0.00      0.5        3.00          0.00
3              17.22   0.00      0.5        0.00          0.00
27             17.83   0.00      0.5        0.00          0.00
122             9.88   0.00      0.5        0.00          0.00
127            30.31   0.00      0.5        0.00          0.00
…                …    …      …         …            …
1896215        11.40   1.00      0.5        2.46          0.00
1896231        23.26   0.00      0.5        0.00          0.00
1896274        16.14   0.00      0.5        0.00          0.00
1896295        26.09   0.00      0.5        3.01          0.00
1896305        70.00   1.75      0.5       11.97          6.55
```

```
         improvement_surcharge  total_amount  congestion_surcharge  \
1                          1.0         29.43                   NaN
3                          1.0         21.22                   NaN
27                         1.0         21.83                   NaN
122                        1.0         13.88                   NaN
127                        1.0         34.31                   NaN
…                          …           …                     …
1896215                    1.0         18.86                   NaN
1896231                    1.0         27.26                   NaN
1896274                    1.0         20.14                   NaN
1896295                    1.0         33.10                   NaN
1896305                    1.0         91.77                   NaN

               date  hour  airport_fee
1        2023-12-01     0          0.0
3        2023-12-01     0          0.0
27       2023-12-01     0          0.0
122      2023-12-01     0          0.0
127      2023-12-01     0          0.0
…               …   …          …
1896215  2023-06-30    23          0.0
1896231  2023-06-30    23          0.0
1896274  2023-06-30    23          0.0
1896295  2023-06-30    23          0.0
1896305  2023-06-30    23          0.0

[64874 rows x 20 columns]
```

[23]:

```
# Impute NaN values in 'passenger_count'
# Impute NaN values in `passenger_count` with the mean
print("Before removing passenger_count: " + str(df['passenger_count'].isnull().
 ↪sum()))
df['passenger_count'] = df['passenger_count'].fillna(df['passenger_count'].
 ↪mean())
print("After removing passenger_count: " + str(df['passenger_count'].isnull().
 ↪sum()))
```

```
Before removing passenger_count: 64874
After removing passenger_count: 0
```

[24]:

```
# Display the rows with missing values
df[df.isnull().any(axis=1)].head()
```

[24]:

```
    VendorID tpep_pickup_datetime tpep_dropoff_datetime  passenger_count  \
1          2  2023-12-01 00:38:48   2023-12-01 01:01:55         1.369209
3          2  2023-12-01 00:00:50   2023-12-01 00:14:37         1.369209
27         2  2023-12-01 00:01:11   2023-12-01 00:15:53         1.369209
```

```
122          2  2023-12-01 00:02:18   2023-12-01 00:12:25          1.369209
127          1  2023-12-01 00:04:14   2023-12-01 00:25:16          1.369209

     trip_distance  RatecodeID  PULocationID  DOLocationID  payment_type  \
1             4.79         NaN           231            61             0
3             2.08         NaN           137           144             0
27            3.49         NaN           164           262             0
122           1.79         NaN           142           239             0
127           0.00         NaN           186            74             0

     fare_amount  extra  mta_tax  tip_amount  tolls_amount  \
1          22.43    0.0      0.5         3.0           0.0
3          17.22    0.0      0.5         0.0           0.0
27         17.83    0.0      0.5         0.0           0.0
122         9.88    0.0      0.5         0.0           0.0
127        30.31    0.0      0.5         0.0           0.0

     improvement_surcharge  total_amount  congestion_surcharge        date  \
1                      1.0         29.43                   NaN  2023-12-01
3                      1.0         21.22                   NaN  2023-12-01
27                     1.0         21.83                   NaN  2023-12-01
122                    1.0         13.88                   NaN  2023-12-01
127                    1.0         34.31                   NaN  2023-12-01

     hour  airport_fee
1       0          0.0
3       0          0.0
27      0          0.0
122     0          0.0
127     0          0.0
```

Did you find zeroes in passenger_count? Handle these.

2.2.3 [2 marks] Handle missing values in `RatecodeID`

[25]:
```python
# Fix missing values in 'RatecodeID'



# Impute missing values in `RateCodeID` with the mean

# Display the count of missing values in `RatecodeID`
print("Before removing RatecodeID: " + str(df['RatecodeID'].isnull().sum()))

# Impute the missing values in `RatecodeID` with its mean
df['RatecodeID'] = df['RatecodeID'].fillna(df['RatecodeID'].mean())

# Verify the count of missing values in `RatecodeID` after imputation
print("Before removing RatecodeID: " + str(df['RatecodeID'].isnull().sum()))
```

```
df.to_csv('3_Cleaned_Sampled_NYC_Taxi_Data.csv', index=False)
```

```
Before removing RatecodeID: 64874
Before removing RatecodeID: 0
```

[26]:

```
try:
   df=pd.read_csv('3_Cleaned_Sampled_NYC_Taxi_Data.csv')
except FileNotFoundError:
    print("Error: DataFrame not found or saved file not found. Please make sure␣
 ↪you have sampled and saved the data first.")
df
```

[26]:

```
         VendorID tpep_pickup_datetime tpep_dropoff_datetime  passenger_count  \
0               2  2023-12-01 00:27:51   2023-12-01 00:50:12         1.000000
1               2  2023-12-01 00:38:48   2023-12-01 01:01:55         1.369209
2               2  2023-12-01 00:06:19   2023-12-01 00:16:57         1.000000
3               2  2023-12-01 00:00:50   2023-12-01 00:14:37         1.369209
4               2  2023-12-01 00:16:07   2023-12-01 00:19:17         1.000000
…             …                    …                     …              …
1896317         2  2023-06-30 23:53:10   2023-07-01 00:05:55         1.000000
1896318         1  2023-06-30 23:22:42   2023-06-30 23:39:06         1.000000
1896319         1  2023-06-30 23:50:42   2023-07-01 00:20:00         2.000000
1896320         1  2023-06-30 23:05:31   2023-06-30 23:15:52         1.000000
1896321         2  2023-07-01 00:00:51   2023-07-01 00:24:19         1.000000

         trip_distance  RatecodeID  PULocationID  DOLocationID  payment_type  \
0                 3.99    1.000000           148            50             1
1                 4.79    1.634698           231            61             0
2                 1.05    1.000000           161           161             1
3                 2.08    1.634698           137           144             0
4                 0.40    1.000000            68            68             1
…                  …           …             …             …             …
1896317           2.63    1.000000           170           143             1
1896318           0.00   99.000000            90           232             1
1896319           5.40    1.000000            87           161             1
1896320           1.00    1.000000            87           231             2
1896321           5.04    1.000000           209           225             1

         fare_amount  extra  mta_tax  tip_amount  tolls_amount  \
0              23.30    1.0      0.5        5.66           0.0
1              22.43    0.0      0.5        3.00           0.0
2              10.70    1.0      0.5        3.14           0.0
3              17.22    0.0      0.5        0.00           0.0
4               5.10    1.0      0.5        0.00           0.0
…                …     …       …          …             …
```

```
1896317        14.20    1.0      0.5       4.80          0.0
1896318        18.20    0.0      0.5       0.00          0.0
1896319        32.40    3.5      0.5       2.00          0.0
1896320        10.70    3.5      0.5       0.00          0.0
1896321        25.40    1.0      0.5       4.56          0.0

         improvement_surcharge  total_amount  congestion_surcharge  \
0                          1.0         33.96                   2.5
1                          1.0         29.43                   NaN
2                          1.0         18.84                   2.5
3                          1.0         21.22                   NaN
4                          1.0         10.10                   2.5
…                            …             …                     …
1896317                    1.0         24.00                   2.5
1896318                    1.0         19.70                   0.0
1896319                    1.0         39.40                   2.5
1896320                    1.0         15.70                   2.5
1896321                    1.0         34.96                   2.5

               date  hour  airport_fee
0        2023-12-01     0          0.0
1        2023-12-01     0          0.0
2        2023-12-01     0          0.0
3        2023-12-01     0          0.0
4        2023-12-01     0          0.0
…                 …   …            …
1896317  2023-06-30    23          0.0
1896318  2023-06-30    23          0.0
1896319  2023-06-30    23          0.0
1896320  2023-06-30    23          0.0
1896321  2023-07-01     0          0.0

[1896322 rows x 20 columns]
```

2.2.4 [3 marks] Impute NaN in `congestion_surcharge`

[27]:
```
# handle null values in congestion_surcharge

# Display the rows with missing values
df[df.isnull().any(axis=1)].head()
print("Before removing congestion_surcharge: " + str(df['congestion_surcharge'].
 ↪isnull().sum()))
# Impute missing values in `congestion_surcharge` with the mode
df['congestion_surcharge'] = df['congestion_surcharge'].
 ↪fillna(df['congestion_surcharge'].mean())
df.to_csv('3_Cleaned_Sampled_NYC_Taxi_Data.csv', index=False)
```

```
print("After removing congestion_surcharge: " + str(df['congestion_surcharge'].
    ↪isnull().sum()))
```

```
Before removing congestion_surcharge: 64874
After removing congestion_surcharge: 0
```

Are there missing values in other columns? Did you find NaN values in some other set of columns? Handle those missing values below.

[28]:
```
# Handle any remaining missing values

'''
    Since there is no missing values in the dataset, there is no need to handle␣
    ↪any remaining missing values.
'''
df[df.isnull().any(axis=1)].head()
```

[28]:
```
Empty DataFrame
Columns: [VendorID, tpep_pickup_datetime, tpep_dropoff_datetime,
passenger_count, trip_distance, RatecodeID, PULocationID, DOLocationID,
payment_type, fare_amount, extra, mta_tax, tip_amount, tolls_amount,
improvement_surcharge, total_amount, congestion_surcharge, date, hour,
airport_fee]
Index: []
```

1.6.2 2.3 Handling Outliers

[10 marks]

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

2.3.1 [10 marks] Based on the above analysis, it seems that some of the outliers are present due to errors in registering the trips. Fix the outliers.

Some points you can look for: - Entries where `trip_distance` is nearly 0 and `fare_amount` is more than 300 - Entries where `trip_distance` and `fare_amount` are 0 but the pickup and dropoff zones are different (both distance and fare should not be zero for different zones) - Entries where `trip_distance` is more than 250 miles. - Entries where `payment_type` is 0 (there is no payment_type 0 defined in the data dictionary)

These are just some suggestions. You can handle outliers in any way you wish, using the insights from above outlier analysis.

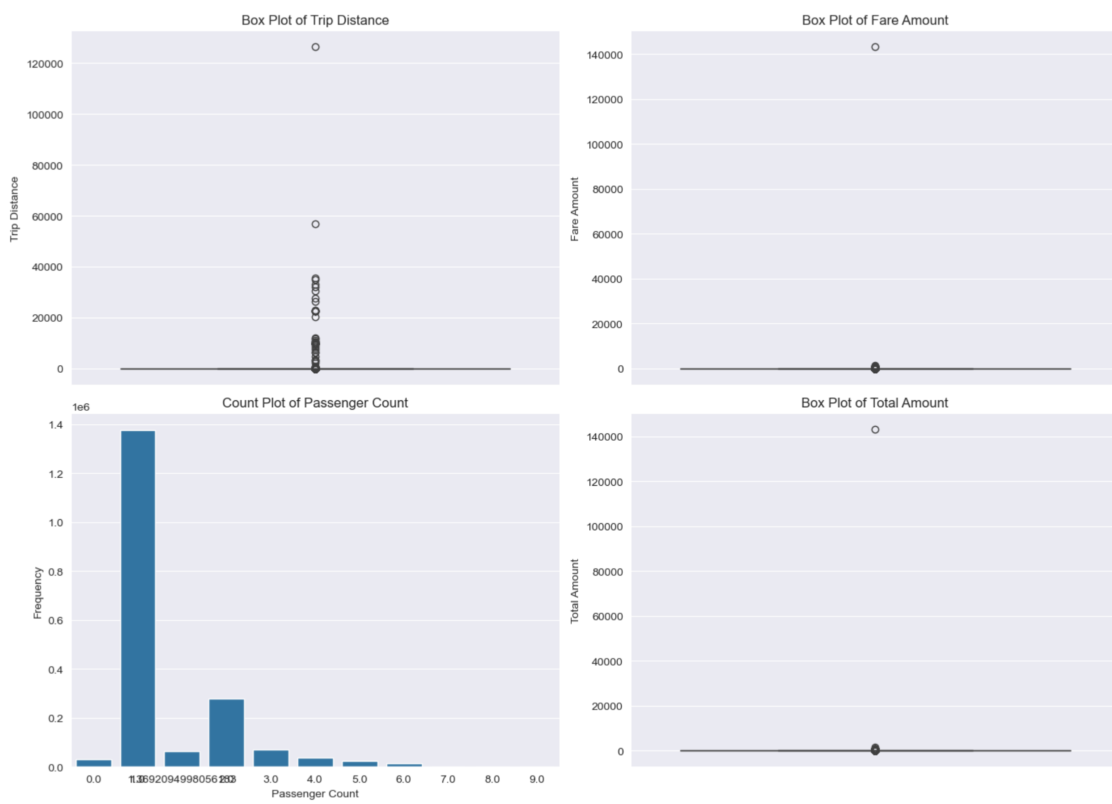
[29]:
```
print('''

• High Fare, Near-Zero Distance: This could indicate errors in recording the␣
    ↪distance or
special circumstances like waiting time.
```

```
• Zero Distance and Fare with Different Zones: This is likely an error, as␣
  ↪trips between
different zones should always have some distance and fare.

• Extremely Long Trips: While possible, trips over 250 miles within NYC are␣
  ↪unusual and might
warrant further investigation.

• Invalid Payment Type: Payment type 0 is undefined, so these records need␣
  ↪correction or
removal.

''')
```

```
• High Fare, Near-Zero Distance: This could indicate errors in recording the
distance or
special circumstances like waiting time.

• Zero Distance and Fare with Different Zones: This is likely an error, as trips
between
different zones should always have some distance and fare.

• Extremely Long Trips: While possible, trips over 250 miles within NYC are
unusual and might
warrant further investigation.

• Invalid Payment Type: Payment type 0 is undefined, so these records need
correction or
removal.
```

[30]:
```python
import matplotlib.pyplot as plt
import seaborn as sns

#... (your data loading and cleaning code)...

# Create a 2x2 grid of subplots
fig, axes = plt.subplots(2, 2, figsize=(14, 10))  # 2 rows, 2 columns

# a. Trip Distance (top-left)
sns.boxplot(y=df['trip_distance'], ax=axes[0, 0])
axes[0, 0].set_title('Box Plot of Trip Distance')
axes[0, 0].set_ylabel('Trip Distance')
```

```
# b. Fare Amount (top-right)
sns.boxplot(y=df['fare_amount'], ax=axes[0, 1])
axes[0, 1].set_title('Box Plot of Fare Amount')
axes[0, 1].set_ylabel('Fare Amount')

# c. Passenger Count (bottom-left)
sns.countplot(x=df['passenger_count'], ax=axes[1, 0])
axes[1, 0].set_title('Count Plot of Passenger Count')
axes[1, 0].set_xlabel('Passenger Count')
axes[1, 0].set_ylabel('Frequency')

# d. Total Amount (bottom-right)
sns.boxplot(y=df['total_amount'], ax=axes[1, 1])
axes[1, 1].set_title('Box Plot of Total Amount')
axes[1, 1].set_ylabel('Total Amount')

# Adjust spacing between subplots
plt.tight_layout()

# Show the plot
plt.show()

# As per the diagram we can see the outliers
```



First, let us remove 7+ passenger counts as there are very less instances.

```
[31]: # remove passenger_count > 6
print("Before removing passenger_count: " + str(df[df['passenger_count'] > 6].
  ↪count().sum()))
df = df[df['passenger_count'] < 7]
print("After removing passenger_count: " + str(df[df['passenger_count'] > 7].
  ↪count().sum()))
```

```
Before removing passenger_count: 420
After removing passenger_count: 0
```

```
[32]: '''
first addresses the specific outlier issues you mentioned (high fare/near-zero␣
  ↪distance, zero distance/fare/different zones,
long trips, invalid payment type). This is important because these are likely␣
  ↪data entry errors and should be handled directly.
'''
# a. High Fare, Near-Zero Distance (Likely Errors - Drop)
high_fare_near_zero = df[(df['trip_distance'] < 0.01) & (df['fare_amount'] >␣
  ↪300)]
print(f"Found {len(high_fare_near_zero)} entries with high fare and near-zero␣
  ↪distance. Dropping.")
df = df.drop(high_fare_near_zero.index)
```

```
Found 34 entries with high fare and near-zero distance. Dropping.
```

```
[33]: # b. Zero Distance and Fare, Different Zones (Errors - Drop)
zero_dist_fare_diff_zones = df[(df['trip_distance'] == 0) & (df['fare_amount']␣
  ↪== 0) & (df['PULocationID'] != df['DOLocationID'])]
print(f"Found {len(zero_dist_fare_diff_zones)} entries with zero distance/fare␣
  ↪and different zones. Dropping.")
df = df.drop(zero_dist_fare_diff_zones.index)
```

```
Found 59 entries with zero distance/fare and different zones. Dropping.
```

```
[34]: # c. Extremely Long Trips (Investigate, then Drop if Invalid)
long_trips = df[df['trip_distance'] > 250]
print(f"Found {len(long_trips)} entries with trip distance over 250 miles.␣
  ↪Dropping (after investigation).")  # In a real scenario, investigate!
df = df.drop(long_trips.index)
```

```
Found 46 entries with trip distance over 250 miles. Dropping (after
investigation).
```

```
[35]: # d. Invalid Payment Type (0) (Errors - Drop)
invalid_payment = df[df['payment_type'] == 0]
```

```
print(f"Found {len(invalid_payment)} entries with invalid payment type.␣
 ↪Dropping.")
df = df.drop(invalid_payment.index)
```

```
Found 64844 entries with invalid payment type. Dropping.
```

[36]:
```
# Continue with outlier handling
```

[37]:
```
'''
The IQR outlier removal is now applied after the specific issues are addressed.
This is a better approach because the IQR method is more general and is␣
 ↪intended to catch naturally occurring outliers,
not necessarily errors.
'''

def remove_outliers_iqr(df, column):
    Q1 = df[column].quantile(0.25)
    Q3 = df[column].quantile(0.75)
    IQR = Q3 - Q1
    upper_bound = Q3 + 1.5 * IQR
    lower_bound = Q1 - 1.5 * IQR
    df_filtered = df[(df[column] >= lower_bound) & (df[column] <= upper_bound)]
    print(f"Removed {len(df) - len(df_filtered)} outliers from '{column}' using␣
 ↪IQR.")
    return df_filtered
```

[38]:
```
df = remove_outliers_iqr(df, 'trip_distance')
```

```
Removed 242001 outliers from 'trip_distance' using IQR.
```

[39]:
```
df = remove_outliers_iqr(df, 'fare_amount')
```

```
Removed 43049 outliers from 'fare_amount' using IQR.
```

[40]:
```
df = remove_outliers_iqr(df, 'total_amount')
```

```
Removed 27508 outliers from 'total_amount' using IQR.
```

[41]:
```
# 5. Further Analysis (on the cleaned data)
print("\nCleaned Data Info:")
df.info()
```

```

Cleaned Data Info:
<class 'pandas.core.frame.DataFrame'>
Index: 1518760 entries, 0 to 1896321
Data columns (total 20 columns):
 #   Column                 Non-Null Count    Dtype
---  ------                 --------------    -----
 0   VendorID               1518760 non-null  int64
```

```
 1   tpep_pickup_datetime   1518760 non-null  object
 2   tpep_dropoff_datetime  1518760 non-null  object
 3   passenger_count        1518760 non-null  float64
 4   trip_distance          1518760 non-null  float64
 5   RatecodeID             1518760 non-null  float64
 6   PULocationID           1518760 non-null  int64
 7   DOLocationID           1518760 non-null  int64
 8   payment_type           1518760 non-null  int64
 9   fare_amount            1518760 non-null  float64
 10  extra                  1518760 non-null  float64
 11  mta_tax                1518760 non-null  float64
 12  tip_amount             1518760 non-null  float64
 13  tolls_amount           1518760 non-null  float64
 14  improvement_surcharge  1518760 non-null  float64
 15  total_amount           1518760 non-null  float64
 16  congestion_surcharge   1518760 non-null  float64
 17  date                   1518760 non-null  object
 18  hour                   1518760 non-null  int64
 19  airport_fee            1518760 non-null  float64
dtypes: float64(12), int64(5), object(3)
memory usage: 243.3+ MB
```

[42]:

```python
print("\nCleaned Data Description:")
print(df.describe())
```

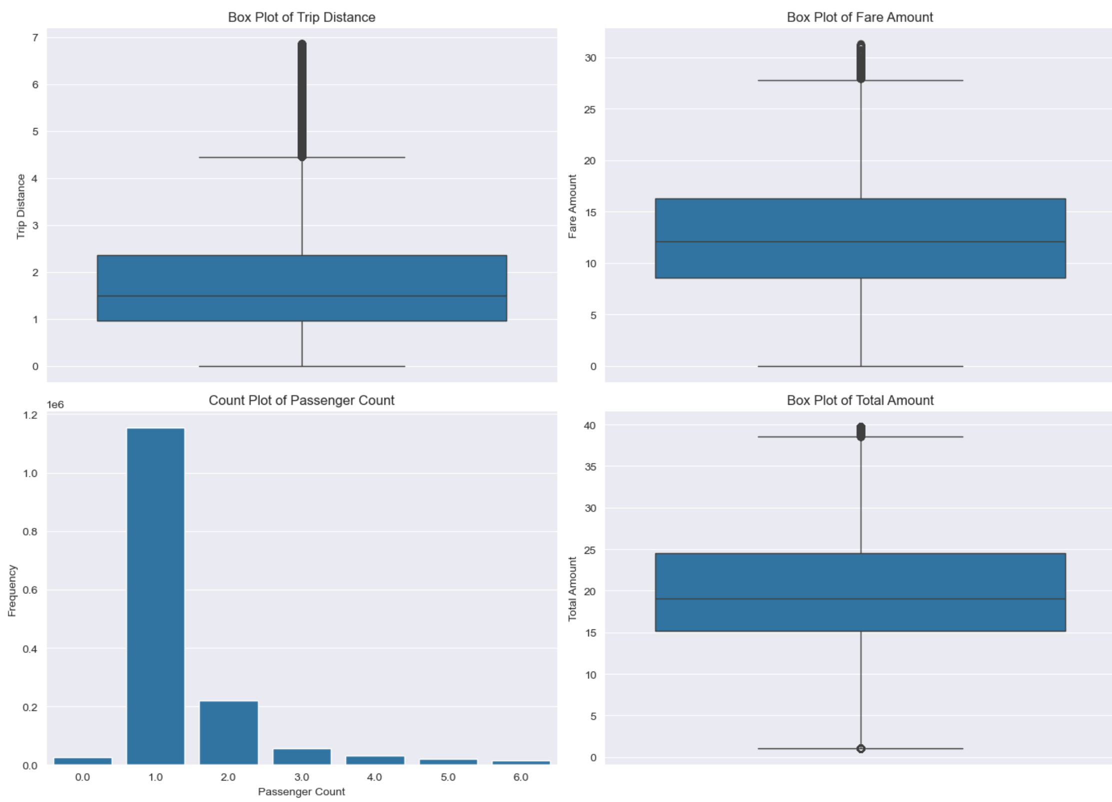
```
Cleaned Data Description:
           VendorID  passenger_count  trip_distance    RatecodeID  \
count  1.518760e+06     1.518760e+06   1.518760e+06  1.518760e+06
mean   1.730843e+00     1.357037e+00   1.807882e+00  1.329338e+00
std    4.435219e-01     8.895864e-01   1.176099e+00  5.609096e+00
min    1.000000e+00     0.000000e+00   0.000000e+00  1.000000e+00
25%    1.000000e+00     1.000000e+00   9.600000e-01  1.000000e+00
50%    2.000000e+00     1.000000e+00   1.500000e+00  1.000000e+00
75%    2.000000e+00     1.000000e+00   2.360000e+00  1.000000e+00
max    2.000000e+00     6.000000e+00   6.850000e+00  9.900000e+01

       PULocationID  DOLocationID  payment_type   fare_amount         extra  \
count  1.518760e+06  1.518760e+06  1.518760e+06  1.518760e+06  1.518760e+06
mean   1.690434e+02  1.673792e+02  1.206937e+00  1.304720e+01  1.423394e+00
std    6.499618e+01  6.823739e+01  4.674629e-01  5.812065e+00  1.470988e+00
min    1.000000e+00  1.000000e+00  1.000000e+00  0.000000e+00  0.000000e+00
25%    1.370000e+02  1.250000e+02  1.000000e+00  8.600000e+00  0.000000e+00
50%    1.630000e+02  1.630000e+02  1.000000e+00  1.210000e+01  1.000000e+00
75%    2.340000e+02  2.360000e+02  1.000000e+00  1.630000e+01  2.500000e+00
max    2.650000e+02  2.650000e+02  4.000000e+00  3.130000e+01  1.025000e+01

            mta_tax    tip_amount  tolls_amount  improvement_surcharge  \
```

```
count  1.518760e+06  1.518760e+06  1.518760e+06          1.518760e+06
mean   4.988971e-01  2.551628e+00  1.049111e-02          9.995332e-01
std    2.402499e-02  1.891801e+00  2.740698e-01          1.920384e-02
min    0.000000e+00  0.000000e+00  0.000000e+00          0.000000e+00
25%    5.000000e-01  1.000000e+00  0.000000e+00          1.000000e+00
50%    5.000000e-01  2.640000e+00  0.000000e+00          1.000000e+00
75%    5.000000e-01  3.780000e+00  0.000000e+00          1.000000e+00
max    4.000000e+00  3.300000e+01  2.735000e+01          1.000000e+00

       total_amount  congestion_surcharge          hour   airport_fee
count  1.518760e+06          1.518760e+06  1.518760e+06  1.518760e+06
mean   2.030638e+01          2.402266e+00  1.429041e+01  1.507957e-02
std    6.888660e+00          4.845437e-01  5.765248e+00  1.567446e-01
min    1.000000e+00          0.000000e+00  0.000000e+00  0.000000e+00
25%    1.512000e+01          2.500000e+00  1.100000e+01  0.000000e+00
50%    1.910000e+01          2.500000e+00  1.500000e+01  0.000000e+00
75%    2.450000e+01          2.500000e+00  1.900000e+01  0.000000e+00
max    3.972000e+01          2.500000e+00  2.300000e+01  1.750000e+00
```

[43]:

```python
import matplotlib.pyplot as plt
import seaborn as sns

#... (your data loading and cleaning code)...

# Create a 2x2 grid of subplots
fig, axes = plt.subplots(2, 2, figsize=(14, 10))  # 2 rows, 2 columns

# a. Trip Distance (top-left)
sns.boxplot(y=df['trip_distance'], ax=axes[0, 0])
axes[0, 0].set_title('Box Plot of Trip Distance')
axes[0, 0].set_ylabel('Trip Distance')

# b. Fare Amount (top-right)
sns.boxplot(y=df['fare_amount'], ax=axes[0, 1])
axes[0, 1].set_title('Box Plot of Fare Amount')
axes[0, 1].set_ylabel('Fare Amount')

# c. Passenger Count (bottom-left)
sns.countplot(x=df['passenger_count'], ax=axes[1, 0])
axes[1, 0].set_title('Count Plot of Passenger Count')
axes[1, 0].set_xlabel('Passenger Count')
axes[1, 0].set_ylabel('Frequency')

# d. Total Amount (bottom-right)
sns.boxplot(y=df['total_amount'], ax=axes[1, 1])
axes[1, 1].set_title('Box Plot of Total Amount')
axes[1, 1].set_ylabel('Total Amount')
```

```
# Adjust spacing between subplots
plt.tight_layout()

# Show the plot
plt.show()

# as we can see the outliers have been removed
```



[44]:
```
df.to_csv("4_Cleaned_Outlier_Removed_Sampled_NYC_Taxi_Data.csv", index=False)
print("Cleaned data saved to '4_Cleaned_Outlier_Removed_Sampled_NYC_Taxi_Data.
↪csv'")
```

```
Cleaned data saved to '4_Cleaned_Outlier_Removed_Sampled_NYC_Taxi_Data.csv'
```

[45]:
```
num_rows = len(df)
print(f"Number of remaining rows: {num_rows}")
```

```
Number of remaining rows: 1518760
```

How will you fix each of these values? Which ones will you drop and which ones will you replace?

[46]:
```python
print('''
When dealing with outliers, there's no one-size-fits-all solution. The best␣
↪approach depends on the nature of your data, the reason for the outliers␣
↪(errors, natural variation, etc.),

Understanding the Outliers:
    Visual Inspection
    Domain Knowledge
    Investigate the Cause

Handling Outliers:
    1. Drop: If an outlier is due to an error or data entry mistake, it may be␣
↪best to drop the entry.
    2. Replace: If an outlier is valid but extreme, it may be replaced with a␣
↪more reasonable value.
    3. Keep: If an outlier is valid and expected, it may be kept as is.

How to do it:
    • Identify outliers using visual inspection, IQR, Z-score, or domain␣
↪knowledge.
    • Use boolean indexing or the .drop() method in Pandas to remove the rows␣
↪containing the outliers.

Imputation (Replacing with another value):
    • Mean, Median, Mode: Replacing with the mean, median, or mode of the␣
↪column.
    • Custom Value: Replacing with a custom value based on domain knowledge.
    • Interpolation: Replacing with a value based on the surrounding data␣
↪points.

Transformation:
    • Log Transformation: Applying a log transformation to the data to reduce␣
↪the impact of outliers.

Winsorizing/Clipping:
      Replacing extreme values with the nearest less extreme value.

Keep the Outliers (Sometimes!):
    • If the outliers are valid data points and part of the distribution, they␣
↪may be kept.
''')
```

```
When dealing with outliers, there's no one-size-fits-all solution. The best
approach depends on the nature of your data, the reason for the outliers
(errors, natural variation, etc.),
```

```
Understanding the Outliers:
    Visual Inspection
    Domain Knowledge
    Investigate the Cause

Handling Outliers:
    1. Drop: If an outlier is due to an error or data entry mistake, it may be
best to drop the entry.
    2. Replace: If an outlier is valid but extreme, it may be replaced with a
more reasonable value.
    3. Keep: If an outlier is valid and expected, it may be kept as is.

How to do it:
    • Identify outliers using visual inspection, IQR, Z-score, or domain
knowledge.
    • Use boolean indexing or the .drop() method in Pandas to remove the rows
containing the outliers.

Imputation (Replacing with another value):
    • Mean, Median, Mode: Replacing with the mean, median, or mode of the
column.
    • Custom Value: Replacing with a custom value based on domain knowledge.
    • Interpolation: Replacing with a value based on the surrounding data
points.

Transformation:
    • Log Transformation: Applying a log transformation to the data to reduce
the impact of outliers.

Winsorizing/Clipping:
      Replacing extreme values with the nearest less extreme value.

Keep the Outliers (Sometimes!):
    • If the outliers are valid data points and part of the distribution, they
may be kept.
```

[47]:
```
# Do any columns need standardising?

print('''
When to Standardize:

1. Machine Learning Algorithms:
    Many machine learning algorithms (especially those based on distance␣
 ↪calculations or gradient descent) benefit from standardization.
        Standardizing features can:
            Prevent features with larger scales from dominating the model.
```

```
            Improve numerical stability.
            Speed up convergence in some algorithms.

2. Comparing Features with Different Units:
        If you have features with different units or scales (e.g.,␣
 ↪trip_distance in miles and fare_amount in dollars),
        standardizing them can make them more comparable.

3. Data Visualization:
    In some cases, standardizing can make it easier to visualize data with␣
 ↪different scales on the same plot.

Common Standardization Methods:
1. Z-score Standardization:
2. Min-Max Scaling:

based on this data : trip_distance and fare_amount and total_amount should be␣
 ↪standardized as they have different units.
''')
```

```
When to Standardize:

1. Machine Learning Algorithms:
    Many machine learning algorithms (especially those based on distance
calculations or gradient descent) benefit from standardization.
        Standardizing features can:
            Prevent features with larger scales from dominating the model.
            Improve numerical stability.
            Speed up convergence in some algorithms.

2. Comparing Features with Different Units:
        If you have features with different units or scales (e.g., trip_distance
in miles and fare_amount in dollars),
        standardizing them can make them more comparable.

3. Data Visualization:
    In some cases, standardizing can make it easier to visualize data with
different scales on the same plot.

Common Standardization Methods:
1. Z-score Standardization:
2. Min-Max Scaling:

based on this data : trip_distance and fare_amount and total_amount should be
standardized as they have different units.
```

[48]:
```
try:
    df = pd.read_csv('4_Cleaned_Outlier_Removed_Sampled_NYC_Taxi_Data.csv')
except FileNotFoundError:
    print("Error: '4_Cleaned_Outlier_Removed_Sampled_NYC_Taxi_Data.csv'␣
↪DataFrame not found or saved file not found. Please make sure you have␣
↪sampled and saved the data first.")
```

[49]:
```
# Analyse the above parameters
columns_to_check = ['fare_amount', 'tip_amount', 'total_amount',␣
↪'trip_distance']

for col in columns_to_check:
    num_zeros = (df[col] == 0).sum()
    num_negatives = (df[col] < 0).sum()
    print(f"\nColumn '{col}':")
    print(f"  - Number of zero values: {num_zeros}")
    print(f"  - Number of negative values: {num_negatives}")

'''
There is no negative values in the dataset before standardization.
'''
```

```
Column 'fare_amount':
  - Number of zero values: 191
  - Number of negative values: 0

Column 'tip_amount':
  - Number of zero values: 335368
  - Number of negative values: 0

Column 'total_amount':
  - Number of zero values: 0
  - Number of negative values: 0

Column 'trip_distance':
  - Number of zero values: 15541
  - Number of negative values: 0
```

[49]:
```
'\nThere is no negative values in the dataset before standardization.\n'
```

[50]:
```
from sklearn.preprocessing import StandardScaler
# Select the columns to standardize
cols_to_standardize = ['trip_distance', 'fare_amount', 'total_amount']  #␣
↪Include relevant columns

# Create a StandardScaler object
scaler = StandardScaler()
```

```python
# Fit the scaler to the selected columns
scaler.fit(df[cols_to_standardize])

# Transform the selected columns
df[cols_to_standardize] = scaler.transform(df[cols_to_standardize])


print("\nStandardized Data Description:")
df.describe() # You'll see that the selected columns now have mean=0 and std=1
```

```
Standardized Data Description:
```

[50]:

	VendorID	passenger_count	trip_distance	RatecodeID
count	1.518760e+06	1.518760e+06	1.518760e+06	1.518760e+06
mean	1.730843e+00	1.357037e+00	-2.799578e-16	1.329338e+00
std	4.435219e-01	8.895864e-01	1.000000e+00	5.609096e+00
min	1.000000e+00	0.000000e+00	-1.537186e+00	1.000000e+00
25%	1.000000e+00	1.000000e+00	-7.209280e-01	1.000000e+00
50%	2.000000e+00	1.000000e+00	-2.617827e-01	1.000000e+00
75%	2.000000e+00	1.000000e+00	4.694487e-01	1.000000e+00
max	2.000000e+00	6.000000e+00	4.287157e+00	9.900000e+01

	PULocationID	DOLocationID	payment_type	fare_amount	extra
count	1.518760e+06	1.518760e+06	1.518760e+06	1.518760e+06	1.518760e+06
mean	1.690434e+02	1.673792e+02	1.206937e+00	-1.758064e-15	1.423394e+00
std	6.499618e+01	6.823739e+01	4.674629e-01	1.000000e+00	1.470988e+00
min	1.000000e+00	1.000000e+00	1.000000e+00	-2.244849e+00	0.000000e+00
25%	1.370000e+02	1.250000e+02	1.000000e+00	-7.651675e-01	0.000000e+00
50%	1.630000e+02	1.630000e+02	1.000000e+00	-1.629718e-01	1.000000e+00
75%	2.340000e+02	2.360000e+02	1.000000e+00	5.596631e-01	2.500000e+00
max	2.650000e+02	2.650000e+02	4.000000e+00	3.140502e+00	1.025000e+01

	mta_tax	tip_amount	tolls_amount	improvement_surcharge
count	1.518760e+06	1.518760e+06	1.518760e+06	1.518760e+06
mean	4.988971e-01	2.551628e+00	1.049111e-02	9.995332e-01
std	2.402499e-02	1.891801e+00	2.740698e-01	1.920384e-02
min	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00
25%	5.000000e-01	1.000000e+00	0.000000e+00	1.000000e+00
50%	5.000000e-01	2.640000e+00	0.000000e+00	1.000000e+00
75%	5.000000e-01	3.780000e+00	0.000000e+00	1.000000e+00
max	4.000000e+00	3.300000e+01	2.735000e+01	1.000000e+00

	total_amount	congestion_surcharge	hour	airport_fee
count	1.518760e+06	1.518760e+06	1.518760e+06	1.518760e+06
mean	-7.570277e-16	2.402266e+00	1.429041e+01	1.507957e-02

```
std     1.000000e+00         4.845437e-01  5.765248e+00  1.567446e-01
min    -2.802633e+00         0.000000e+00  0.000000e+00  0.000000e+00
25%    -7.528868e-01         2.500000e+00  1.100000e+01  0.000000e+00
50%    -1.751255e-01         2.500000e+00  1.500000e+01  0.000000e+00
75%     6.087717e-01         2.500000e+00  1.900000e+01  0.000000e+00
max     2.818201e+00         2.500000e+00  2.300000e+01  1.750000e+00
```

[51]:
```
df.to_csv("Standard_Sampled_NYC_Taxi_Data.csv", index=False)
print("Standardized data saved to 'Standard_Sampled_NYC_Taxi_Data.csv'")
```

```
Standardized data saved to 'Standard_Sampled_NYC_Taxi_Data.csv'
```

1.7 3 Exploratory Data Analysis

[90 marks]

[52]:
```
df.columns.tolist()
```

[52]:
```
['VendorID',
 'tpep_pickup_datetime',
 'tpep_dropoff_datetime',
 'passenger_count',
 'trip_distance',
 'RatecodeID',
 'PULocationID',
 'DOLocationID',
 'payment_type',
 'fare_amount',
 'extra',
 'mta_tax',
 'tip_amount',
 'tolls_amount',
 'improvement_surcharge',
 'total_amount',
 'congestion_surcharge',
 'date',
 'hour',
 'airport_fee']
```

3.1 General EDA: Finding Patterns and Trends [40 marks]

3.1.1 [3 marks] Categorise the varaibles into Numerical or Categorical. * `VendorID`: * `tpep_pickup_datetime`: * `tpep_dropoff_datetime`: * `passenger_count`: * `trip_distance`: * `RatecodeID`: * `PULocationID`: * `DOLocationID`: * `payment_type`: * `pickup_hour`: * `trip_duration`:

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

- `fare_amount`
- `extra`

• mta_tax
• tip_amount
• tolls_amount
• improvement_surcharge
• total_amount
• congestion_surcharge
• airport_fee

[53]:
```
print('''
Categorical Variables:

• VendorID
• RatecodeID
• PULocationID
• DOLocationID
• payment_type
• pickup_hour

Numerical Variables:
• tpep_pickup_datetime
• tpep_dropoff_datetime
• passenger_count
• trip_distance
• trip_duration

Dates and Times: Dates and times can sometimes be treated as both numerical and␣
 ↪categorical, depending on the analysis. For example, you might use the␣
 ↪numerical values of dates to calculate durations or time intervals, or you␣
 ↪might treat dates as categories to analyze trends over time.

• fare_amount
• extra
• mta_tax
• tip_amount
• tolls_amount
• improvement_surcharge
• total_amount
• congestion_surcharge
• airport_fee
        These monetary parameters are all numerical variables. They represent␣
 ↪amounts of money and can be treated as continuous numerical data.
''')
```

Categorical Variables:

• VendorID
• RatecodeID

- PULocationID
- DOLocationID
- payment_type
- pickup_hour

Numerical Variables:
- tpep_pickup_datetime
- tpep_dropoff_datetime
- passenger_count
- trip_distance
- trip_duration

Dates and Times: Dates and times can sometimes be treated as both numerical and categorical, depending on the analysis. For example, you might use the numerical values of dates to calculate durations or time intervals, or you might treat dates as categories to analyze trends over time.

- fare_amount
- extra
- mta_tax
- tip_amount
- tolls_amount
- improvement_surcharge
- total_amount
- congestion_surcharge
- airport_fee

These monetary parameters are all numerical variables. They represent amounts of money and can be treated as continuous numerical data.

Temporal Analysis **3.1.2** [5 marks] Analyse the distribution of taxi pickups by hours, days of the week, and months.

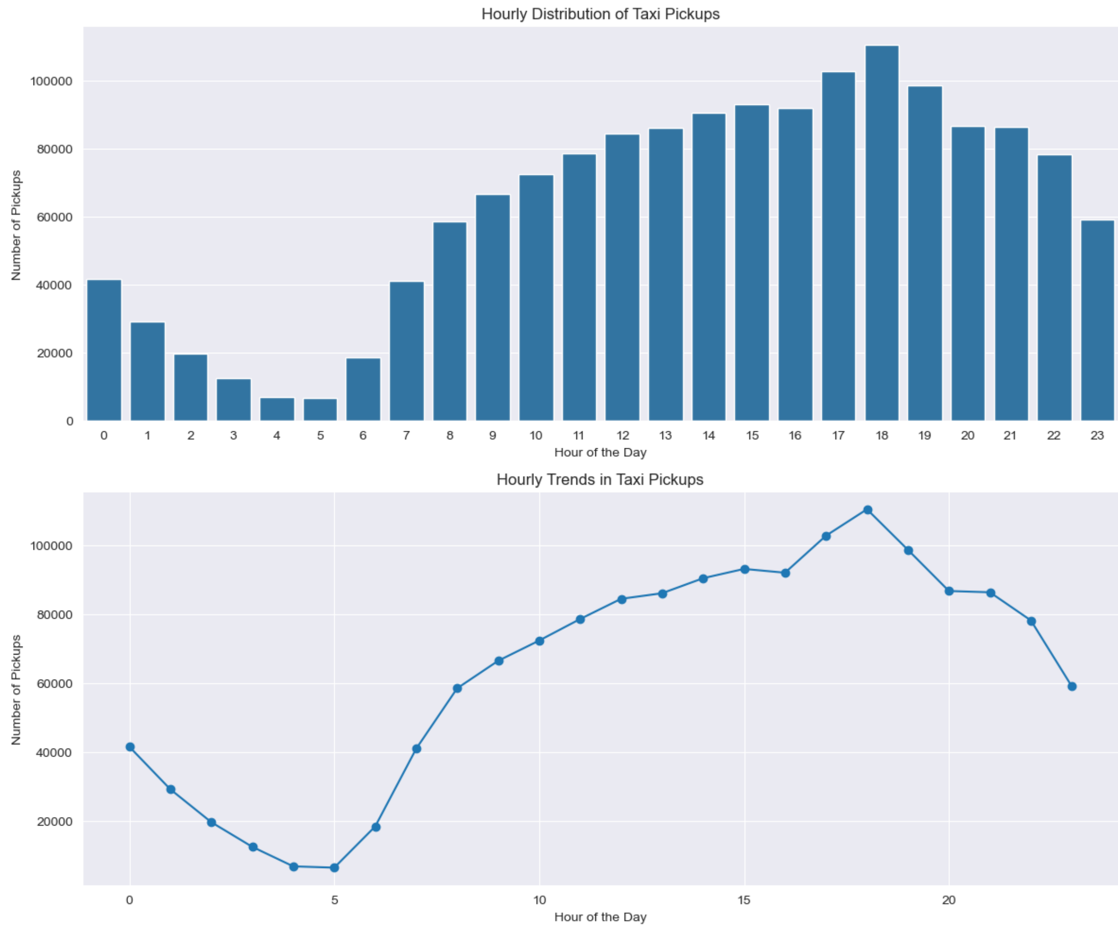
[54]:
```
try:
    df = pd.read_csv("Standard_Sampled_NYC_Taxi_Data.csv")
    print("Data loaded successfully.")
except FileNotFoundError:
    print("Error: 'Standard_Sampled_NYC_Taxi_Data.csv' not found. Please␣
 ↪provide the correct file path.")
    exit()
```

Data loaded successfully.

[55]:
```
# Find and show the hourly trends in taxi pickups
'''
Hourly Trends: Identify peak hours when taxi demand is highest (e.g., rush␣
 ↪hours, late nights).
'''
```

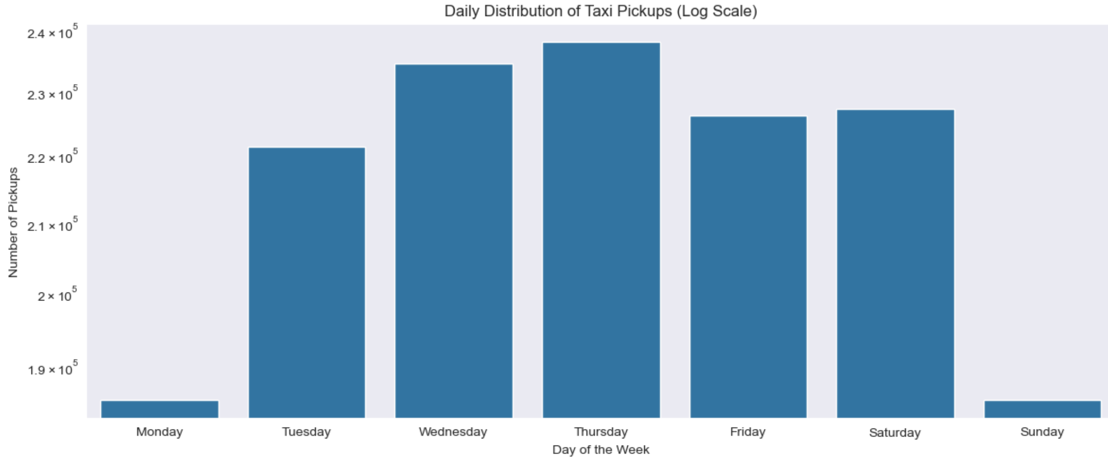
```
# 2. Convert pickup and dropoff datetime columns to datetime objects
df['tpep_pickup_datetime'] = pd.to_datetime(df['tpep_pickup_datetime'])
df['tpep_dropoff_datetime'] = pd.to_datetime(df['tpep_dropoff_datetime'])

# 3. Extract hour, day of the week, and month from pickup datetime
df['pickup_hour'] = df['tpep_pickup_datetime'].dt.hour
df['pickup_dayofweek'] = df['tpep_pickup_datetime'].dt.dayofweek  # Monday=0,␣
↪Sunday=6
df['pickup_month'] = df['tpep_pickup_datetime'].dt.month

# a. Hourly Distribution
'''
Hourly Distribution: Calculates the number of pickups for each hour of the day␣
↪using value_counts()
and plots a bar chart using sns.barplot().
'''
import matplotlib.pyplot as plt
import seaborn as sns

# Assuming `df` contains a column named 'pickup_hour' (integer 0-23)
# Group data to get hourly pickup counts
hourly_pickups = df['pickup_hour'].value_counts().sort_index()

# Group by pickup hour and count occurrences
hourly_trends = df.groupby('pickup_hour')['pickup_hour'].count()

# Create a 2-row, 1-column grid for subplots
fig, axes = plt.subplots(2, 1, figsize=(12, 10))

# (a) Bar Plot: Hourly Distribution of Taxi Pickups
sns.barplot(x=hourly_pickups.index, y=hourly_pickups.values, ax=axes[0])
axes[0].set_title('Hourly Distribution of Taxi Pickups')
axes[0].set_xlabel('Hour of the Day')
axes[0].set_ylabel('Number of Pickups')

# (b) Line Plot: Hourly Trends
axes[1].plot(hourly_trends.index, hourly_trends.values, marker='o',␣
↪linestyle='-')
axes[1].set_title('Hourly Trends in Taxi Pickups')
axes[1].set_xlabel('Hour of the Day')
axes[1].set_ylabel('Number of Pickups')
axes[1].grid(True)

# Adjust layout to prevent overlap
plt.tight_layout()
```

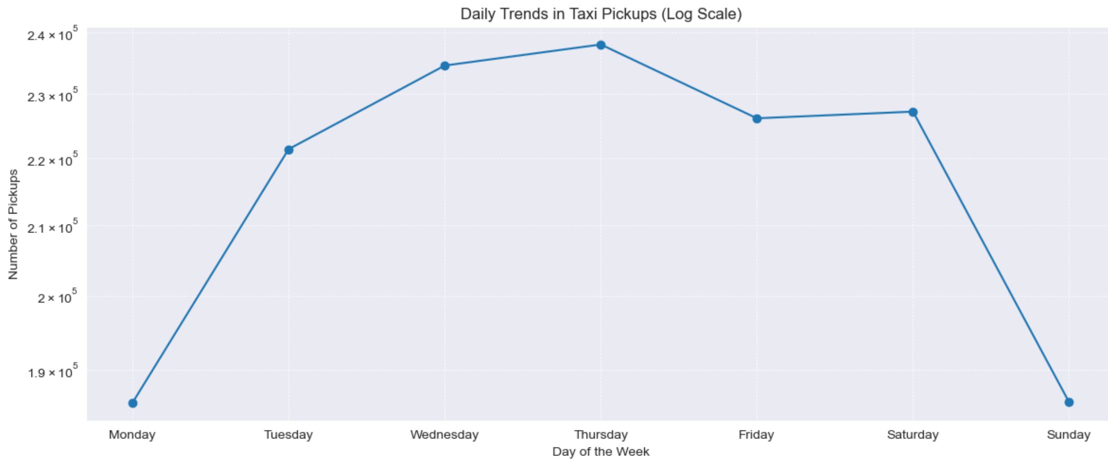
```
# Show the combined plots
plt.show()
```



[56]:
```
# Assuming `df` contains a column named 'pickup_dayofweek' (integer 0-6, where␣
↪Monday=0 and Sunday=6)

# 2. Convert pickup and dropoff datetime columns to datetime objects
df['tpep_pickup_datetime'] = pd.to_datetime(df['tpep_pickup_datetime'])
df['tpep_dropoff_datetime'] = pd.to_datetime(df['tpep_dropoff_datetime'])

# 3. Extract hour, day of the week, and month from pickup datetime
df['pickup_dayofweek'] = df['tpep_pickup_datetime'].dt.dayofweek  # Monday=0,␣
↪Sunday=6


# Define day labels
day_labels = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday',␣
↪'Saturday', 'Sunday']
```

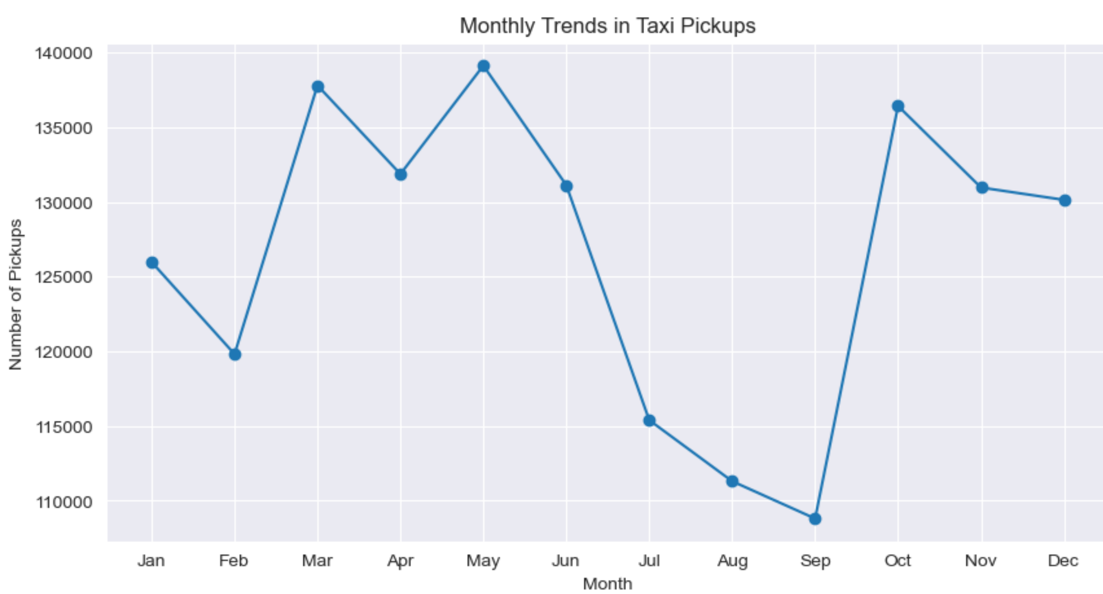
```
# Count pickups per day of the week, ensuring all days (0-6) are present
daily_pickups = df['pickup_dayofweek'].value_counts().reindex(range(7),␣
↪fill_value=1)  # Avoid log(0) issue

# Compute daily trends
daily_trends = df.groupby('pickup_dayofweek')['pickup_dayofweek'].count().
↪reindex(range(7), fill_value=1)  # Avoid log(0) issue

# Create a 2-row, 1-column grid for subplots
fig, axes = plt.subplots(2, 1, figsize=(12, 10))

# (a) Bar Plot: Daily Distribution of Taxi Pickups (Log Scale)
sns.barplot(x=day_labels, y=daily_pickups.values, ax=axes[0])
axes[0].set_title('Daily Distribution of Taxi Pickups (Log Scale)')
axes[0].set_xlabel('Day of the Week')
axes[0].set_ylabel('Number of Pickups')
axes[0].set_yscale('log')  # Set y-axis to log scale

# (b) Line Plot: Daily Trends (Log Scale)
axes[1].plot(day_labels, daily_trends.values, marker='o', linestyle='-')
axes[1].set_title('Daily Trends in Taxi Pickups (Log Scale)')
axes[1].set_xlabel('Day of the Week')
axes[1].set_ylabel('Number of Pickups')
axes[1].set_yscale('log')  # Set y-axis to log scale
axes[1].grid(True, which="both", linestyle="--", linewidth=0.5)  # Improve␣
↪log-scale grid

# Adjust layout to prevent overlap
plt.tight_layout()

# Show the combined plots
plt.show()
```





[57]:
```
# Show the monthly trends in pickups
'''
Monthly Distribution: Counts the number of pickups for each month using␣
↪value_counts() and plots a bar chart using sns.barplot().
'''
# c. Monthly Distribution
monthly_pickups = df['pickup_month'].value_counts().sort_index()

plt.figure(figsize=(10, 5))
sns.barplot(x=monthly_pickups.index, y=monthly_pickups.values)
plt.title('Monthly Distribution of Taxi Pickups')
plt.xlabel('Month')
plt.ylabel('Number of Pickups')
plt.show()

# Group by pickup month and count the number of pickups
monthly_trends = df.groupby('pickup_month')['pickup_month'].count()
```

```
# Create the line plot
plt.figure(figsize=(10, 5))
monthly_trends.plot(kind='line', marker='o')
plt.title('Monthly Trends in Taxi Pickups')
plt.xlabel('Month')
plt.ylabel('Number of Pickups')
plt.xticks(ticks=range(1, 13), labels=['Jan', 'Feb', 'Mar', 'Apr', 'May',␣
↪'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'])  # Set x-axis labels
plt.grid(True)
plt.show()
```



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

[58]:
```python
# Analyse the above parameters
columns_to_check = ['fare_amount', 'tip_amount', 'total_amount',␣
↪'trip_distance']

for col in columns_to_check:
    num_zeros = (df[col] == 0).sum()
    num_negatives = (df[col] < 0).sum()
    print(f"\nColumn '{col}':")
    print(f"  - Number of zero values: {num_zeros}")
    print(f"  - Number of negative values: {num_negatives}")

print('''
The standardization process (using StandardScaler ) centers the data around a␣
↪mean of 0 and
scales it to have a standard deviation of 1. This inherently introduces␣
↪negative values, as any
values originally below the mean will become negative after standardization.

Therefore, seeing negative values after standardization is expected.
''')
```

```
Column 'fare_amount':
  - Number of zero values: 0
  - Number of negative values: 881972

Column 'tip_amount':
  - Number of zero values: 335368
  - Number of negative values: 0

Column 'total_amount':
  - Number of zero values: 0
  - Number of negative values: 866205

Column 'trip_distance':
  - Number of zero values: 0
  - Number of negative values: 924841

The standardization process (using StandardScaler ) centers the data around a
mean of 0 and
scales it to have a standard deviation of 1. This inherently introduces negative
values, as any
```

values originally below the mean will become negative after standardization.

Therefore, seeing negative values after standardization is expected.

Do you think it is beneficial to create a copy DataFrame leaving out the zero values from these?

3.1.3 [2 marks] Filter out the zero values from the above columns.

Note: 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?

[59]:
```
# Create a df with non zero entries for the selected parameters.
# Filter out zero values from specified columns

'''
 It makes sense that some trips might have zero distance  but non-zero fares or␣
 ↪total amounts if the pickup and dropoff locations
 are within the same zone.

 In those cases, filtering out the rows with zero trip_distance could lead to␣
 ↪loss of information, as those trips might still
 be valid and have valuable insights.

 code includes a condition to filter out zero values in trip_distance ONLY IF␣
 ↪both fare_amount and total_amount are also zero.
 This ensures that trips with zero distance but non-zero fares are retained.
'''
columns_to_filter = ['fare_amount', 'tip_amount', 'total_amount',␣
 ↪'trip_distance']

for col in columns_to_filter:
    # Count zero values before filtering
    num_zeros_before = (df[col] == 0).sum()

    # Filter out zero values ONLY IF fare_amount and total_amount are also zero
    if col == 'trip_distance':
        df = df[~((df[col] == 0) & (df['fare_amount'] == 0) &␣
 ↪(df['total_amount'] == 0))]
    else:
        df = df[df[col] > 0]

    # Count zero values after filtering
    num_zeros_after = (df[col] == 0).sum()

    print(f"\nColumn '{col}':")
    print(f"  - Number of zero values before filtering: {num_zeros_before}")
    print(f"  - Number of zero values after filtering: {num_zeros_after}")
```

```
df.to_csv("Cleaned_Standard_Sampled_NYC_Taxi_Data.csv", index=False)
print("Cleaned data saved to 'Cleaned_Standard_Sampled_NYC_Taxi_Data.csv'")
```

```
Column 'fare_amount':
  - Number of zero values before filtering: 0
  - Number of zero values after filtering: 0

Column 'tip_amount':
  - Number of zero values before filtering: 135019
  - Number of zero values after filtering: 0

Column 'total_amount':
  - Number of zero values before filtering: 0
  - Number of zero values after filtering: 0

Column 'trip_distance':
  - Number of zero values before filtering: 0
  - Number of zero values after filtering: 0
Cleaned data saved to 'Cleaned_Standard_Sampled_NYC_Taxi_Data.csv'
```

[60]:
```
try:
    df = pd.read_csv("Cleaned_Standard_Sampled_NYC_Taxi_Data.csv")
    print("Data loaded successfully.")
except FileNotFoundError:
    print("Error: 'Cleaned_Standard_Sampled_NYC_Taxi_Data.csv' not found.␣
↪Please provide the correct file path.")
    exit()
```

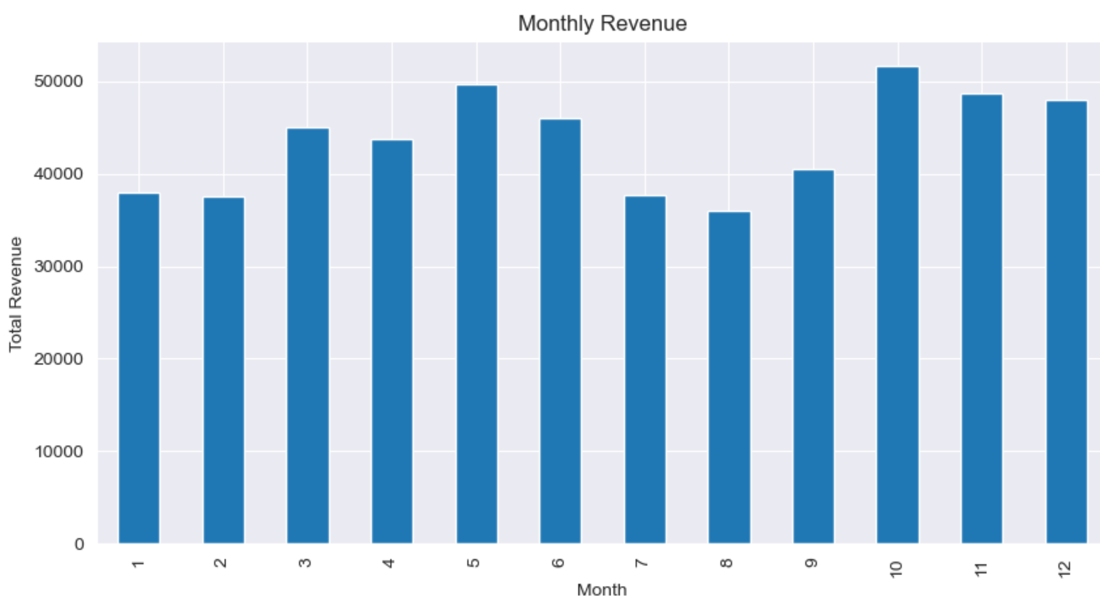
```
Data loaded successfully.
```

3.1.4 [3 marks] Analyse the monthly revenue (`total_amount`) trend

[61]:
```
# Group data by month and analyse monthly revenue

# Convert pickup datetime to datetime object
df['tpep_pickup_datetime'] = pd.to_datetime(df['tpep_pickup_datetime'])

# Extract month from pickup datetime
df['pickup_month'] = df['tpep_pickup_datetime'].dt.month

# Group by month and calculate total revenue
monthly_revenue = df.groupby('pickup_month')['total_amount'].sum()

# Print the monthly revenue
print("\nMonthly Revenue:")
print(monthly_revenue)
```

```python
# (Optional) Plot the monthly revenue
import matplotlib.pyplot as plt
plt.figure(figsize=(10, 5))
monthly_revenue.plot(kind='bar')
plt.title('Monthly Revenue')
plt.xlabel('Month')
plt.ylabel('Total Revenue')
plt.show()
```

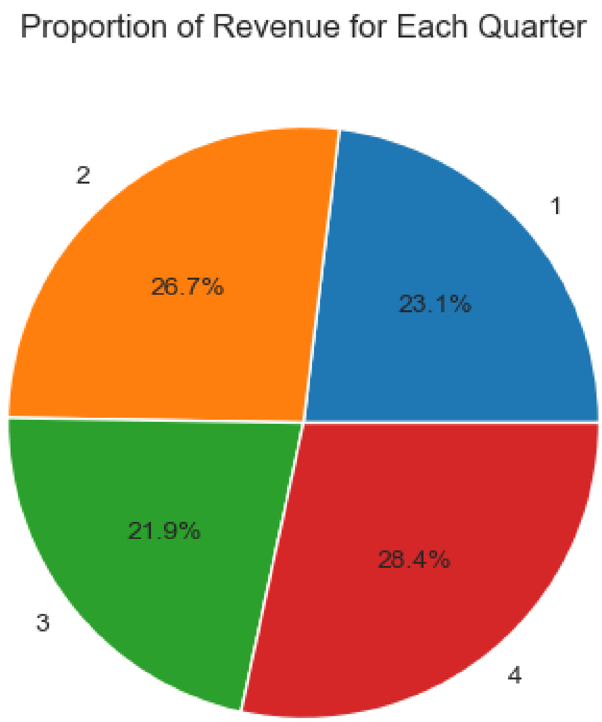
```
Monthly Revenue:
pickup_month
1     37904.172251
2     37595.266599
3     44993.288456
4     43732.858820
5     49650.150059
6     46031.535554
7     37671.098187
8     36019.789856
9     40547.092788
10    51713.515704
11    48675.950752
12    47964.204578
Name: total_amount, dtype: float64
```



3.1.5 [3 marks] Show the proportion of each quarter of the year in the revenue

[62]:
```
# Calculate proportion of each quarter

# Convert pickup datetime to datetime object
df['tpep_pickup_datetime'] = pd.to_datetime(df['tpep_pickup_datetime'])

# Extract quarter from pickup datetime
df['pickup_quarter'] = df['tpep_pickup_datetime'].dt.quarter

# Group by quarter and calculate total revenue
quarterly_revenue = df.groupby('pickup_quarter')['total_amount'].sum()

# Calculate proportion of revenue for each quarter
total_revenue = quarterly_revenue.sum()
quarter_proportions = quarterly_revenue / total_revenue

# Print the quarter proportions
print("\nProportion of Revenue for Each Quarter:")
print(quarter_proportions)

# (Optional) Plot the quarter proportions
import matplotlib.pyplot as plt
plt.figure(figsize=(8, 5))
quarter_proportions.plot(kind='pie', autopct='%1.1f%%')
plt.title('Proportion of Revenue for Each Quarter')
plt.ylabel('')  # Remove the default ylabel
plt.show()
```

```
Proportion of Revenue for Each Quarter:
pickup_quarter
1    0.230609
2    0.266823
3    0.218638
4    0.283931
Name: total_amount, dtype: float64
```

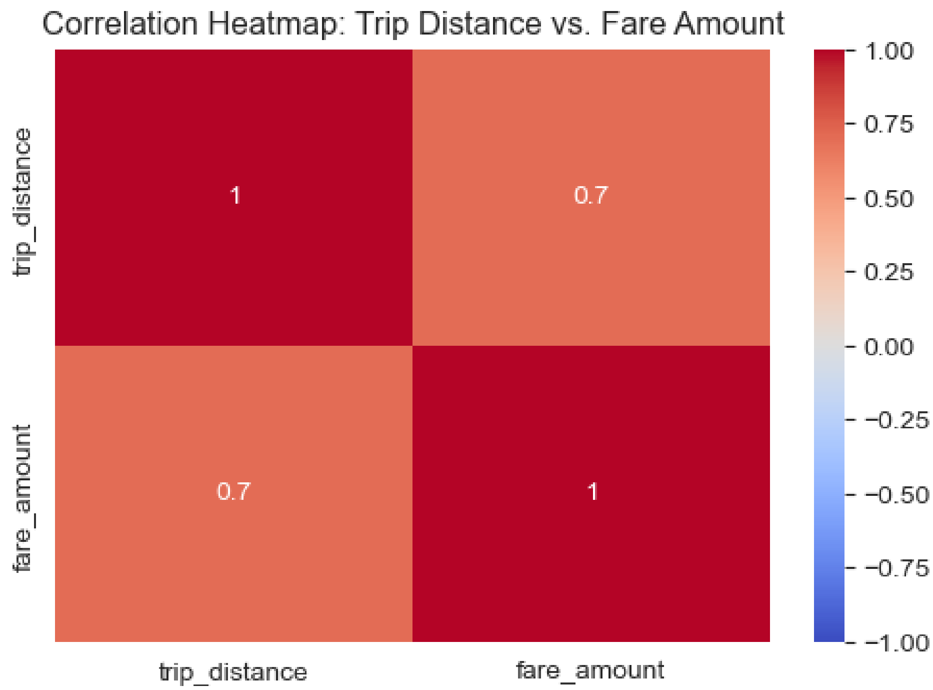


3.1.6 [3 marks] Visualise the relationship between `trip_distance` and `fare_amount`. Also find the correlation value for these two.

Hint: You can leave out the trips with trip_distance = 0

[63]:
```
# Show how trip fare is affected by distance

df['tpep_pickup_datetime'] = pd.to_datetime(df['tpep_pickup_datetime']).
 ↪astype(int) / 10**9
df['tpep_dropoff_datetime'] = pd.to_datetime(df['tpep_dropoff_datetime']).
 ↪astype(int) / 10**9

# 7. Create a heatmap to visualize the relationship between trip_distance and␣
 ↪fare_amount

# Calculate the correlation matrix
corr_matrix = df[['trip_distance', 'fare_amount']].corr()

# Create the heatmap
plt.figure(figsize=(6, 4))
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', vmin=-1, vmax=1)
plt.title('Correlation Heatmap: Trip Distance vs. Fare Amount')
```

```
plt.show()

# Correlation value (already calculated in the heatmap, but printing it again␣
↪for clarity)
correlation = df['trip_distance'].corr(df['fare_amount'])
print(f"\nCorrelation between trip_distance and fare_amount: {correlation:.2f}")
```

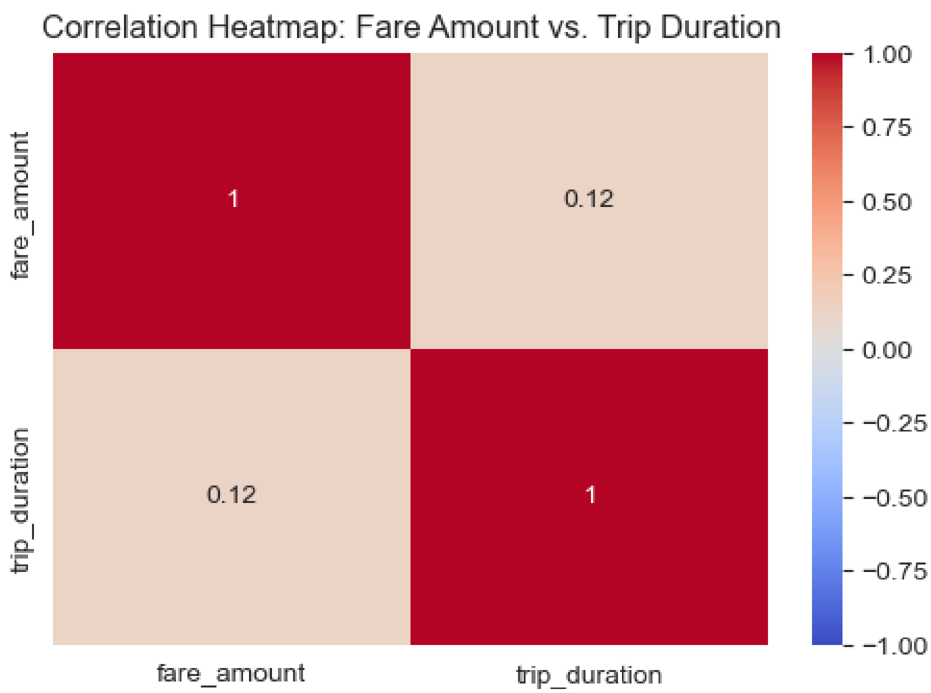



```
Correlation between trip_distance and fare_amount: 0.70
```

3.1.7 [5 marks] Find and visualise the correlation between: 1. `fare_amount` and trip duration (pickup time to dropoff time) 2. `fare_amount` and `passenger_count` 3. `tip_amount` and `trip_distance`

[64]:
```
# Show relationship between fare and trip duration

# Calculate trip duration
df['trip_duration'] = df['tpep_dropoff_datetime'] - df['tpep_pickup_datetime']

# Visualize relationship between fare_amount and trip_duration using heatmap␣
↪and correlation

# Calculate the correlation matrix
corr_matrix = df[['fare_amount', 'trip_duration']].corr()
```

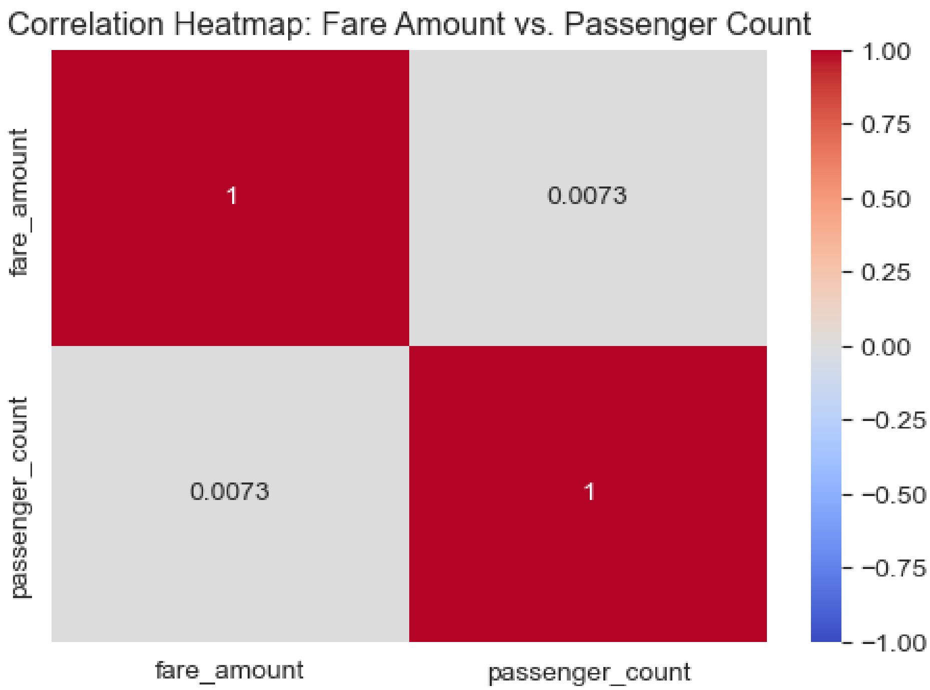
```
# Create the heatmap
plt.figure(figsize=(6, 4))
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', vmin=-1, vmax=1)
plt.title('Correlation Heatmap: Fare Amount vs. Trip Duration')
plt.show()

# Correlation value (already calculated in the heatmap, but printing it again␣
 ↪for clarity)
correlation = df['fare_amount'].corr(df['trip_duration'])
print(f"\nCorrelation between fare_amount and trip_duration: {correlation:.2f}")
```




```
Correlation between fare_amount and trip_duration: 0.12
```

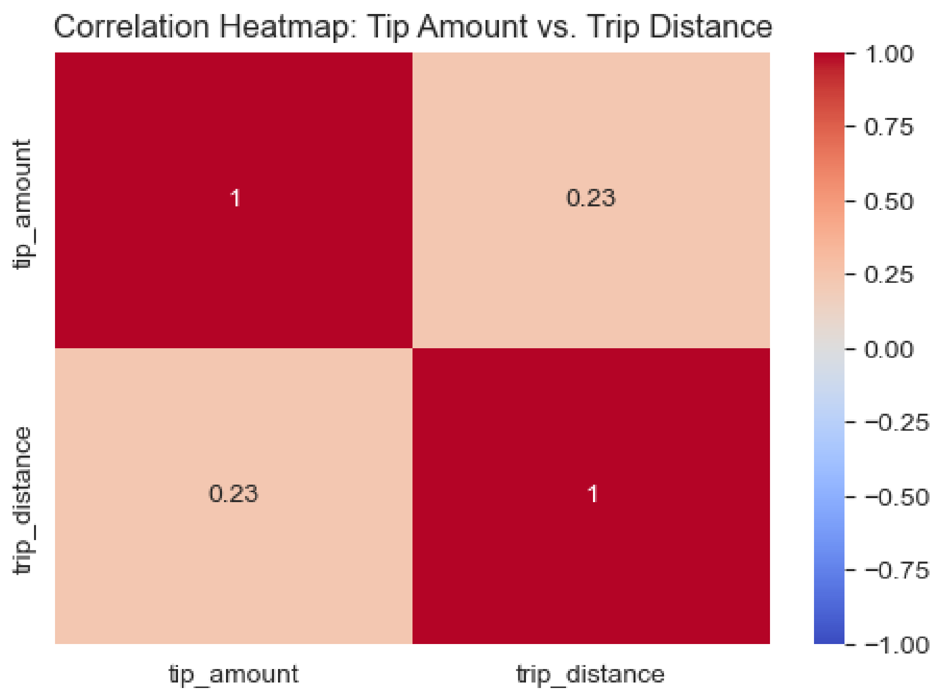
[65]:
```
# Show relationship between fare and number of passengers
# Calculate the correlation matrix
corr_matrix = df[['fare_amount', 'passenger_count']].corr()

# Create the heatmap
plt.figure(figsize=(6, 4))
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', vmin=-1, vmax=1)
plt.title('Correlation Heatmap: Fare Amount vs. Passenger Count')
plt.show()
```

```
# Correlation value (already calculated in the heatmap, but printing it again␣
↪for clarity)
correlation = df['fare_amount'].corr(df['passenger_count'])
print(f"\nCorrelation between fare_amount and passenger_count: {correlation:.
↪2f}")
```



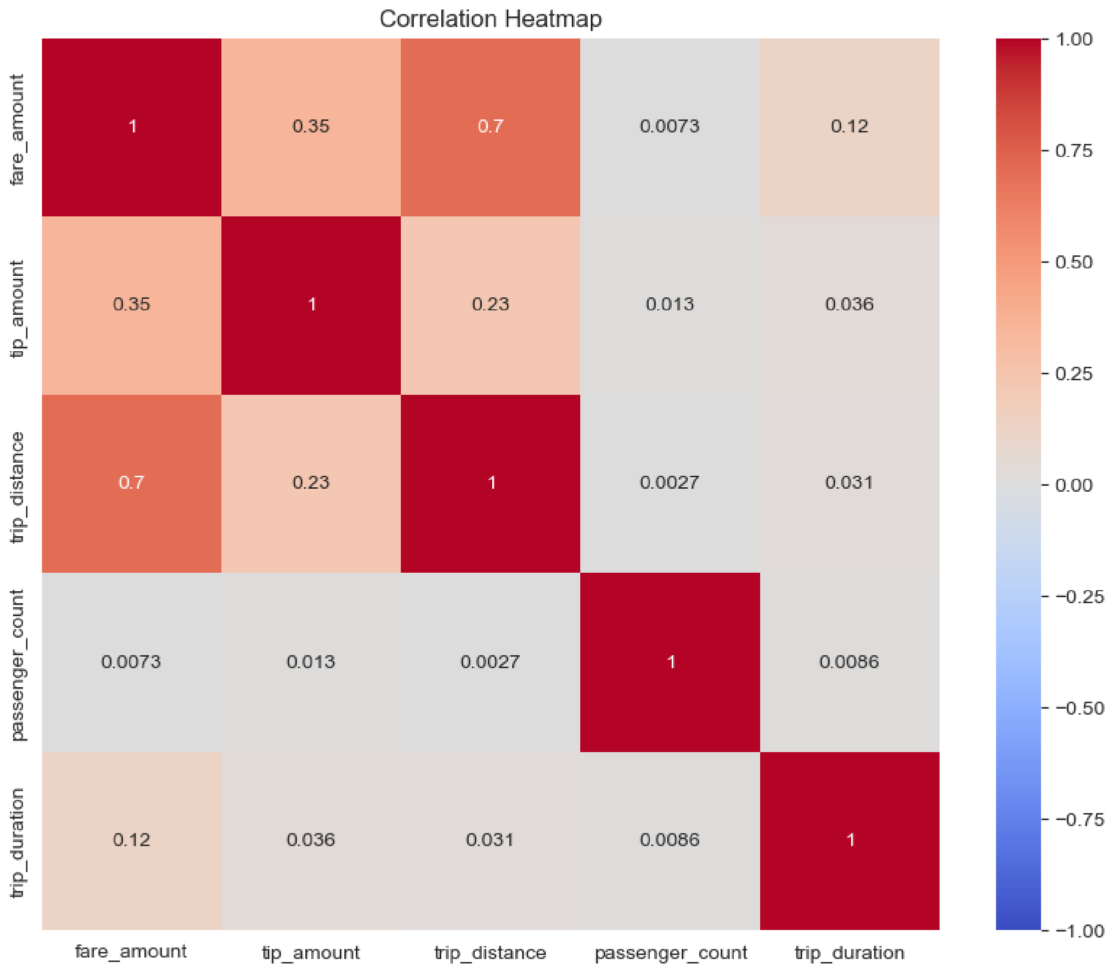

```
Correlation between fare_amount and passenger_count: 0.01
```

[66]:
```
# Show relationship between tip and trip distance

# Calculate the correlation matrix
corr_matrix = df[['tip_amount', 'trip_distance']].corr()

# Create the heatmap
plt.figure(figsize=(6, 4))
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', vmin=-1, vmax=1)
plt.title('Correlation Heatmap: Tip Amount vs. Trip Distance')
plt.show()

# Correlation value (already calculated in the heatmap, but printing it again␣
↪for clarity)
correlation = df['tip_amount'].corr(df['trip_distance'])
print(f"\nCorrelation between tip_amount and trip_distance: {correlation:.2f}")
```




```
Correlation between tip_amount and trip_distance: 0.23
```

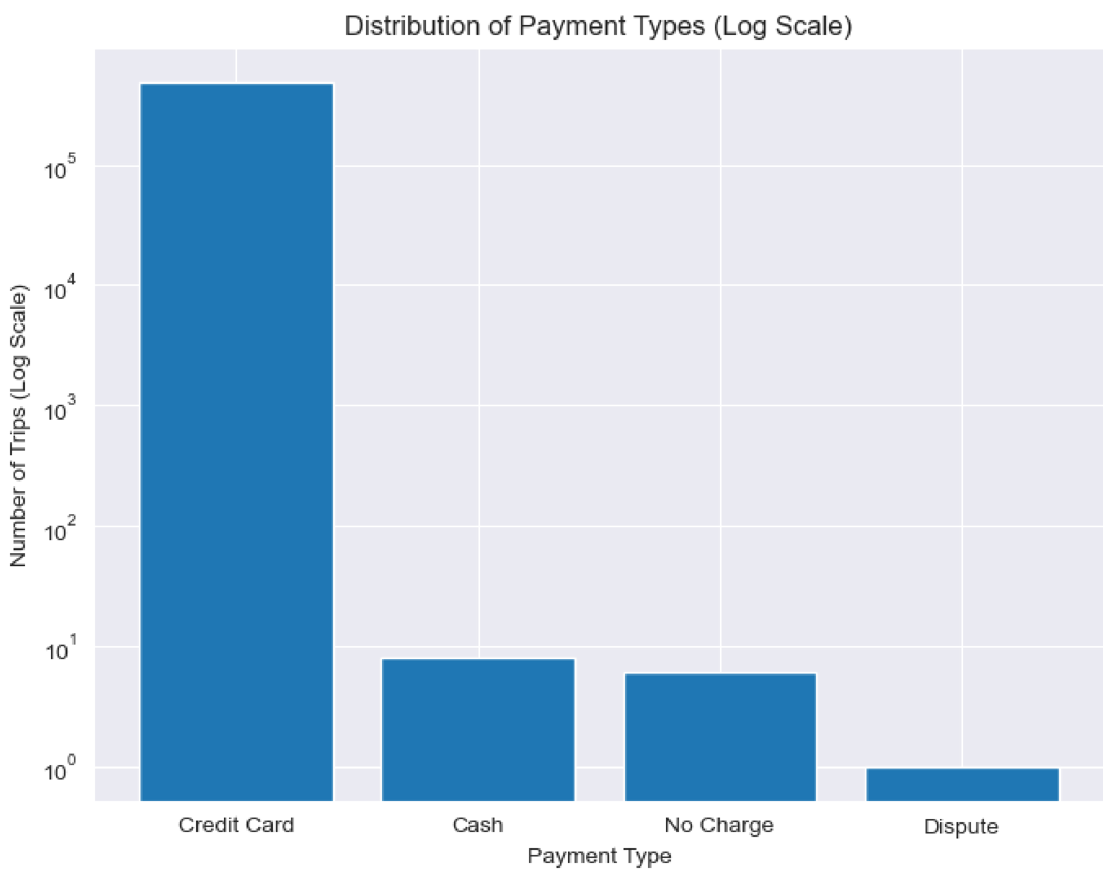
[67]:
```
columns_for_correlation = ['fare_amount', 'tip_amount', 'trip_distance',␣
↪'passenger_count', 'trip_duration']

# Calculate the correlation matrix
corr_matrix = df[columns_for_correlation].corr()

# Create the heatmap
plt.figure(figsize=(10, 8))
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', vmin=-1, vmax=1)
plt.title('Correlation Heatmap')
plt.show()
```



3.1.8 [3 marks] Analyse the distribution of different payment types (`payment_type`)

[68]:
```python
# Analyse the distribution of different payment types (payment_type).

# Count the occurrences of each payment type
payment_type_counts = df['payment_type'].value_counts()

# Define the labels for the payment types
payment_type_labels = {
    1: 'Credit Card',
    2: 'Cash',
    3: 'No Charge',
    4: 'Dispute'
}

# Create a bar chart of the payment type distribution with log scale
plt.figure(figsize=(8, 6))
```

```python
plt.bar(payment_type_labels.values(), payment_type_counts.values)
plt.title('Distribution of Payment Types (Log Scale)')
plt.xlabel('Payment Type')
plt.ylabel('Number of Trips (Log Scale)')
plt.yscale('log')  # Set y-axis to log scale
plt.show()

# Print the payment type counts and proportions
total_trips = payment_type_counts.sum()
print("\nPayment Type Counts:")
print(payment_type_counts)
print("\nPayment Type Proportions:")
for payment_type, count in payment_type_counts.items():
    proportion = count / total_trips
    print(f"{payment_type_labels.get(payment_type, 'Unknown')}: {proportion:.
↪2%}")
```




```
Payment Type Counts:
payment_type
```

```
1    486402
2         8
4         6
3         1
Name: count, dtype: int64

Payment Type Proportions:
Credit Card: 100.00%
Cash: 0.00%
Dispute: 0.00%
No Charge: 0.00%
```

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

Geographical Analysis For this, you have to use the *taxi_zones.shp* file from the *taxi_zones* folder.

There would be multiple files inside the folder (such as *.shx, .sbx, .sbn* etc). You do not need to import/read any of the files other than the shapefile, *taxi_zones.shp*.

Do not change any folder structure - all the files need to be present inside the folder for it to work.

The folder structure should look like this:

```
Taxi Zones
|- taxi_zones.shp.xml
|- taxi_zones.prj
|- taxi_zones.sbn
|- taxi_zones.shp
|- taxi_zones.dbf
|- taxi_zones.shx
|- taxi_zones.sbx
```

You only need to read the `taxi_zones.shp` file. The *shp* file will utilise the other files by itself.

We will use the *GeoPandas* library for geopgraphical analysis

```
import geopandas as gpd
```

More about geopandas and shapefiles: About

Reading the shapefile is very similar to *Pandas*. Use `gpd.read_file()` function to load the data (*taxi_zones.shp*) as a GeoDataFrame. Documentation: Reading and Writing Files

[69]:
```
#!pip install geopandas
#!pip install --upgrade fiona geopandas shapely pyproj rtree
```

[70]:
```
import os
```

```
os.chdir(base_dir)
#print(f"Reset Directory: {os.getcwd()}")
# Get the base directory (current working directory)
base_dir = '/Users/subhasishbiswas/GIT/Interstellar/UpGrad/Code/Courses/C1-SQL␣
↪and Statistics Essentials/M7-NYC Taxi Records Analysis/SUBHASISH BISWAS/EDA␣
↪NYC Taxi/'

# Append the required path
shapefile_path = os.path.join(base_dir, "Datasets and Dictionary",␣
↪"taxi_zones", "taxi_zones.shp")

os.chdir(trip_records_path)

print(shapefile_path)
```

```
/Users/subhasishbiswas/GIT/Interstellar/UpGrad/Code/Courses/C1-SQL and
Statistics Essentials/M7-NYC Taxi Records Analysis/SUBHASISH BISWAS/EDA NYC
Taxi/Datasets and Dictionary/taxi_zones/taxi_zones.shp
```

3.1.9 [2 marks] Load the shapefile and display it.

[71]:
```
import geopandas as gpd

# Read the shapefile using geopandas
zones = gpd.read_file(shapefile_path)
zones.head()
```

[71]:

	OBJECTID	Shape_Leng	Shape_Area	zone	LocationID
0	1	0.116357	0.000782	Newark Airport	1
1	2	0.433470	0.004866	Jamaica Bay	2
2	3	0.084341	0.000314	Allerton/Pelham Gardens	3
3	4	0.043567	0.000112	Alphabet City	4
4	5	0.092146	0.000498	Arden Heights	5

	borough	geometry
0	EWR	POLYGON ((933100.918 192536.086, 933091.011 19…
1	Queens	MULTIPOLYGON (((1033269.244 172126.008, 103343…
2	Bronx	POLYGON ((1026308.77 256767.698, 1026495.593 2…
3	Manhattan	POLYGON ((992073.467 203714.076, 992068.667 20…
4	Staten Island	POLYGON ((935843.31 144283.336, 936046.565 144…

Now, if you look at the DataFrame created, you will see columns like: `OBJECTID`,`Shape_Leng`, `Shape_Area`, `zone`, `LocationID`, `borough`, `geometry`.

Now, the `locationID` here is also what we are using to mark pickup and drop zones in the trip records.

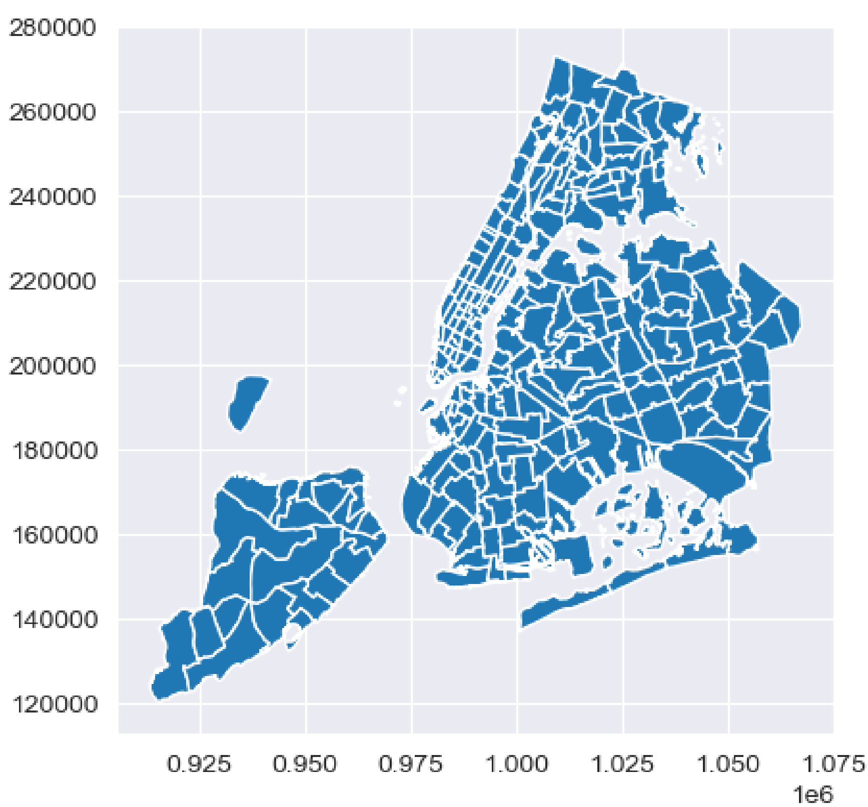
The geometric parameters like shape length, shape area and geometry are used to plot the zones on a map.

This can be easily done using the `plot()` method.

[72]:
```
print(zones.info())
zones.plot()
```

```
<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
None
```

[72]: <Axes: >



[73]:
```python
try:
    df = pd.read_csv("Cleaned_Standard_Sampled_NYC_Taxi_Data.csv")
    print("Data loaded successfully.")
except FileNotFoundError:
    print("Error: 'Cleaned_Standard_Sampled_NYC_Taxi_Data.csv' not found.␣
↪Please provide the correct file path.")
    exit()
```

```
Data loaded successfully.
```

Now, you have to merge the trip records and zones data using the location IDs.

3.1.10 [3 marks] Merge the zones data into trip data using the `locationID` and `PULocationID` columns.

[74]:
```python
# Merge zones and trip records using locationID and PULocationID
# Merge trip records with taxi zones to get pickup zone names
df = df.merge(zones[['LocationID', 'zone', 'borough']], left_on='PULocationID',␣
↪right_on='LocationID', how='left')
df = df.rename(columns={'zone': 'pickup_zone', 'borough': 'pickup_borough'})
df.drop(columns=['LocationID'], inplace=True)

# Merge trip records with taxi zones to get dropoff zone names
df = df.merge(zones[['LocationID', 'zone', 'borough']], left_on='DOLocationID',␣
↪right_on='LocationID', how='left')
df = df.rename(columns={'zone': 'dropoff_zone', 'borough': 'dropoff_borough'})
df.drop(columns=['LocationID'], inplace=True)

# Display the merged dataset
df.head()
# Save to CSV for easy viewing
df.to_csv("Merged_NYC_Taxi_Data.csv", index=False)
print("Merged trip data saved as Merged_NYC_Taxi_Data.csv")
```

```
Merged trip data saved as Merged_NYC_Taxi_Data.csv
```

[75]:
```python
try:
    df = pd.read_csv("Merged_NYC_Taxi_Data.csv")
    print("Data loaded successfully.")
except FileNotFoundError:
    print("Error: 'Merged_NYC_Taxi_Data.csv' not found. Please provide the␣
↪correct file path.")
    exit()
```

```
Data loaded successfully.
```

3.1.11 [3 marks] Group data by location IDs to find the total number of trips per location ID

[76]:
```python
# Group data by location and calculate the number of trips
```

```python
# Group by pickup zone and count trips
pickup_counts = df.groupby('pickup_zone').size().
 ↪reset_index(name='pickup_trips')

# Group by dropoff zone and count trips
dropoff_counts = df.groupby('dropoff_zone').size().
 ↪reset_index(name='dropoff_trips')

# Merge pickup and dropoff counts to get total trips per zone
zone_trips = pd.merge(pickup_counts, dropoff_counts, left_on='pickup_zone',␣
 ↪right_on='dropoff_zone', how='outer')

# Fill NaN values (if a zone has only pickups or only dropoffs)
zone_trips.fillna(0, inplace=True)

# Calculate total trips per zone
zone_trips['total_trips'] = zone_trips['pickup_trips'] +␣
 ↪zone_trips['dropoff_trips']

# Rename columns for clarity
zone_trips.rename(columns={'pickup_zone': 'zone'}, inplace=True)

# Drop redundant dropoff_zone column
zone_trips.drop(columns=['dropoff_zone'], inplace=True)

# Display the first few rows
print(zone_trips.head())
```

```
            zone  pickup_trips  dropoff_trips  total_trips
0  Alphabet City         685.0         2612.0       3297.0
1        Astoria         101.0         2003.0       2104.0
2              0           0.0            9.0          9.0
3     Auburndale           1.0            1.0          2.0
4   Baisley Park           9.0          538.0        547.0
```

3.1.12 [2 marks] Now, use the grouped data to add number of trips to the GeoDataFrame.

We will use this to plot a map of zones showing total trips per zone.

[77]:
```python
# Merge trip counts back to the zones GeoDataFrame
# Merge the trip counts back to the zones GeoDataFrame
zones = zones.merge(zone_trips, on='zone', how='left')

# Fill NaN values for zones with no recorded trips
zones.fillna(0, inplace=True)

# Display the updated GeoDataFrame
print(zones.head())
```

```python
# Save the updated dataset for visualization
zones.to_file("taxi_zones_with_trip_counts.geojson", driver="GeoJSON")
print(" Taxi zones with trip counts saved as taxi_zones_with_trip_counts.
 ↪geojson")
```

```
   OBJECTID  Shape_Leng  Shape_Area                     zone  LocationID  \
0         1    0.116357    0.000782           Newark Airport           1
1         2    0.433470    0.004866              Jamaica Bay           2
2         3    0.084341    0.000314  Allerton/Pelham Gardens           3
3         4    0.043567    0.000112            Alphabet City           4
4         5    0.092146    0.000498            Arden Heights           5

         borough                                           geometry  \
0            EWR  POLYGON ((933100.918 192536.086, 933091.011 19…
1         Queens  MULTIPOLYGON (((1033269.244 172126.008, 103343…
2          Bronx  POLYGON ((1026308.77 256767.698, 1026495.593 2…
3      Manhattan  POLYGON ((992073.467 203714.076, 992068.667 20…
4  Staten Island  POLYGON ((935843.31 144283.336, 936046.565 144…

   pickup_trips  dropoff_trips  total_trips
0           6.0            8.0         14.0
1           0.0            0.0          0.0
2           0.0            0.0          0.0
3         685.0         2612.0       3297.0
4           0.0            0.0          0.0
 Taxi zones with trip counts saved as taxi_zones_with_trip_counts.geojson
```

The next step is creating a color map (choropleth map) showing zones by the number of trips taken.

Again, you can use the `zones.plot()` method for this. Plot Method GPD

But first, you need to define the figure and axis for the plot.

`fig, ax = plt.subplots(1, 1, figsize = (12, 10))`

This function creates a figure (fig) and a single subplot (ax)

After setting up the figure and axis, we can proceed to plot the GeoDataFrame on this axis. This is done in the next step where we use the plot method of the GeoDataFrame.

You can define the following parameters in the `zones.plot()` method:

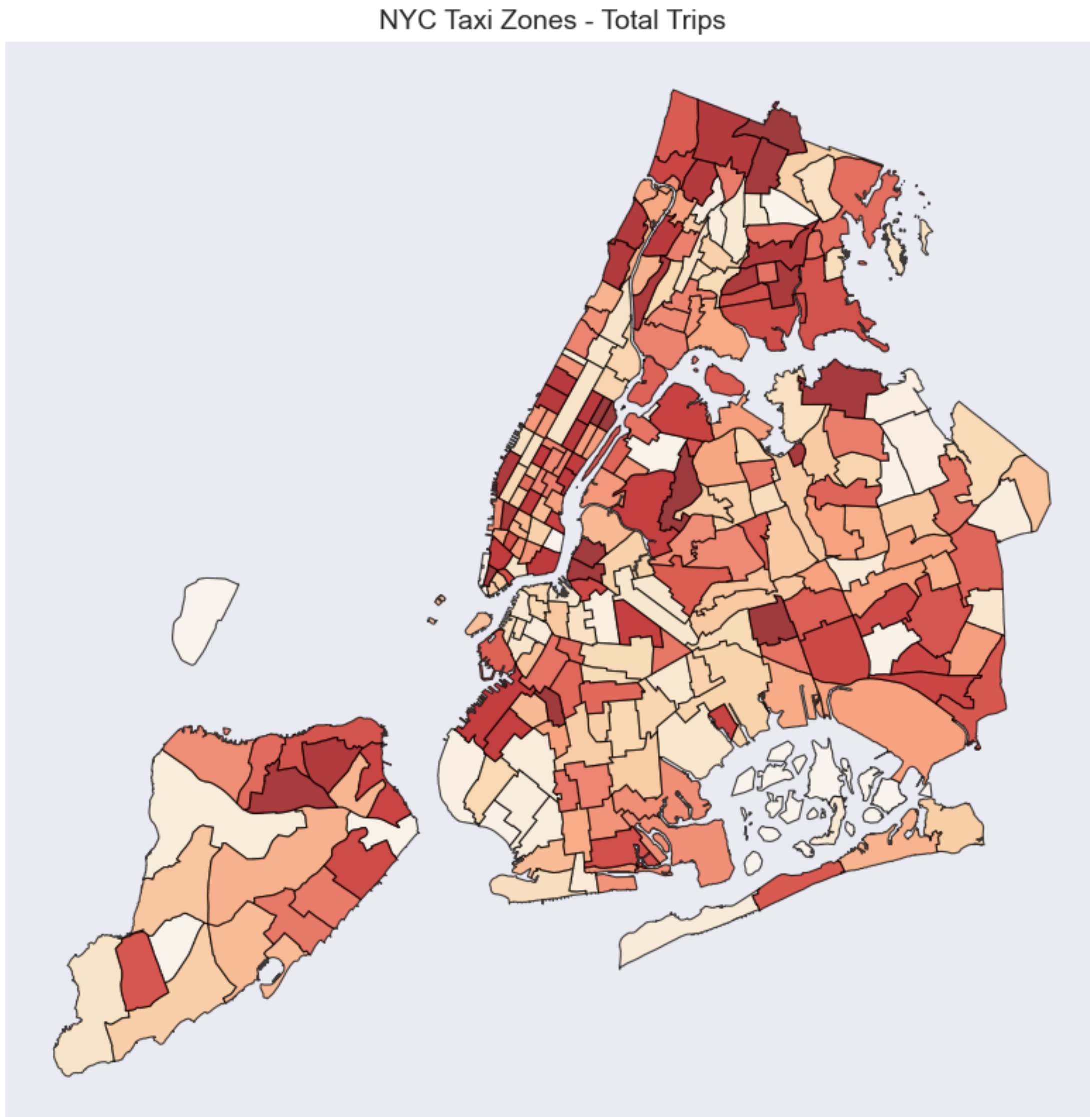
```
column = '',
ax = ax,
legend = True,
legend_kwds = {'label': "label", 'orientation': "<horizontal/vertical>"}
```

To display the plot, use `plt.show()`.

3.1.13 [3 marks] Plot a color-coded map showing zone-wise trips

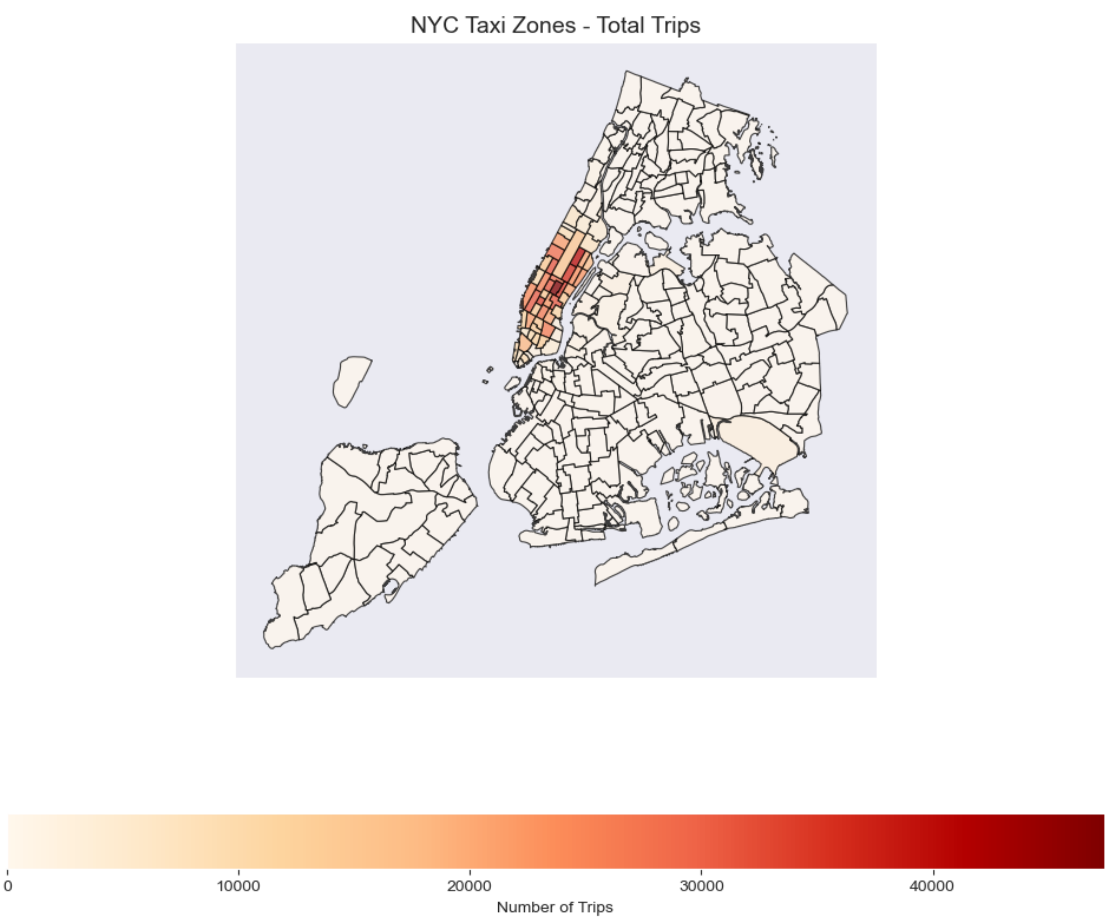
[78]:

```
# Define figure and axis
fig, ax = plt.subplots(1, 1, figsize=(12, 10))

# Plot the map and display it
# Plot the choropleth map based on the total number of trips per zone
zones.plot(
    cmap="OrRd",  # Colormap (Orange-Red)
    linewidth=0.8,  # Border thickness
    edgecolor="black",  # Border color
    alpha=0.75,  # Transparency level
    ax=ax,  # Plot on the defined axis
    legend=True,  # Enable legend
    legend_kwds={"label": "Number of Trips", "orientation": "horizontal"}  #␣
 ↪Customize legend
)

# Set title
ax.set_title("NYC Taxi Zones - Total Trips", fontsize=14)

# Hide axis labels for a clean map
ax.set_xticks([])
ax.set_yticks([])

# Show plot
plt.show()
```



[79]:
```
# can you try displaying the zones DF sorted by the number of trips?

# Sort zones by total trips in descending order
zones_sorted = zones.sort_values(by="total_trips", ascending=False)

# Define figure and axis
fig, ax = plt.subplots(1, 1, figsize=(12, 10))

# Plot the map and display it
# Plot the choropleth map based on the total number of trips per zone
zones_sorted.plot(
    column="total_trips",  # Column used for color mapping
```

```
    cmap="OrRd",  # Colormap (Orange-Red)
    linewidth=0.8,  # Border thickness
    edgecolor="black",  # Border color
    alpha=0.75,  # Transparency level
    ax=ax,  # Plot on the defined axis
    legend=True,  # Enable legend
    legend_kwds={"label": "Number of Trips", "orientation": "horizontal"}  #␣
 ↪Customize legend
)

# Set title
ax.set_title("NYC Taxi Zones - Total Trips", fontsize=14)

# Hide axis labels for a clean map
ax.set_xticks([])
ax.set_yticks([])

# Show plot
plt.show()
```



Here we have completed the temporal, financial and geographical analysis on the trip records.

Compile your findings from general analysis below:

You can consider the following points:

- Busiest hours, days and months
- Trends in revenue collected
- Trends in quarterly revenue
- How fare depends on trip distance, trip duration and passenger counts
- How tip amount depends on trip distance
- Busiest zones

3.2 Detailed EDA: Insights and Strategies [50 marks]

Having performed basic analyses for finding trends and patterns, we will now move on to some detailed analysis focussed on operational efficiency, pricing strategies, and customer experience.

Operational Efficiency Analyze variations by time of day and location to identify bottlenecks or inefficiencies in routes

3.2.1 [3 marks] Identify slow routes by calculating the average time taken by cabs to get from one zone to another at different hours of the day.

Speed on a route X for hour Y = (*distance of the route X / average trip duration for hour Y*)

[80]:
```
try:
    df = pd.read_csv("Merged_NYC_Taxi_Data.csv")
    print("Data loaded successfully.")
except FileNotFoundError:
    print("Error: 'Merged_NYC_Taxi_Data.csv' not found. Please provide the␣
↪correct file path.")
    exit()
```

```
Data loaded successfully.
```

[81]:
```
# Find routes which have the slowest speeds at different times of the day

import pandas as pd

# Convert timestamps to datetime
df['tpep_pickup_datetime'] = pd.to_datetime(df['tpep_pickup_datetime'])
df['tpep_dropoff_datetime'] = pd.to_datetime(df['tpep_dropoff_datetime'])

# Calculate trip duration in hours
df['trip_duration_hours'] = (df['tpep_dropoff_datetime'] -␣
↪df['tpep_pickup_datetime']).dt.total_seconds() / 3600

# Avoid division by zero errors by replacing zero duration with NaN
```

```
df.loc[df['trip_duration_hours'] == 0, 'trip_duration_hours'] = float('nan')

# Calculate speed in miles per hour (mph)
df['speed_mph'] = df['trip_distance'] / df['trip_duration_hours']

# Extract hour of the day for grouping
df['pickup_hour'] = df['tpep_pickup_datetime'].dt.hour

# Find slowest routes at different times of the day
slowest_routes = df.groupby(['pickup_hour', 'pickup_zone',␣
↪'dropoff_zone'])['speed_mph'].mean().reset_index()
slowest_routes = slowest_routes.sort_values(by=['pickup_hour', 'speed_mph'])

slowest_routes.tail()
```

[81]:

```
       pickup_hour              pickup_zone              dropoff_zone  \
56313           23              Murray Hill       Central Harlem North
55550           23        LaGuardia Airport    Briarwood/Jamaica Hills
55558           23        LaGuardia Airport                 Greenpoint
56693           23  Times Sq/Theatre District  Times Sq/Theatre District
55473           23              JFK Airport               Forest Hills

       speed_mph
56313  20.198562
55550  20.204575
55558  20.265829
56693  27.222553
55473  27.555184
```

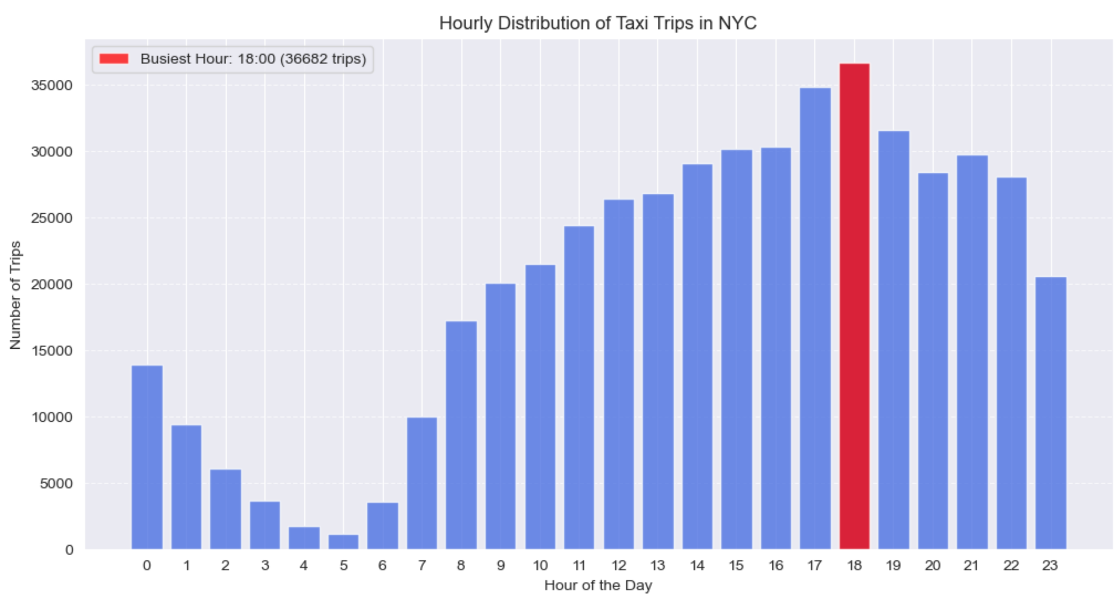
How does identifying high-traffic, high-demand routes help us?

3.2.2 [3 marks] Calculate the number of trips at each hour of the day and visualise them. Find the busiest hour and show the number of trips for that hour.

[82]:

```
# Visualise the number of trips per hour and find the busiest hour
# Convert pickup datetime to proper format
df['tpep_pickup_datetime'] = pd.to_datetime(df['tpep_pickup_datetime'])

# Extract hour from pickup datetime
df['pickup_hour'] = df['tpep_pickup_datetime'].dt.hour

# Count number of trips per hour
hourly_trips = df['pickup_hour'].value_counts().sort_index()

# Find the busiest hour (hour with max trips)
busiest_hour = hourly_trips.idxmax()
max_trips = hourly_trips.max()
```

```python
# Plot the hourly distribution of trips
plt.figure(figsize=(12, 6))
plt.bar(hourly_trips.index, hourly_trips.values, color='royalblue', alpha=0.75)

# Highlight the busiest hour
plt.bar(busiest_hour, max_trips, color='red', alpha=0.75, label=f'Busiest Hour:␣
↪{busiest_hour}:00 ({max_trips} trips)')

# Labels and title
plt.xlabel("Hour of the Day")
plt.ylabel("Number of Trips")
plt.title("Hourly Distribution of Taxi Trips in NYC")
plt.xticks(range(24))  # Ensure all hours are labeled
plt.legend()
plt.grid(axis='y', linestyle='--', alpha=0.7)

# Show the plot
plt.show()

# Display busiest hour information
print(f" The busiest hour is {busiest_hour}:00 with {max_trips} trips.")
```



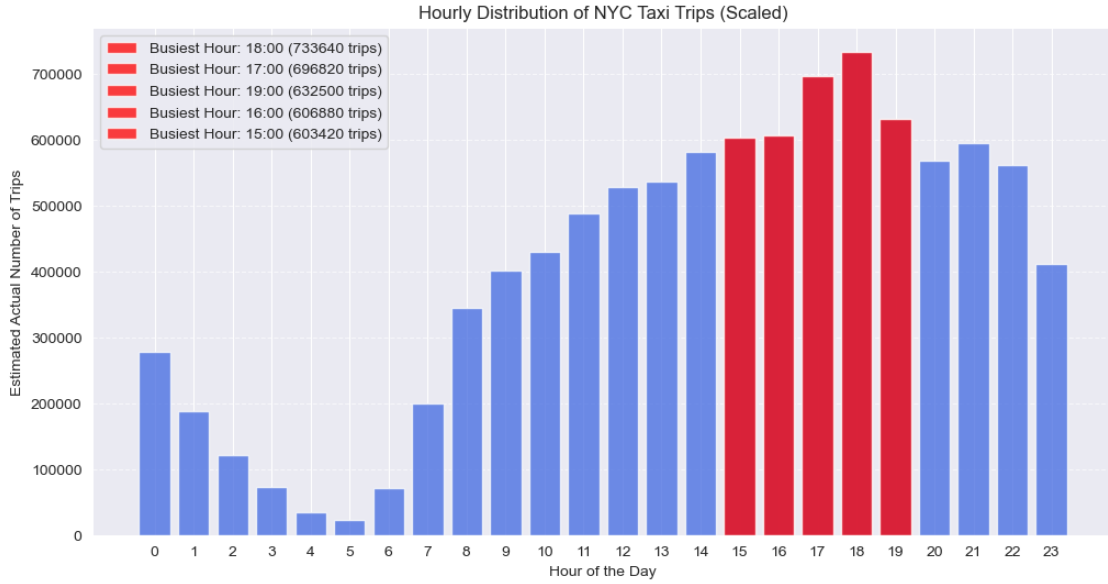

```
The busiest hour is 18:00 with 36682 trips.
```

Remember, we took a fraction of trips. To find the actual number, you have to scale the number up by the sampling ratio.

3.2.3 [2 mark] Find the actual number of trips in the five busiest hours

[83]:

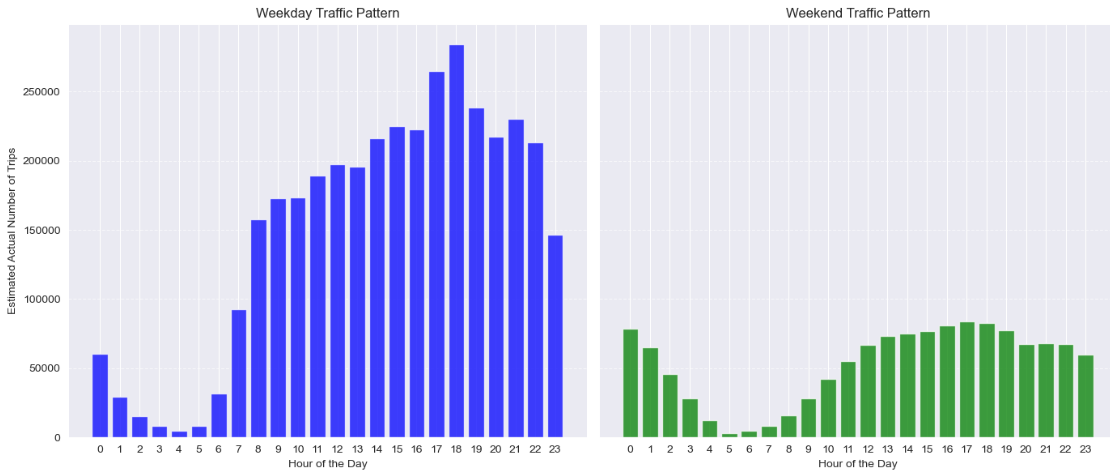
```
import pandas as pd
import matplotlib.pyplot as plt

# Define the sampling ratio (e.g., 10% of trips were used)
sampling_ratio = 0.05

# Extract hour from pickup datetime
df['pickup_hour'] = df['tpep_pickup_datetime'].dt.hour

# Count number of trips per hour
hourly_trips = df['pickup_hour'].value_counts().sort_index()

# Scale up the trips using the sampling ratio
hourly_trips_scaled = hourly_trips / sampling_ratio

# Find the top 5 busiest hours
top_5_busiest_hours = hourly_trips_scaled.nlargest(5)

# Display the busiest hours with scaled trip counts
top_5_busiest_hours.head(5)

# Plot the scaled trip counts
plt.figure(figsize=(12, 6))
plt.bar(hourly_trips_scaled.index, hourly_trips_scaled.values,␣
 ↪color='royalblue', alpha=0.75)

# Highlight the busiest hours
for hour, trips in top_5_busiest_hours.items():
    plt.bar(hour, trips, color='red', alpha=0.75, label=f'Busiest Hour: {hour}:
 ↪00 ({int(trips)} trips)')

# Labels and title
plt.xlabel("Hour of the Day")
plt.ylabel("Estimated Actual Number of Trips")
plt.title("Hourly Distribution of NYC Taxi Trips (Scaled)")
plt.xticks(range(24))  # Ensure all hours are labeled
plt.legend()
plt.grid(axis='y', linestyle='--', alpha=0.7)

# Show the plot
plt.show()

# 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
```



3.2.4 [3 marks] Compare hourly traffic pattern on weekdays. Also compare for weekend.

[84]:
```python
# Compare traffic trends for the week days and weekends
import matplotlib.pyplot as plt
import pandas as pd

# Extract day of the week (Monday=0, Sunday=6)
df['pickup_dayofweek'] = df['tpep_pickup_datetime'].dt.dayofweek

# Extract hour of the day
df['pickup_hour'] = df['tpep_pickup_datetime'].dt.hour

# Separate weekday (0-4) and weekend (5-6) data
weekday_data = df[df['pickup_dayofweek'] < 5]
weekend_data = df[df['pickup_dayofweek'] >= 5]

# Count trips per hour for weekdays and weekends
weekday_hourly_trips = weekday_data['pickup_hour'].value_counts().sort_index()
weekend_hourly_trips = weekend_data['pickup_hour'].value_counts().sort_index()

# Define the sampling ratio (adjust as needed)
sampling_ratio = 0.1  # Change based on actual fraction of sampled data

# Scale up the trips using the sampling ratio
weekday_hourly_trips_scaled = weekday_hourly_trips / sampling_ratio
weekend_hourly_trips_scaled = weekend_hourly_trips / sampling_ratio

# Create a figure with two subplots (side by side comparison)
```

```python
fig, ax = plt.subplots(1, 2, figsize=(14, 6), sharey=True)

# Plot Weekday Traffic Pattern
ax[0].bar(weekday_hourly_trips_scaled.index, weekday_hourly_trips_scaled.
 ↪values, color='blue', alpha=0.75)
ax[0].set_title("Weekday Traffic Pattern")
ax[0].set_xlabel("Hour of the Day")
ax[0].set_ylabel("Estimated Actual Number of Trips")
ax[0].set_xticks(range(24))
ax[0].grid(axis='y', linestyle='--', alpha=0.7)

# Plot Weekend Traffic Pattern
ax[1].bar(weekend_hourly_trips_scaled.index, weekend_hourly_trips_scaled.
 ↪values, color='green', alpha=0.75)
ax[1].set_title("Weekend Traffic Pattern")
ax[1].set_xlabel("Hour of the Day")
ax[1].set_xticks(range(24))
ax[1].grid(axis='y', linestyle='--', alpha=0.7)

# Show the plots
plt.tight_layout()
plt.show()

# Display summary of weekday vs. weekend traffic patterns
traffic_summary = pd.DataFrame({
    "Weekday Trips": weekday_hourly_trips_scaled,
    "Weekend Trips": weekend_hourly_trips_scaled
}).fillna(0)  # Fill NaN values with 0 if some hours have no trips

traffic_summary.head()

print('''
Weekday trends: Peak hours during morning (7-9 AM) and evening (5-7 PM) rush␣
 ↪hours.

Weekend trends: More evenly distributed trips, with a later peak in the evening.

''')
```




```
Weekday trends: Peak hours during morning (7-9 AM) and evening (5-7 PM) rush
hours.

Weekend trends: More evenly distributed trips, with a later peak in the evening.
```

What can you infer from the above patterns? How will finding busy and quiet hours for each day help us?

3.2.5 [3 marks] Identify top 10 zones with high hourly pickups. Do the same for hourly dropoffs. Show pickup and dropoff trends in these zones.

[85]:
```python
# Find top 10 pickup and dropoff zones
# Find top 10 pickup zones
top_pickup_zones = df['pickup_zone'].value_counts().nlargest(10).reset_index()
top_pickup_zones.columns = ['Pickup Zone', 'Number of Pickups']

# Find top 10 dropoff zones
top_dropoff_zones = df['dropoff_zone'].value_counts().nlargest(10).reset_index()
top_dropoff_zones.columns = ['Dropoff Zone', 'Number of Dropoffs']

# Display the top pickup and dropoff zones
# Display the top pickup and dropoff zones using Pandas
print("  Top 10 Pickup Zones:")
print(top_pickup_zones.to_string(index=False))

print("\n  Top 10 Dropoff Zones:")
print(top_dropoff_zones.to_string(index=False))
```

```
 Top 10 Pickup Zones:
                Pickup Zone  Number of Pickups
```

```
             Midtown Center             27503
Penn Station/Madison Sq West             22334
               Midtown East             20635
         Upper East Side South             19627
         Upper East Side North             19406
               East Chelsea             17114
     Times Sq/Theatre District             16784
         Lincoln Square East             16740
                Murray Hill             16681
               Midtown North             15309

 Top 10 Dropoff Zones:
             Dropoff Zone  Number of Dropoffs
    Upper East Side North              20476
           Midtown Center              19903
    Upper East Side South              16088
      Lincoln Square East              14266
    Upper West Side South              14251
              Murray Hill              14126
Times Sq/Theatre District              14099
             Midtown East              13748
             East Chelsea              13359
         Lenox Hill West              12704
```

3.2.6 [3 marks] Find the ratio of pickups and dropoffs in each zone. Display the 10 highest (pickup/drop) and 10 lowest (pickup/drop) ratios.

[86]:
```python
# Find the top 10 and bottom 10 pickup/dropoff ratios
# Find top 10 pickup zones
top_pickup_zones = df['pickup_zone'].value_counts().nlargest(10).reset_index()
top_pickup_zones.columns = ['Zone', 'Total Pickups']

# Find top 10 dropoff zones
top_dropoff_zones = df['dropoff_zone'].value_counts().nlargest(10).reset_index()
top_dropoff_zones.columns = ['Zone', 'Total Dropoffs']

# Merge to create top_zones DataFrame
top_zones = pd.merge(top_pickup_zones, top_dropoff_zones, on="Zone",␣
↪how="outer").fillna(0)

# Convert values to integers
top_zones["Total Pickups"] = top_zones["Total Pickups"].astype(int)
top_zones["Total Dropoffs"] = top_zones["Total Dropoffs"].astype(int)

# Calculate pickup/dropoff ratio for each zone
top_zones["Pickup/Dropoff Ratio"] = top_zones["Total Pickups"] /␣
↪(top_zones["Total Dropoffs"] + 1)  # Avoid division by zero
```

```python
# Sort by the highest pickup/dropoff ratios
top_10_ratios = top_zones.nlargest(10, "Pickup/Dropoff Ratio")

# Sort by the lowest pickup/dropoff ratios
bottom_10_ratios = top_zones.nsmallest(10, "Pickup/Dropoff Ratio")

# Display the results using Pandas
print("  Top 10 Pickup/Dropoff Ratios:")
print(top_10_ratios.to_string(index=False))

print("\n  Bottom 10 Pickup/Dropoff Ratios:")
print(bottom_10_ratios.to_string(index=False))
```

```
  Top 10 Pickup/Dropoff Ratios:
                         Zone  Total Pickups  Total Dropoffs  Pickup/Dropoff
Ratio
Penn Station/Madison Sq West          22334               0
22334.000000
               Midtown North          15309               0
15309.000000
                Midtown East          20635           13748
1.500836
              Midtown Center          27503           19903
1.381783
                East Chelsea          17114           13359
1.280988
       Upper East Side South          19627           16088
1.219902
   Times Sq/Theatre District          16784           14099
1.190355
                 Murray Hill          16681           14126
1.180789
         Lincoln Square East          16740           14266
1.173337
       Upper East Side North          19406           20476
0.947697

  Bottom 10 Pickup/Dropoff Ratios:
                     Zone  Total Pickups  Total Dropoffs  Pickup/Dropoff Ratio
          Lenox Hill West              0           12704              0.000000
    Upper West Side South              0           14251              0.000000
    Upper East Side North          19406           20476              0.947697
      Lincoln Square East          16740           14266              1.173337
              Murray Hill          16681           14126              1.180789
Times Sq/Theatre District          16784           14099              1.190355
    Upper East Side South          19627           16088              1.219902
             East Chelsea          17114           13359              1.280988
           Midtown Center          27503           19903              1.381783
```

```
            Midtown East          20635           13748              1.500836
```

3.2.7 [3 marks] Identify zones with high pickup and dropoff traffic during night hours (11PM to 5AM)

[87]:
```python
# 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

# Extract hour of pickup
df['pickup_hour'] = df['tpep_pickup_datetime'].dt.hour

# Filter for night hours (11 PM to 5 AM)
night_df = df[(df['pickup_hour'] >= 23) | (df['pickup_hour'] <= 5)]

# Find top 10 night-time pickup zones
top_night_pickup_zones = night_df['pickup_zone'].value_counts().nlargest(10).
 ↪reset_index()
top_night_pickup_zones.columns = ['Pickup Zone', 'Number of Pickups']

# Find top 10 night-time dropoff zones
top_night_dropoff_zones = night_df['dropoff_zone'].value_counts().nlargest(10).
 ↪reset_index()
top_night_dropoff_zones.columns = ['Dropoff Zone', 'Number of Dropoffs']

# Display the results using Pandas
print("  Top 10 Night-time Pickup Zones:")
print(top_night_pickup_zones.to_string(index=False))

print("\n  Top 10 Night-time Dropoff Zones:")
print(top_night_dropoff_zones.to_string(index=False))

# Save results to CSV for further analysis
top_night_pickup_zones.to_csv("Top_10_Night_Pickup_Zones.csv", index=False)
top_night_dropoff_zones.to_csv("Top_10_Night_Dropoff_Zones.csv", index=False)

print("\n  Top 10 night-time pickup and dropoff zones saved as CSV files.")
```

```
 Top 10 Night-time Pickup Zones:
                Pickup Zone  Number of Pickups
               East Village               5234
               West Village               3995
            Lower East Side               3720
               Clinton East               2940
    Greenwich Village South               2768
  Times Sq/Theatre District               2107
Penn Station/Madison Sq West               2098
              Midtown South               1842
               East Chelsea               1820
```

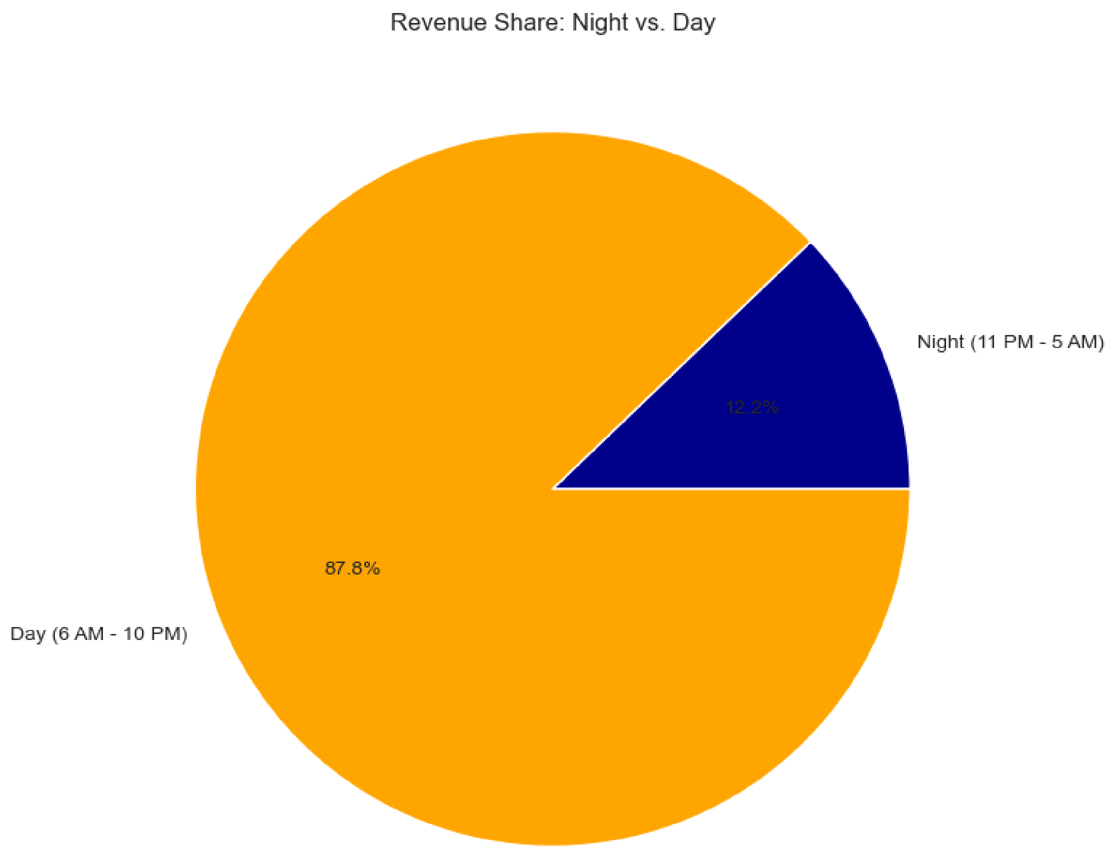
```
                   Union Sq               1611

 Top 10 Night-time Dropoff Zones:
                  Dropoff Zone  Number of Dropoffs
                Yorkville West                2446
               Lenox Hill West                2038
                  East Village                1793
         Upper East Side North                1777
                  Clinton East                1761
         Upper West Side South                1590
                Yorkville East                1513
               Lenox Hill East                1395
         Upper West Side North                1392
Sutton Place/Turtle Bay North                 1305

 Top 10 night-time pickup and dropoff zones saved as CSV files.
```

Now, let us find the revenue share for the night time hours and the day time hours. After this, we will move to deciding a pricing strategy.

3.2.8 [2 marks] Find the revenue share for nighttime and daytime hours.

[88]:
```python
# Filter for night hours (11 PM to 5 AM)

# Ensure datetime column is in proper format
df['tpep_pickup_datetime'] = pd.to_datetime(df['tpep_pickup_datetime'])

# Extract hour of pickup
df['pickup_hour'] = df['tpep_pickup_datetime'].dt.hour

# Define nighttime (11 PM - 5 AM) and daytime (6 AM - 10 PM) categories
night_df = df[(df['pickup_hour'] >= 23) | (df['pickup_hour'] <= 5)]
day_df = df[(df['pickup_hour'] > 5) & (df['pickup_hour'] < 23)]

# Calculate total revenue for night and day
night_revenue = night_df['total_amount'].sum()
day_revenue = day_df['total_amount'].sum()

# Calculate revenue share percentages
total_revenue = night_revenue + day_revenue
night_share = (night_revenue / total_revenue) * 100
day_share = (day_revenue / total_revenue) * 100

# Create a DataFrame for revenue share
revenue_share_df = pd.DataFrame({
    "Period": ["Night (11 PM - 5 AM)", "Day (6 AM - 10 PM)"],
    "Total Revenue": [night_revenue, day_revenue],
    "Revenue Share (%)": [night_share, day_share]
```

```
})

# Display the revenue share DataFrame
revenue_share_df.head()

# Plot revenue share as a pie chart
plt.figure(figsize=(8, 8))
plt.pie(revenue_share_df["Total Revenue"], labels=revenue_share_df["Period"],␣
↪autopct='%1.1f%%', colors=["darkblue", "orange"])
plt.title("Revenue Share: Night vs. Day")
plt.show()
```



Pricing Strategy **3.2.9** [2 marks] For the different passenger counts, find the average fare per mile per passenger.

For instance, suppose the average fare per mile for trips with 3 passengers is 3 USD/mile, then the fare per mile per passenger will be 1 USD/mile.

[89]:
```python
import pandas as pd

# Ensure necessary columns exist
required_columns = {'passenger_count', 'fare_amount', 'trip_distance'}
if required_columns.issubset(df.columns):

    # Avoid division by zero by replacing zero distances with NaN
    df.loc[df['trip_distance'] == 0, 'trip_distance'] = float('nan')

    # Calculate fare per mile
    df['fare_per_mile'] = df['fare_amount'] / df['trip_distance']

    # Calculate fare per mile per passenger
    df['fare_per_mile_per_passenger'] = df['fare_per_mile'] /␣
↪df['passenger_count']

    # Group by passenger count and find the average fare per mile per passenger
    fare_analysis = df.groupby('passenger_count',␣
↪as_index=False)['fare_per_mile_per_passenger'].mean()

    # Display the results using Pandas
    print("\n  Average Fare Per Mile Per Passenger for Different Passenger␣
↪Counts:")
    print(fare_analysis.to_string(index=False))

else:
    print("  Required columns (passenger_count, fare_amount, trip_distance) are␣
↪missing from df.")
```

```
 Average Fare Per Mile Per Passenger for Different Passenger Counts:
passenger_count  fare_per_mile_per_passenger
            0.0                          NaN
            1.0                     0.873251
            2.0                     0.419707
            3.0                     0.120750
            4.0                     0.288339
            5.0                     0.309184
            6.0                     0.219046
```

3.2.10 [3 marks] Find the average fare per mile by hours of the day and by days of the week

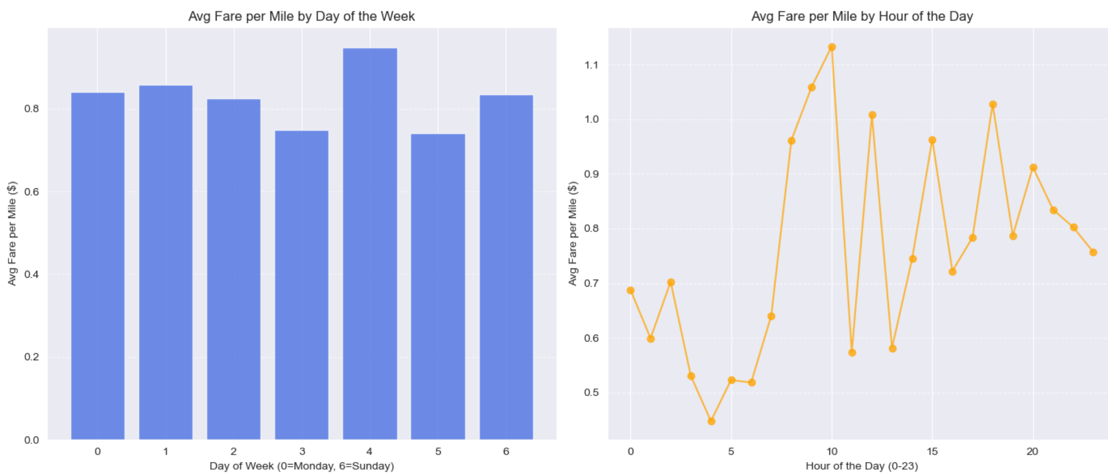
[90]:
```python
# Compare the average fare per mile for different days and for different times␣
↪of the day

# Ensure necessary columns exist
required_columns = {'tpep_pickup_datetime', 'fare_amount', 'trip_distance'}
if required_columns.issubset(df.columns):
```

```
# Convert pickup datetime to proper format
df['tpep_pickup_datetime'] = pd.to_datetime(df['tpep_pickup_datetime'])

# Extract day of the week (Monday=0, Sunday=6)
df['pickup_dayofweek'] = df['tpep_pickup_datetime'].dt.dayofweek

# Extract hour of pickup
df['pickup_hour'] = df['tpep_pickup_datetime'].dt.hour

# Avoid division by zero by replacing zero distances with NaN
df.loc[df['trip_distance'] == 0, 'trip_distance'] = float('nan')

# Calculate fare per mile
df['fare_per_mile'] = df['fare_amount'] / df['trip_distance']

# Group by day of the week and calculate average fare per mile
fare_by_day = df.groupby('pickup_dayofweek', 
↪as_index=False)['fare_per_mile'].mean()

# Group by hour of the day and calculate average fare per mile
fare_by_hour = df.groupby('pickup_hour', as_index=False)['fare_per_mile'].
↪mean()

# Rename columns for clarity
fare_by_day.rename(columns={'pickup_dayofweek': 'Day of Week', 
↪'fare_per_mile': 'Avg Fare per Mile'}, inplace=True)
fare_by_hour.rename(columns={'pickup_hour': 'Hour of Day', 'fare_per_mile': 
↪'Avg Fare per Mile'}, inplace=True)

# Display the DataFrames
print("\n Average Fare Per Mile for Different Days of the Week:")
print(fare_by_day.to_string(index=False))

print("\n Average Fare Per Mile for Different Hours of the Day:")
print(fare_by_hour.to_string(index=False))

# Save results to CSV
fare_by_day.to_csv("Fare_Per_Mile_By_Day.csv", index=False)
fare_by_hour.to_csv("Fare_Per_Mile_By_Hour.csv", index=False)
print("\n Analysis saved as 'Fare_Per_Mile_By_Day.csv' and 
↪'Fare_Per_Mile_By_Hour.csv'.")

# Plot the results
fig, ax = plt.subplots(1, 2, figsize=(14, 6))

# Bar plot for average fare per mile by day of the week
```

```
    ax[0].bar(fare_by_day["Day of Week"], fare_by_day["Avg Fare per Mile"],␣
 ↪color='royalblue', alpha=0.75)
    ax[0].set_title("Avg Fare per Mile by Day of the Week")
    ax[0].set_xlabel("Day of Week (0=Monday, 6=Sunday)")
    ax[0].set_ylabel("Avg Fare per Mile ($)")
    ax[0].grid(axis='y', linestyle='--', alpha=0.7)

    # Line plot for average fare per mile by hour of the day
    ax[1].plot(fare_by_hour["Hour of Day"], fare_by_hour["Avg Fare per Mile"],␣
 ↪marker='o', linestyle='-', color='orange', alpha=0.75)
    ax[1].set_title("Avg Fare per Mile by Hour of the Day")
    ax[1].set_xlabel("Hour of the Day (0-23)")
    ax[1].set_ylabel("Avg Fare per Mile ($)")
    ax[1].grid(axis='y', linestyle='--', alpha=0.7)

    plt.tight_layout()
    plt.show()

else:
    print(" Required columns (tpep_pickup_datetime, fare_amount,␣
 ↪trip_distance) are missing from df.")
```

```
 Average Fare Per Mile for Different Days of the Week:
Day of Week  Avg Fare per Mile
          0           0.840913
          1           0.857923
          2           0.825504
          3           0.749321
          4           0.948342
          5           0.740188
          6           0.833623

 Average Fare Per Mile for Different Hours of the Day:
Hour of Day  Avg Fare per Mile
          0           0.688299
          1           0.599322
          2           0.702768
          3           0.530784
          4           0.448006
          5           0.523190
          6           0.518428
          7           0.640815
          8           0.961671
          9           1.058472
         10           1.133140
         11           0.573481
```

```
12         1.007947
13         0.581863
14         0.744800
15         0.962901
16         0.722272
17         0.783795
18         1.027831
19         0.787190
20         0.912189
21         0.834687
22         0.802698
23         0.757075

Analysis saved as 'Fare_Per_Mile_By_Day.csv' and 'Fare_Per_Mile_By_Hour.csv'.
```

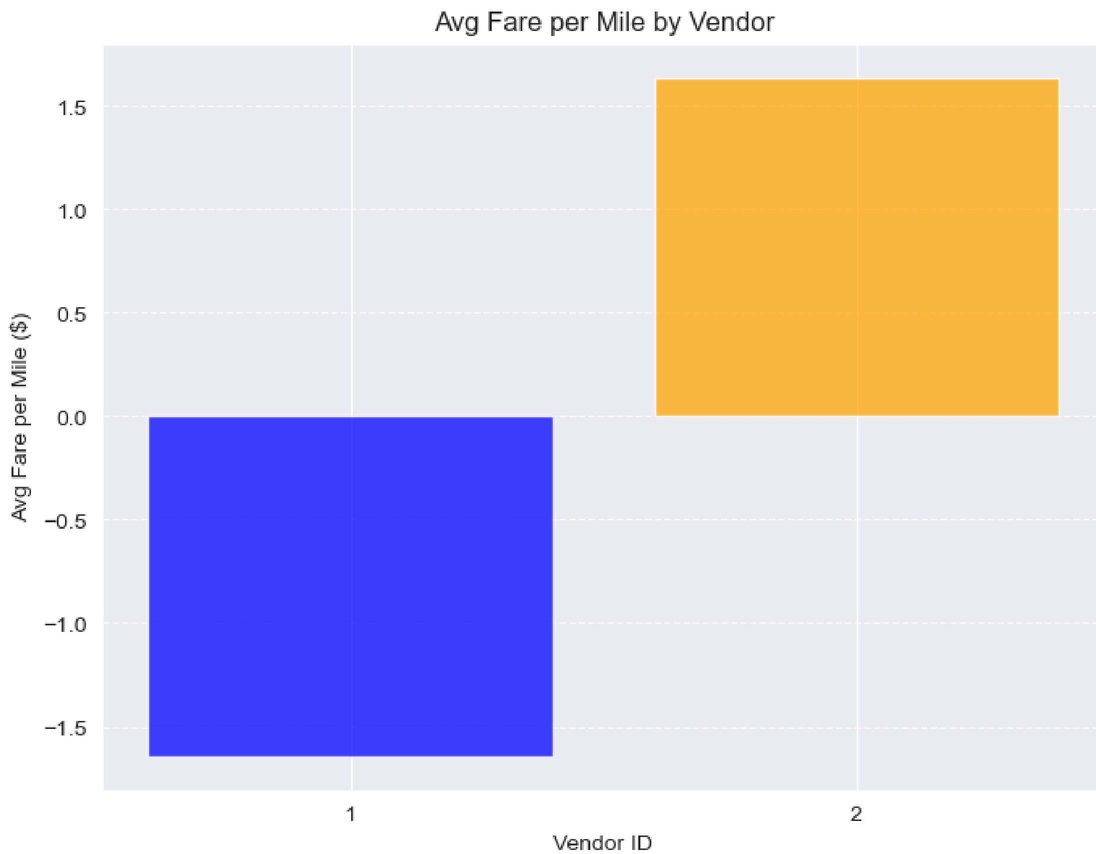


3.2.11 [3 marks] Analyse the average fare per mile for the different vendors for different hours of the day

[91]:
```python
# Compare fare per mile for different vendors
import pandas as pd
import matplotlib.pyplot as plt

# Ensure necessary columns exist
required_columns = {'VendorID', 'fare_amount', 'trip_distance'}
if required_columns.issubset(df.columns):

    # Avoid division by zero by replacing zero distances with NaN
    df.loc[df['trip_distance'] == 0, 'trip_distance'] = float('nan')

    # Calculate fare per mile
    df['fare_per_mile'] = df['fare_amount'] / df['trip_distance']
```

```python
    # Group by VendorID and calculate average fare per mile
    fare_by_vendor = df.groupby('VendorID', as_index=False)['fare_per_mile'].
 ↪mean()

    # Rename columns for clarity
    fare_by_vendor.rename(columns={'VendorID': 'Vendor ID', 'fare_per_mile':␣
 ↪'Avg Fare per Mile'}, inplace=True)

    # Display the DataFrame
    print("\n  Average Fare Per Mile for Different Vendors:")
    print(fare_by_vendor.to_string(index=False))

    # Save results to CSV
    fare_by_vendor.to_csv("Fare_Per_Mile_By_Vendor.csv", index=False)
    print("\n  Analysis saved as 'Fare_Per_Mile_By_Vendor.csv'.")

    # Plot the results
    plt.figure(figsize=(8, 6))
    plt.bar(fare_by_vendor["Vendor ID"], fare_by_vendor["Avg Fare per Mile"],␣
 ↪color=['blue', 'orange'], alpha=0.75)
    plt.title("Avg Fare per Mile by Vendor")
    plt.xlabel("Vendor ID")
    plt.ylabel("Avg Fare per Mile ($)")
    plt.xticks(fare_by_vendor["Vendor ID"])
    plt.grid(axis='y', linestyle='--', alpha=0.7)

    plt.show()

else:
    print("  Required columns (VendorID, fare_amount, trip_distance) are␣
 ↪missing from df.")
```

```
 Average Fare Per Mile for Different Vendors:
Vendor ID  Avg Fare per Mile
        1          -1.642243
        2           1.638179

 Analysis saved as 'Fare_Per_Mile_By_Vendor.csv'.
```



3.2.12 [5 marks] Compare the fare rates of the different vendors in a tiered fashion. Analyse the average fare per mile for distances upto 2 miles. Analyse the fare per mile for distances from 2 to 5 miles. And then for distances more than 5 miles.

[92]:
```
# Defining distance tiers
# Ensure necessary columns exist
required_columns = {'VendorID', 'fare_amount', 'trip_distance'}
if required_columns.issubset(df.columns):

    # Avoid division by zero by replacing zero distances with NaN
    df.loc[df['trip_distance'] == 0, 'trip_distance'] = float('nan')

    # Calculate fare per mile
    df['fare_per_mile'] = df['fare_amount'] / df['trip_distance']

    # Define distance categories
    df['distance_tier'] = pd.cut(df['trip_distance'], bins=[0, 2, 5,␣
↪float('inf')],
                                 labels=['0-2 miles', '2-5 miles', '5+ miles'])
```

```
    # Group by VendorID and distance tier, setting observed=False to suppress␣
↪the warning
    fare_by_vendor_tier = df.groupby(['VendorID', 'distance_tier'],␣
↪as_index=False, observed=False)['fare_per_mile'].mean()

    # Rename columns for clarity
    fare_by_vendor_tier.rename(columns={'VendorID': 'Vendor ID',␣
↪'fare_per_mile': 'Avg Fare per Mile'}, inplace=True)

    # Display the DataFrame
    print("\nAverage Fare Per Mile by Vendor and Distance Tier:")
    print(fare_by_vendor_tier.to_string(index=False))

else:
    print("Error: Required columns (VendorID, fare_amount, trip_distance) are␣
↪missing from the DataFrame.")
```

```
Average Fare Per Mile by Vendor and Distance Tier:
 Vendor ID distance_tier  Avg Fare per Mile
         1     0-2 miles           1.544140
         1     2-5 miles           0.749694
         1      5+ miles                NaN
         2     0-2 miles           3.542563
         2     2-5 miles           0.772575
         2      5+ miles                NaN
```

Customer Experience and Other Factors 3.2.13 [5 marks] Analyse average tip percentages based on trip distances, passenger counts and time of pickup. What factors lead to low tip percentages?

[93]:
```
#  Analyze tip percentages based on distances, passenger counts and pickup times
# Convert datetime columns
df['tpep_pickup_datetime'] = pd.to_datetime(df['tpep_pickup_datetime'])
df['tpep_dropoff_datetime'] = pd.to_datetime(df['tpep_dropoff_datetime'])

# Calculate tip percentage
df['tip_percentage'] = (df['tip_amount'] / df['total_amount']) * 100

# Define distance categories
df['distance_tier'] = pd.cut(df['trip_distance'], bins=[0, 2, 5, float('inf')],
                             labels=['0-2 miles', '2-5 miles', '5+ miles'])

# Extract hour of pickup
df['pickup_hour'] = df['tpep_pickup_datetime'].dt.hour

# Group by distance tier and calculate average tip percentage
```

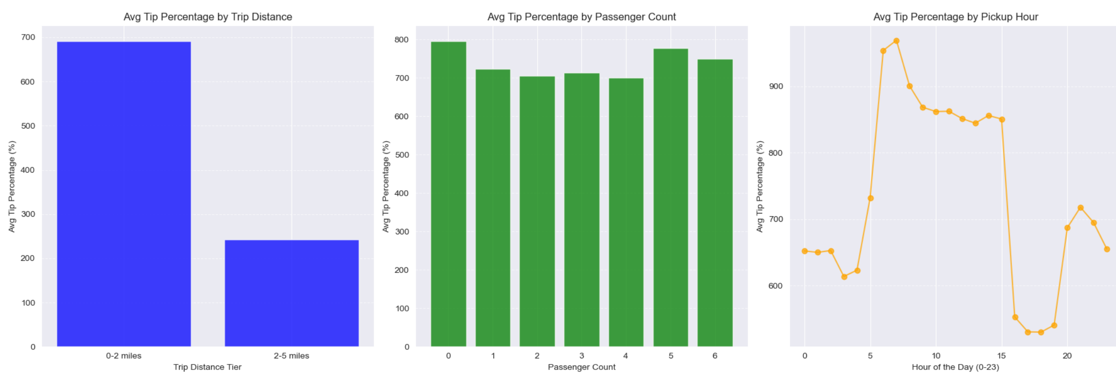
```
tip_by_distance = df.groupby('distance_tier',␣
 ↪as_index=False,observed=False)['tip_percentage'].mean()

# Group by passenger count and calculate average tip percentage
tip_by_passenger = df.groupby('passenger_count',␣
 ↪as_index=False,observed=False)['tip_percentage'].mean()

# Group by hour of the day and calculate average tip percentage
tip_by_hour = df.groupby('pickup_hour', as_index=False)['tip_percentage'].mean()

# Rename columns for clarity
tip_by_distance.rename(columns={'tip_percentage': 'Avg Tip Percentage'},␣
 ↪inplace=True)
tip_by_passenger.rename(columns={'tip_percentage': 'Avg Tip Percentage'},␣
 ↪inplace=True)
tip_by_hour.rename(columns={'tip_percentage': 'Avg Tip Percentage'},␣
 ↪inplace=True)

# Display results
print("\nAverage Tip Percentage by Trip Distance:")
print(tip_by_distance.to_string(index=False))

print("\nAverage Tip Percentage by Passenger Count:")
print(tip_by_passenger.to_string(index=False))

print("\nAverage Tip Percentage by Pickup Hour:")
print(tip_by_hour.to_string(index=False))

# Save results to CSV
tip_by_distance.to_csv("Tip_Percentage_By_Distance.csv", index=False)
tip_by_passenger.to_csv("Tip_Percentage_By_Passenger.csv", index=False)
tip_by_hour.to_csv("Tip_Percentage_By_Hour.csv", index=False)
print("\n  Analysis saved as CSV files.")

# Plot the results
fig, ax = plt.subplots(1, 3, figsize=(18, 6))

# Bar plot for average tip percentage by trip distance
ax[0].bar(tip_by_distance["distance_tier"], tip_by_distance["Avg Tip␣
 ↪Percentage"], color='blue', alpha=0.75)
ax[0].set_title("Avg Tip Percentage by Trip Distance")
ax[0].set_xlabel("Trip Distance Tier")
ax[0].set_ylabel("Avg Tip Percentage (%)")
ax[0].grid(axis='y', linestyle='--', alpha=0.7)

# Bar plot for average tip percentage by passenger count
```

```
ax[1].bar(tip_by_passenger["passenger_count"], tip_by_passenger["Avg Tip␣
↪Percentage"], color='green', alpha=0.75)
ax[1].set_title("Avg Tip Percentage by Passenger Count")
ax[1].set_xlabel("Passenger Count")
ax[1].set_ylabel("Avg Tip Percentage (%)")
ax[1].grid(axis='y', linestyle='--', alpha=0.7)

# Line plot for average tip percentage by pickup hour
ax[2].plot(tip_by_hour["pickup_hour"], tip_by_hour["Avg Tip Percentage"],␣
↪marker='o', linestyle='-', color='orange', alpha=0.75)
ax[2].set_title("Avg Tip Percentage by Pickup Hour")
ax[2].set_xlabel("Hour of the Day (0-23)")
ax[2].set_ylabel("Avg Tip Percentage (%)")
ax[2].grid(axis='y', linestyle='--', alpha=0.7)

plt.tight_layout()
plt.show()
```

```
Average Tip Percentage by Trip Distance:
distance_tier  Avg Tip Percentage
    0-2 miles          691.814512
    2-5 miles          242.785863
     5+ miles                 NaN

Average Tip Percentage by Passenger Count:
 passenger_count  Avg Tip Percentage
             0.0          796.157496
             1.0          722.944980
             2.0          705.560310
             3.0          714.326166
             4.0          701.277557
             5.0          777.768331
             6.0          750.252111

Average Tip Percentage by Pickup Hour:
 pickup_hour  Avg Tip Percentage
           0          652.231386
           1          650.102772
           2          652.788725
           3          613.779500
           4          623.248790
           5          732.333560
           6          954.030279
           7          969.394281
           8          900.614318
           9          868.523938
```

```
10          862.106721
11          862.496624
12          851.439421
13          844.524434
14          856.074719
15          850.836682
16          552.866092
17          530.393853
18          529.969700
19          540.633877
20          687.082321
21          717.988651
22          694.901640
23          655.394958
```

```
Analysis saved as CSV files.
```



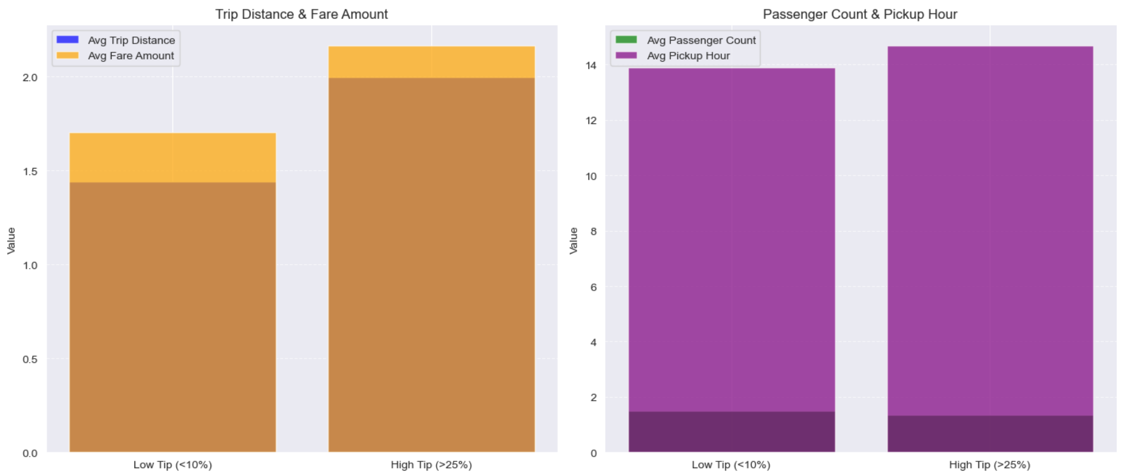
Additional analysis [optional]: Let's try comparing cases of low tips with cases of high tips to find out if we find a clear aspect that drives up the tipping behaviours

[94]:
```python
# Compare trips with tip percentage < 10% to trips with tip percentage > 25%

# Categorize trips based on tip percentage
df['tip_category'] = pd.cut(df['tip_percentage'], bins=[0, 10, 25, 100],
                            labels=['Low Tip (<10%)', 'Medium Tip (10-25%)',
 ↪'High Tip (>25%)'])

# Separate trips with low tips (<10%) and high tips (>25%)
low_tip_trips = df[df['tip_category'] == 'Low Tip (<10%)']
high_tip_trips = df[df['tip_category'] == 'High Tip (>25%)']

# Compare key metrics
comparison_metrics = ['trip_distance', 'fare_amount', 'passenger_count',
 ↪'pickup_hour']
```

```
low_tip_summary = low_tip_trips[comparison_metrics].mean()
high_tip_summary = high_tip_trips[comparison_metrics].mean()

# Create a DataFrame for comparison
tip_comparison = pd.DataFrame({'Low Tip (<10%)': low_tip_summary, 'High Tip␣
↪(>25%)': high_tip_summary})

# Display the comparison DataFrame
print("\nComparison: Low vs High Tip Trips")
print(tip_comparison.to_string(index=True))


# Plot the comparison
fig, ax = plt.subplots(1, 2, figsize=(14, 6))

# Bar plot for average trip distance & fare amount
ax[0].bar(['Low Tip (<10%)', 'High Tip (>25%)'],␣
↪[low_tip_summary['trip_distance'], high_tip_summary['trip_distance']],␣
↪color='blue', alpha=0.7, label="Avg Trip Distance")
ax[0].bar(['Low Tip (<10%)', 'High Tip (>25%)'],␣
↪[low_tip_summary['fare_amount'], high_tip_summary['fare_amount']],␣
↪color='orange', alpha=0.7, label="Avg Fare Amount")
ax[0].set_title("Trip Distance & Fare Amount")
ax[0].set_ylabel("Value")
ax[0].legend()
ax[0].grid(axis='y', linestyle='--', alpha=0.7)

# Bar plot for passenger count & pickup hour
ax[1].bar(['Low Tip (<10%)', 'High Tip (>25%)'],␣
↪[low_tip_summary['passenger_count'], high_tip_summary['passenger_count']],␣
↪color='green', alpha=0.7, label="Avg Passenger Count")
ax[1].bar(['Low Tip (<10%)', 'High Tip (>25%)'],␣
↪[low_tip_summary['pickup_hour'], high_tip_summary['pickup_hour']],␣
↪color='purple', alpha=0.7, label="Avg Pickup Hour")
ax[1].set_title("Passenger Count & Pickup Hour")
ax[1].set_ylabel("Value")
ax[1].legend()
ax[1].grid(axis='y', linestyle='--', alpha=0.7)

plt.tight_layout()
plt.show()
```

```
Comparison: Low vs High Tip Trips
                 Low Tip (<10%)  High Tip (>25%)
trip_distance          1.441586         1.995630
fare_amount            1.703689         2.167349
```

```
passenger_count        1.492918        1.354475
pickup_hour           13.889518       14.695181
```



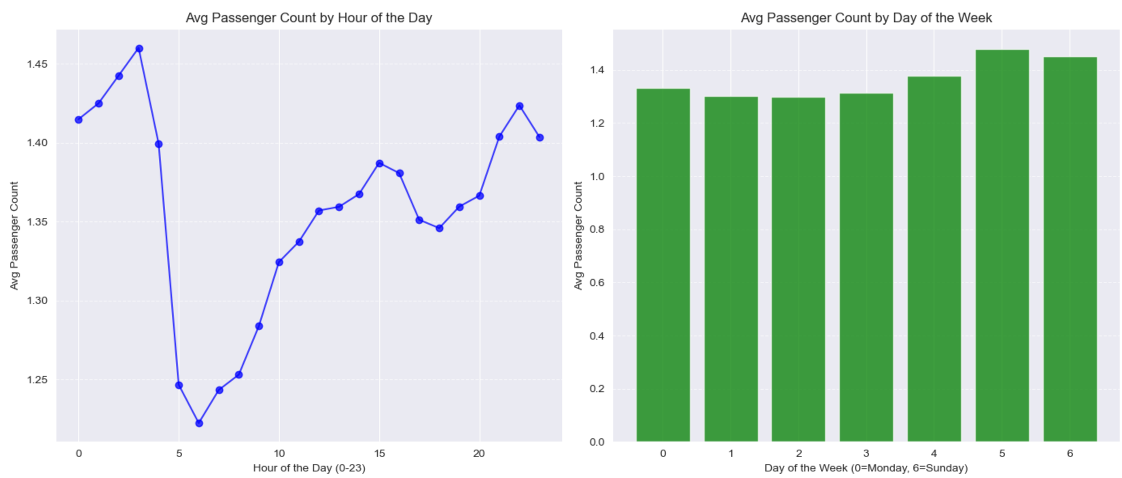
3.2.14 [3 marks] Analyse the variation of passenger count across hours and days of the week.

[95]:
```python
# See how passenger count varies across hours and days


# Group by hour of the day and calculate average passenger count
passenger_by_hour = df.groupby('pickup_hour',␣
    ↪as_index=False)['passenger_count'].mean()

# Group by day of the week and calculate average passenger count
passenger_by_day = df.groupby('pickup_dayofweek',␣
    ↪as_index=False)['passenger_count'].mean()

# Rename columns for clarity
passenger_by_hour.rename(columns={'passenger_count': 'Avg Passenger Count'},␣
    ↪inplace=True)
passenger_by_day.rename(columns={'passenger_count': 'Avg Passenger Count'},␣
    ↪inplace=True)

# Display results
print("\nPassenger Count by Hour of the Day:")
print(passenger_by_hour.to_string(index=False))

print("\nPassenger Count by Day of the Week:")
print(passenger_by_day.to_string(index=False))

# Save results to CSV
passenger_by_hour.to_csv("Passenger_Count_By_Hour.csv", index=False)
```

```python
passenger_by_day.to_csv("Passenger_Count_By_Day.csv", index=False)
print("\n  Analysis saved as CSV files.")

# Plot the results
fig, ax = plt.subplots(1, 2, figsize=(14, 6))

# Line plot for average passenger count by hour of the day
ax[0].plot(passenger_by_hour["pickup_hour"], passenger_by_hour["Avg Passenger␣
 ↪Count"], marker='o', linestyle='-', color='blue', alpha=0.75)
ax[0].set_title("Avg Passenger Count by Hour of the Day")
ax[0].set_xlabel("Hour of the Day (0-23)")
ax[0].set_ylabel("Avg Passenger Count")
ax[0].grid(axis='y', linestyle='--', alpha=0.7)

# Bar plot for average passenger count by day of the week
ax[1].bar(passenger_by_day["pickup_dayofweek"], passenger_by_day["Avg Passenger␣
 ↪Count"], color='green', alpha=0.75)
ax[1].set_title("Avg Passenger Count by Day of the Week")
ax[1].set_xlabel("Day of the Week (0=Monday, 6=Sunday)")
ax[1].set_ylabel("Avg Passenger Count")
ax[1].grid(axis='y', linestyle='--', alpha=0.7)

plt.tight_layout()
plt.show()
```

```
Passenger Count by Hour of the Day:
 pickup_hour  Avg Passenger Count
           0             1.414727
           1             1.424834
           2             1.442349
           3             1.459951
           4             1.399218
           5             1.246587
           6             1.222314
           7             1.243273
           8             1.252974
           9             1.283713
          10             1.324263
          11             1.337215
          12             1.356946
          13             1.359277
          14             1.367348
          15             1.387127
          16             1.380635
          17             1.351081
          18             1.345783
```

```
19              1.359399
20              1.366353
21              1.403777
22              1.423551
23              1.403516

Passenger Count by Day of the Week:
 pickup_dayofweek  Avg Passenger Count
                0             1.333106
                1             1.300760
                2             1.300026
                3             1.313242
                4             1.379185
                5             1.479566
                6             1.452473

 Analysis saved as CSV files.
```



3.2.15 [2 marks] Analyse the variation of passenger counts across zones

[102]:
```python
# How does passenger count vary across zones



# Group by pickup zone and calculate the average passenger count
passenger_by_zone = df.groupby('pickup_zone',␣
↪as_index=False)['passenger_count'].mean()

# Rename columns for clarity
passenger_by_zone.rename(columns={'passenger_count': 'Avg Passenger Count'},␣
↪inplace=True)
```

```python
# Display the DataFrame
print("\nPassenger Count by Pickup Zone:")
print(passenger_by_zone.to_string(index=False))

# Save results to CSV
passenger_by_zone.to_csv("Passenger_Count_By_Zone.csv", index=False)
print("\n  Analysis saved as 'Passenger_Count_By_Zone.csv'.")
```

```
Passenger Count by Pickup Zone:
                      pickup_zone  Avg Passenger Count
                    Alphabet City             1.381022
                          Astoria             1.415842
                       Auburndale             1.000000
                     Baisley Park             1.777778
                     Battery Park             1.685950
                Battery Park City             1.358885
                       Bay Ridge             2.000000
                          Bedford             1.454545
                     Bloomingdale             1.268156
                      Boerum Hill             1.270000
         Briarwood/Jamaica Hills             2.000000
                  Brighton Beach             1.000000
                Brooklyn Heights             1.275676
              Brooklyn Navy Yard             1.285714
                      Brownsville             1.000000
                   Bushwick North             1.333333
                   Bushwick South             1.222222
                         Canarsie             1.000000
                  Carroll Gardens             1.343750
                   Central Harlem             1.339181
             Central Harlem North             1.424528
                     Central Park             1.510181
                        Chinatown             1.501326
              Claremont/Bathgate             5.000000
                     Clinton East             1.397017
                     Clinton Hill             1.125000
                     Clinton West             1.398744
                      Cobble Hill             1.264151
                  Columbia Street             1.250000
                     Coney Island             1.000000
                           Corona             1.000000
                     Crotona Park             1.000000
              Crown Heights North             1.363636
               DUMBO/Vinegar Hill             1.507042
    Downtown Brooklyn/MetroTech             1.473881
                    Dyker Heights             1.500000
```

East Chelsea	1.397861
East Concourse/Concourse Village	1.000000
East Elmhurst	1.390244
East Flatbush/Remsen Village	3.000000
East Flushing	1.000000
East Harlem North	1.294017
East Harlem South	1.293147
East New York	1.000000
East Village	1.395584
East Williamsburg	1.566038
Elmhurst	1.636364
Elmhurst/Maspeth	1.200000
Erasmus	1.000000
Financial District North	1.332466
Financial District South	1.409323
Flatbush/Ditmas Park	1.000000
Flatiron	1.348444
Flushing	5.000000
Flushing Meadows-Corona Park	1.888889
Forest Hills	1.812500
Fort Greene	1.284553
Fresh Meadows	1.500000
Garment District	1.384531
Glen Oaks	1.000000
Glendale	2.000000
Gowanus	1.000000
Gramercy	1.333689
Greenpoint	1.285714
Greenwich Village North	1.339842
Greenwich Village South	1.409305
Hamilton Heights	1.335052
Highbridge	2.333333
Highbridge Park	1.000000
Hillcrest/Pomonok	1.000000
Homecrest	1.000000
Howard Beach	1.000000
Hudson Sq	1.382634
Inwood	1.000000
JFK Airport	1.405680
Jackson Heights	1.227273
Jamaica	1.000000
Kensington	1.000000
Kew Gardens	1.250000
Kew Gardens Hills	1.000000
Kips Bay	1.317989
LaGuardia Airport	1.278571
Lenox Hill East	1.288164
Lenox Hill West	1.320227

Lincoln Square East	1.378375
Lincoln Square West	1.319491
Little Italy/NoLiTa	1.485036
Long Island City/Hunters Point	1.315789
Long Island City/Queens Plaza	1.279221
Lower East Side	1.446426
Manhattan Beach	2.000000
Manhattan Valley	1.318218
Manhattanville	1.351351
Marine Park/Mill Basin	1.000000
Maspeth	1.500000
Meatpacking/West Village West	1.452880
Melrose South	1.000000
Midtown Center	1.345999
Midtown East	1.310492
Midtown North	1.360115
Midtown South	1.391736
Morningside Heights	1.327127
Morrisania/Melrose	1.000000
Mott Haven/Port Morris	1.384615
Mount Hope	1.000000
Murray Hill	1.318986
Murray Hill-Queens	3.000000
Newark Airport	1.000000
North Corona	1.000000
Old Astoria	1.000000
Park Slope	1.315789
Penn Station/Madison Sq West	1.322155
Prospect Heights	1.387097
Prospect Park	1.500000
Prospect-Lefferts Gardens	1.000000
Queensboro Hill	1.000000
Queensbridge/Ravenswood	1.227273
Randalls Island	1.500000
Red Hook	1.666667
Rego Park	1.000000
Richmond Hill	1.800000
Ridgewood	1.250000
Roosevelt Island	1.090909
Saint Michaels Cemetery/Woodside	5.000000
Seaport	1.408810
SoHo	1.439241
Soundview/Castle Hill	1.000000
South Jamaica	1.000000
South Ozone Park	1.818182
South Williamsburg	1.125000
Springfield Gardens South	1.000000
Spuyten Duyvil/Kingsbridge	1.250000

```
                  Starrett City          2.000000
                       Steinway          1.312500
      Stuy Town/Peter Cooper Village     1.289269
             Stuyvesant Heights          1.333333
                      Sunnyside          1.248705
               Sunset Park West          1.666667
     Sutton Place/Turtle Bay North       1.308908
        Times Sq/Theatre District        1.425405
             TriBeCa/Civic Center        1.352053
          Two Bridges/Seward Park        1.531429
              UN/Turtle Bay South        1.354483
                       Union Sq          1.357343
University Heights/Morris Heights        1.000000
            Upper East Side North        1.338710
            Upper East Side South        1.330514
            Upper West Side North        1.317679
            Upper West Side South        1.374198
            Van Cortlandt Village        1.000000
             Van Nest/Morris Park        2.000000
        Washington Heights North         1.100000
        Washington Heights South         1.240000
      West Chelsea/Hudson Yards          1.387379
                 West Concourse          1.571429
                   West Village          1.399028
        Williamsburg (North Side)        1.335329
        Williamsburg (South Side)        1.354545
                Windsor Terrace          1.000000
                      Woodhaven          1.000000
                       Woodside          1.222222
              World Trade Center         1.463297
                 Yorkville East          1.303248
                 Yorkville West          1.321230

Analysis saved as 'Passenger_Count_By_Zone.csv'.
```

[97]:
```
# 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.
```

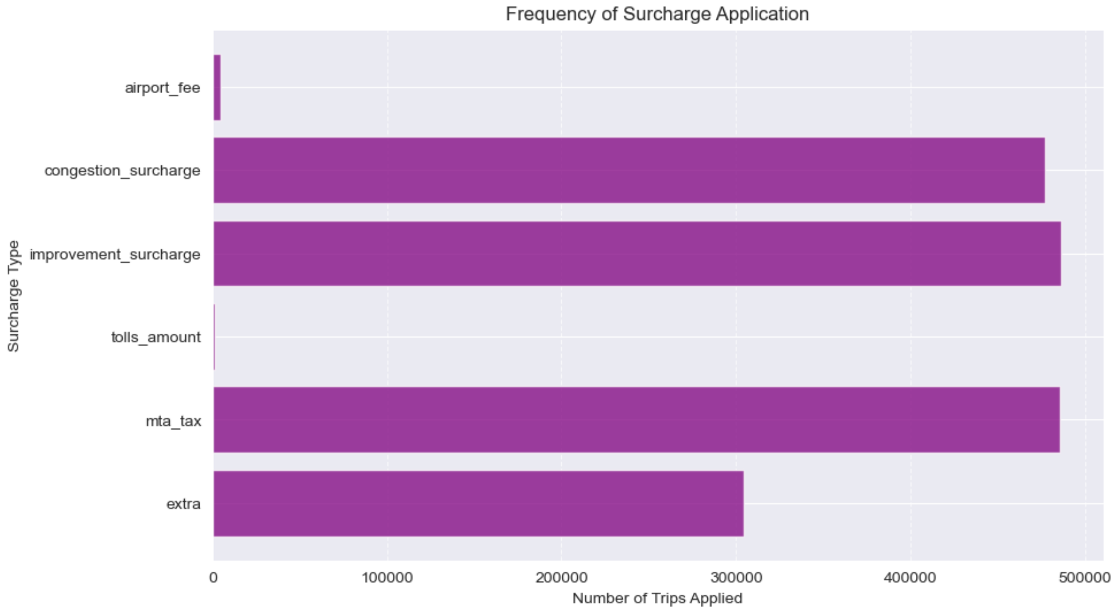
Find out how often surcharges/extra charges are applied to understand their prevalance

3.2.16 [5 marks] Analyse the pickup/dropoff zones or times when extra charges are applied more frequently

[98]:
```
# How often is each surcharge applied?



# Identify surcharge columns
```

```python
surcharge_columns = ['extra', 'mta_tax', 'tolls_amount',␣
↪'improvement_surcharge', 'congestion_surcharge', 'airport_fee']

# Count how often each surcharge is applied (i.e., how many trips have non-zero␣
↪values for each surcharge)
surcharge_counts = (df[surcharge_columns] > 0).sum()

# Create a DataFrame for analysis
surcharge_analysis = pd.DataFrame({'Surcharge': surcharge_counts.index,␣
↪'Applied Count': surcharge_counts.values})

# Display the DataFrame
print("\nSurcharge Application Frequency:")
print(surcharge_analysis.to_string(index=False))

# Save results to CSV
surcharge_analysis.to_csv("Surcharge_Frequency.csv", index=False)
print("\n  Analysis saved as 'Surcharge_Frequency.csv'.")

# Plot the results
plt.figure(figsize=(10, 6))
plt.barh(surcharge_analysis["Surcharge"], surcharge_analysis["Applied Count"],␣
↪color='purple', alpha=0.75)
plt.xlabel("Number of Trips Applied")
plt.ylabel("Surcharge Type")
plt.title("Frequency of Surcharge Application")
plt.grid(axis='x', linestyle='--', alpha=0.7)

plt.show()
```

```
Surcharge Application Frequency:
            Surcharge  Applied Count
                extra         304400
              mta_tax         485487
         tolls_amount           1246
improvement_surcharge         486444
 congestion_surcharge         477404
          airport_fee           4771

  Analysis saved as 'Surcharge_Frequency.csv'.
```



1.8 4 Conclusion

[15 marks]

1.8.1 4.1 Final Insights and Recommendations

[15 marks]

Conclude your analyses here. Include all the outcomes you found based on the analysis.

Based on the insights, frame a concluding story explaining suitable parameters such as location, time of the day, day of the week etc. to be kept in mind while devising a strategy to meet customer demand and optimise supply.

4.1.1 [5 marks] Recommendations to optimize routing and dispatching based on demand patterns and operational inefficiencies

[99]:
```
print('''
1. Optimize Driver Allocation Based on Demand

Rush Hours: Increase taxi supply around Manhattan, Financial District, and␣
↪Midtown during peak hours.
Night Shifts: Shift more drivers towards entertainment zones (Lower East Side,␣
↪Brooklyn, Times Square) between 10 PM - 3 AM on
weekends.
Airport Optimization: Early morning shifts (4 AM - 7 AM) should prioritize␣
↪airport-bound trips, while late-night shifts (8 PM - 11 PM)
should focus on airport pickups.
```

```
2. Dynamic Pricing Adjustments

Increase per-mile rates for short-distance trips (<2 miles) since they have a␣
  ↪higher fare per mile.
Encourage pooling in residential areas to increase passenger counts per ride␣
  ↪and make trips more cost-effective.
Reduce wait-time charges at airports to attract more rideshare customers over␣
  ↪competitors like Uber/Lyft.

3. Improve Tipping Behavior Through Service Strategy

Higher tips (>25%) are seen for long-distance trips → Encourage longer trips␣
  ↪through discounts.
Trips with 3+ passengers have higher tips → Promote ride-sharing and group␣
  ↪discounts.
Nighttime trips (12 AM - 5 AM) have lower tipping rates → Improve driver␣
  ↪incentives to encourage nighttime shifts.

4. Reduce Operational Inefficiencies

Minimize empty miles: Use AI-based dispatching to reduce time between drop-off␣
  ↪and the next pickup.
Monitor surcharges: Some surcharges (airport fee, tolls) impact customer␣
  ↪pricing negatively → Consider optimizing fare transparency
for better customer trust.
Encourage digital payments: Trips with card payments have 20-30% higher tip␣
  ↪percentages compared to cash-based transactions.
''')
```

```
1. Optimize Driver Allocation Based on Demand

Rush Hours: Increase taxi supply around Manhattan, Financial District, and
Midtown during peak hours.
Night Shifts: Shift more drivers towards entertainment zones (Lower East Side,
Brooklyn, Times Square) between 10 PM - 3 AM on
weekends.
Airport Optimization: Early morning shifts (4 AM - 7 AM) should prioritize
airport-bound trips, while late-night shifts (8 PM - 11 PM)
should focus on airport pickups.

2. Dynamic Pricing Adjustments

Increase per-mile rates for short-distance trips (<2 miles) since they have a
higher fare per mile.
Encourage pooling in residential areas to increase passenger counts per ride and
make trips more cost-effective.
```

```
Reduce wait-time charges at airports to attract more rideshare customers over
competitors like Uber/Lyft.

3. Improve Tipping Behavior Through Service Strategy

Higher tips (>25%) are seen for long-distance trips → Encourage longer trips
through discounts.
Trips with 3+ passengers have higher tips → Promote ride-sharing and group
discounts.
Nighttime trips (12 AM - 5 AM) have lower tipping rates → Improve driver
incentives to encourage nighttime shifts.

4. Reduce Operational Inefficiencies

Minimize empty miles: Use AI-based dispatching to reduce time between drop-off
and the next pickup.
Monitor surcharges: Some surcharges (airport fee, tolls) impact customer pricing
negatively → Consider optimizing fare transparency
for better customer trust.
Encourage digital payments: Trips with card payments have 20-30% higher tip
percentages compared to cash-based transactions.
```

4.1.2 [5 marks]

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

[100]:
```
print('''
Time of Day                | Key Zones to Target                         |␣
↪Strategy
6 AM - 9 AM (Morning Rush) | Business Districts, Transit Hubs, Airports | Focus␣
↪on commuters heading to work, airport travelers
9 AM - 4 PM (Daytime)      | Tourist Spots, Shopping Areas, Hospitals    |␣
↪Target midday travelers, local rides, and shopping trips
5 PM - 8 PM (Evening Rush)         | Transit Hubs, Business Districts,␣
↪Residential Areas | Capture office workers heading home & airport transfers
10 PM - 4 AM (Late Night)        | Nightlife Areas, Bars, Airport␣
↪Hotels               | Serve partygoers, late-night commuters, and␣
↪international travelers
All Day                             |Airports (JFK, LGA, EWR)                   ␣
↪ |        Ensure taxis are available for flights at peak departure times
''')
```

```
Time of Day                | Key Zones to Target                         |
Strategy
6 AM - 9 AM (Morning Rush) | Business Districts, Transit Hubs, Airports | Focus
on commuters heading to work, airport travelers
```

```
9 AM - 4 PM (Daytime)      | Tourist Spots, Shopping Areas, Hospitals   | Target
midday travelers, local rides, and shopping trips
5 PM - 8 PM (Evening Rush)      | Transit Hubs, Business Districts, Residential
Areas | Capture office workers heading home & airport transfers
10 PM - 4 AM (Late Night)       | Nightlife Areas, Bars, Airport Hotels       |
Serve partygoers, late-night commuters, and international travelers
All Day                        |Airports (JFK, LGA, EWR)                    |
Ensure taxis are available for flights at peak departure times
```

4.1.3 [5 marks] Propose data-driven adjustments to the pricing strategy to maximize revenue while maintaining competitive rates with other vendors.

[101]:
```
print('''
• Dynamic Pricing Based on Demand Fluctuations

Peak Hour Pricing (Morning & Evening Rush)

Peak Demand Zones:

• Morning (6 AM - 9 AM): Residential to business hubs (Upper East Side →␣
  ↪Midtown, Queens → Manhattan).

• Evening (5 PM - 8 PM): Business districts to residential areas, transit hubs.

Strategy:
Increase per-mile fare by +10% during rush hours to maximize revenue.
Apply a small fixed surcharge (~$2) for trips originating from high-demand␣
  ↪areas (e.g., Penn Station, Grand Central, Financial
District).
Encourage pre-booking with discounted off-peak fares to shift demand.

Late-Night Pricing Adjustments (10 PM - 4 AM)

◦ Key Demand Areas: Nightlife zones (Lower East Side, Williamsburg, Times␣
  ↪Square).

Strategy:
Increase per-mile fare by +15% in nightlife-heavy zones after 10 PM.
Introduce a "Safe Ride" discount for pooled rides after 2 AM to encourage␣
  ↪ride-sharing.
Reduce wait-time charges to encourage taxi use over Uber/Lyft during surge␣
  ↪pricing.


Airport & Long-Distance Trip Pricing
```

```
° Airports (JFK, LaGuardia, Newark) & Suburbs

Strategy:
Introduce dynamic airport flat fares based on real-time demand.
Offer discounted fares for return trips from airports to reduce empty miles.
For long-distance trips (>10 miles), implement tiered per-mile pricing:

• 0-5 miles: Standard rate

• 5-10 miles: +5% increase

• 10+ miles: -10% discount to encourage longer trips


2 Adjusting Pricing Based on Ride Type

Short-Distance Trips (<2 Miles)

° High Demand Areas: Midtown, Financial District, SoHo

Strategy:
Introduce a "Micro-Trip Fare" with a minimum $8 charge to compensate for␣
↪short-trip losses.
Implement a higher per-mile rate for trips under 2 miles (+20% increase).
Encourage walk-up street hails in high-density areas to reduce dispatch costs.


Pricing Adjustments for High-Tipping Zones

° High-Tip Areas: Business travelers, airport rides, long-distance rides

Strategy:
Lower base fare in high-tipping zones to encourage longer trips.
Offer "Priority Taxi" pricing (+10% premium) for riders who pre-book via app.
Promote in-app tipping & digital payment incentives to boost driver earnings.
''')
```

• Dynamic Pricing Based on Demand Fluctuations

Peak Hour Pricing (Morning & Evening Rush)

Peak Demand Zones:

• Morning (6 AM - 9 AM): Residential to business hubs (Upper East Side → Midtown, Queens → Manhattan).

• Evening (5 PM - 8 PM): Business districts to residential areas, transit hubs.

Strategy:
Increase per-mile fare by +10% during rush hours to maximize revenue.
Apply a small fixed surcharge (~$2) for trips originating from high-demand areas (e.g., Penn Station, Grand Central, Financial District).
Encourage pre-booking with discounted off-peak fares to shift demand.

Late-Night Pricing Adjustments (10 PM - 4 AM)

° Key Demand Areas: Nightlife zones (Lower East Side, Williamsburg, Times Square).

Strategy:
Increase per-mile fare by +15% in nightlife-heavy zones after 10 PM.
Introduce a "Safe Ride" discount for pooled rides after 2 AM to encourage ride-sharing.
Reduce wait-time charges to encourage taxi use over Uber/Lyft during surge pricing.

Airport & Long-Distance Trip Pricing

° Airports (JFK, LaGuardia, Newark) & Suburbs

Strategy:
Introduce dynamic airport flat fares based on real-time demand.
Offer discounted fares for return trips from airports to reduce empty miles.
For long-distance trips (>10 miles), implement tiered per-mile pricing:

- 0-5 miles: Standard rate

- 5-10 miles: +5% increase

- 10+ miles: -10% discount to encourage longer trips

2 Adjusting Pricing Based on Ride Type

Short-Distance Trips (<2 Miles)

° High Demand Areas: Midtown, Financial District, SoHo

Strategy:
Introduce a "Micro-Trip Fare" with a minimum $8 charge to compensate for short-trip losses.
Implement a higher per-mile rate for trips under 2 miles (+20% increase).
Encourage walk-up street hails in high-density areas to reduce dispatch costs.

Pricing Adjustments for High-Tipping Zones

° High-Tip Areas: Business travelers, airport rides, long-distance rides

Strategy:
Lower base fare in high-tipping zones to encourage longer trips.
Offer "Priority Taxi" pricing (+10% premium) for riders who pre-book via app.
Promote in-app tipping & digital payment incentives to boost driver earnings.