Data Analysis Railway in UK

Part 1

Toolkit & Loading Data and Inspecting

Toolkit

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

Loading Data and Inspecting

```
df = pd.read_csv('/content/railway.csv')
```

```
df.head()
```

	Transaction ID	Date of Purchase	Time of Purchase	Purchase Type	Payment Method	Railcard	Ticket Class	Ticket Type	Price	Departure Station	Arrival Destination	Date of Journey	Departu Ti
0	da8a6ba8-b3dc-4677-b176	2023-12-08	12:41:11	Online	Contactless	Adult	Standard	Advance	43	London Paddington	Liverpool Lime Street	2024-01-01	11:00
1	b0cdd1b0-f214-4197-be53	2023-12-16	11:23:01	Station	Credit Card	Adult	Standard	Advance	23	London Kings Cross	York	2024-01-01	09:45
2	f3ba7a96-f713-40d9-9629	2023-12-19	19:51:27	Online	Credit Card	NaN	Standard	Advance	3	Liverpool Lime Street	Manchester Piccadilly	2024-01-02	18:15
3	b2471f11-4fe7-4c87-8ab4	2023-12-20	23:00:36	Station	Credit Card	NaN	Standard	Advance	13	London Paddington	Reading	2024-01-01	21:30
4	2be00b45-0762-485e-a7a3	2023-12-27	18:22:56	Online	Contactless	NaN	Standard	Advance	76	Liverpool Lime Street	London Euston	2024-01-01	16:45

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 31653 entries, 0 to 31652
Data columns (total 18 columns):
 #   Column               Non-Null Count  Dtype 
---  ------               --------------  ----- 
 0   Transaction ID       31653 non-null  object
 1   Date of Purchase     31653 non-null  object
 2   Time of Purchase     31653 non-null  object
 3   Purchase Type        31653 non-null  object
 4   Payment Method       31653 non-null  object
 5   Railcard             10735 non-null  object
 6   Ticket Class         31653 non-null  object
 7   Ticket Type          31653 non-null  object
 8   Price                31653 non-null  int64 
 9   Departure Station    31653 non-null  object
 10  Arrival Destination  31653 non-null  object
 11  Date of Journey      31653 non-null  object
 12  Departure Time       31653 non-null  object
 13  Arrival Time         31653 non-null  object
 14  Actual Arrival Time  29773 non-null  object
 15  Journey Status       31653 non-null  object
 16  Reason for Delay     4172 non-null   object
 17  Refund Request       31653 non-null  object
dtypes: int64(1), object(17)
memory usage: 4.3+ MB
```

```
df.describe().round(4)
```

	Price
count	31653.0000
mean	23.4392
std	29.9976
min	1.0000
25%	5.0000
50%	11.0000
75%	35.0000
max	267.0000

```
pip install ydata-profiling
```

```
Requirement already satisfied: ydata-profiling in /usr/local/lib/python3.11/dist-packages (4.16.1)
Requirement already satisfied: scipy<1.16,>=1.4.1 in /usr/local/lib/python3.11/dist-packages (from ydata-profiling) (1.15.3)
Requirement already satisfied: pandas!=1.4.0,<3.0,>1.1 in /usr/local/lib/python3.11/dist-packages (from ydata-profiling) (2.2.2)
Requirement already satisfied: matplotlib<=3.10,>=3.5 in /usr/local/lib/python3.11/dist-packages (from ydata-profiling) (3.10.0)
Requirement already satisfied: pydantic>=2 in /usr/local/lib/python3.11/dist-packages (from ydata-profiling) (2.11.7)
Requirement already satisfied: PyYAML<6.1,>=5.0.0 in /usr/local/lib/python3.11/dist-packages (from ydata-profiling) (6.0.2)
Requirement already satisfied: jinja2<3.2,>=2.11.1 in /usr/local/lib/python3.11/dist-packages (from ydata-profiling) (3.1.6)
Requirement already satisfied: visions<0.8.2,>=0.7.5 in /usr/local/lib/python3.11/dist-packages (from visions[type_image_path]<0.8.2
Requirement already satisfied: numpy<2.2,>=1.16.0 in /usr/local/lib/python3.11/dist-packages (from ydata-profiling) (2.0.2)
Requirement already satisfied: htmlmin==0.1.12 in /usr/local/lib/python3.11/dist-packages (from ydata-profiling) (0.1.12)
Requirement already satisfied: phik<0.13,>=0.11.1 in /usr/local/lib/python3.11/dist-packages (from ydata-profiling) (0.12.5)
Requirement already satisfied: requests<3,>=2.24.0 in /usr/local/lib/python3.11/dist-packages (from ydata-profiling) (2.32.3)
Requirement already satisfied: tqdm<5,>=4.48.2 in /usr/local/lib/python3.11/dist-packages (from ydata-profiling) (4.67.1)
Requirement already satisfied: seaborn<0.14,>=0.10.1 in /usr/local/lib/python3.11/dist-packages (from ydata-profiling) (0.13.2)
Requirement already satisfied: multimethod<2,>=1.4 in /usr/local/lib/python3.11/dist-packages (from ydata-profiling) (1.12)
Requirement already satisfied: statsmodels<1,>=0.13.2 in /usr/local/lib/python3.11/dist-packages (from ydata-profiling) (0.14.5)
Requirement already satisfied: typeguard<5,>=3 in /usr/local/lib/python3.11/dist-packages (from ydata-profiling) (4.4.4)
Requirement already satisfied: imagehash==4.3.1 in /usr/local/lib/python3.11/dist-packages (from ydata-profiling) (4.3.1)
Requirement already satisfied: wordcloud>=1.9.3 in /usr/local/lib/python3.11/dist-packages (from ydata-profiling) (1.9.4)
Requirement already satisfied: dacite>=1.8 in /usr/local/lib/python3.11/dist-packages (from ydata-profiling) (1.9.2)
Requirement already satisfied: numba<=0.61,>=0.56.0 in /usr/local/lib/python3.11/dist-packages (from ydata-profiling) (0.60.0)
Requirement already satisfied: PyWavelets in /usr/local/lib/python3.11/dist-packages (from imagehash==4.3.1->ydata-profiling) (1.9.0
Requirement already satisfied: pillow in /usr/local/lib/python3.11/dist-packages (from imagehash==4.3.1->ydata-profiling) (11.3.0)
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.11/dist-packages (from jinja2<3.2,>=2.11.1->ydata-profiling
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.11/dist-packages (from matplotlib<=3.10,>=3.5->ydata-profi
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.11/dist-packages (from matplotlib<=3.10,>=3.5->ydata-profiling
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.11/dist-packages (from matplotlib<=3.10,>=3.5->ydata-prof
Requirement already satisfied: kiwisolver>=1.3.1 in /usr/local/lib/python3.11/dist-packages (from matplotlib<=3.10,>=3.5->ydata-prof
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.11/dist-packages (from matplotlib<=3.10,>=3.5->ydata-profil
Requirement already satisfied: pyparsing>=2.3.1 in /usr/local/lib/python3.11/dist-packages (from matplotlib<=3.10,>=3.5->ydata-profi
Requirement already satisfied: python-dateutil>=2.7 in /usr/local/lib/python3.11/dist-packages (from matplotlib<=3.10,>=3.5->ydata-p
Requirement already satisfied: llvmlite<0.44,>=0.43.0dev0 in /usr/local/lib/python3.11/dist-packages (from numba<=0.61,>=0.56.0->yda
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.11/dist-packages (from pandas!=1.4.0,<3.0,>1.1->ydata-profilin
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.11/dist-packages (from pandas!=1.4.0,<3.0,>1.1->ydata-profil
Requirement already satisfied: joblib>=0.14.1 in /usr/local/lib/python3.11/dist-packages (from phik<0.13,>=0.11.1->ydata-profiling)
Requirement already satisfied: annotated-types>=0.6.0 in /usr/local/lib/python3.11/dist-packages (from pydantic>=2->ydata-profiling
Requirement already satisfied: pydantic-core==2.33.2 in /usr/local/lib/python3.11/dist-packages (from pydantic>=2->ydata-profiling)
Requirement already satisfied: typing-extensions>=4.12.2 in /usr/local/lib/python3.11/dist-packages (from pydantic>=2->ydata-profili
Requirement already satisfied: typing-inspection>=0.4.0 in /usr/local/lib/python3.11/dist-packages (from pydantic>=2->ydata-profilin
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.11/dist-packages (from requests<3,>=2.24.0->ydata
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.11/dist-packages (from requests<3,>=2.24.0->ydata-profiling)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.11/dist-packages (from requests<3,>=2.24.0->ydata-profil
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.11/dist-packages (from requests<3,>=2.24.0->ydata-profil
Requirement already satisfied: patsy>=0.5.6 in /usr/local/lib/python3.11/dist-packages (from statsmodels<1,>=0.13.2->ydata-profiling
Requirement already satisfied: attrs>=19.3.0 in /usr/local/lib/python3.11/dist-packages (from visions<0.8.2,>=0.7.5->visions[type_in
Requirement already satisfied: networkx>=2.4 in /usr/local/lib/python3.11/dist-packages (from visions<0.8.2,>=0.7.5->visions[type_in
Requirement already satisfied: puremagic in /usr/local/lib/python3.11/dist-packages (from visions<0.8.2,>=0.7.5->visions[type_image_
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.11/dist-packages (from python-dateutil>=2.7->matplotlib<=3.10,>=3
```

```
from ydata_profiling import ProfileReport
ProfileReport(df)
```

Summarize dataset: 100% 29/29 [00:16<00:00, 2.67it/s, Completed]

```
  0%|          | 0/18 [00:00<?, ?it/s]
  6%|▌         | 1/18 [00:01<00:18,  1.08s/it]
100%|██████████| 18/18 [00:12<00:00,  1.43it/s]
```

Generate report structure: 100% 1/1 [00:13<00:00, 13.59s/it]

Render HTML: 100% 1/1 [00:01<00:00, 1.73s/it]

YData Profiling Report

Overview Variables Interactions Correlations Missing values Sample

Overview

Brought to you by YData

Overview Alerts 10 Reproduction

Dataset statistics

Number of variables	18
Number of observations	31653
Missing cells	50279
Missing cells (%)	8.8%
Duplicate rows	0
Duplicate rows (%)	0.0%
Total size in memory	4.3 MiB
Average record size in memory	144.0 B

Variable types

Text	1
DateTime	6
Categorical	9
Numeric	1
Boolean	1

Variables

Select Columns

Part 2

Data Cleaning and Preparation

Handle missing values

```
df.isnull().sum()
```

	0
Transaction ID	0
Date of Purchase	0
Time of Purchase	0
Purchase Type	0
Payment Method	0
Railcard	20918
Ticket Class	0
Ticket Type	0
Price	0
Departure Station	0
Arrival Destination	0
Date of Journey	0
Departure Time	0
Arrival Time	0
Actual Arrival Time	1880
Journey Status	0
Reason for Delay	27481
Refund Request	0

dtype: int64

```
total_rows = len(df)

missing_railcard = df['Railcard'].isnull().sum()
missing_actual_arrival_time = df['Actual Arrival Time'].isnull().sum()
missing_reason_for_delay = df['Reason for Delay'].isnull().sum()

print(f"Missing values in 'Railcard': {missing_railcard} ({missing_railcard / total_rows:.2%})")
print(f"Missing values in 'Actual Arrival Time': {missing_actual_arrival_time} ({missing_actual_arrival_time / total_rows:.2%})")
print(f"Missing values in 'Reason for Delay': {missing_reason_for_delay} ({missing_reason_for_delay / total_rows:.2%})")
```

```
Missing values in 'Railcard': 20918 (66.09%)
Missing values in 'Actual Arrival Time': 1880 (5.94%)
Missing values in 'Reason for Delay': 27481 (86.82%)
```

```
df['Railcard'] = df['Railcard'].fillna('Unknown')
df['Reason for Delay'] = df['Reason for Delay'].fillna('Unknown Reason')
on_time_mask = df['Journey Status'] == 'On Time'
df.loc[on_time_mask, 'Actual Arrival Time'] = df.loc[on_time_mask, 'Actual Arrival Time'].fillna(df.loc[on_time_mask, 'Arrival Time'])

# Remove trailing space from 'Adult ' in 'Railcard' column
df['Railcard'] = df['Railcard'].replace('Adult ', 'Adult')

print("Percentage of missing values after handling:")
print(df.isnull().mean() * 100)
```

```
Percentage of missing values after handling:
Transaction ID         0.000000
Date of Purchase       0.000000
Time of Purchase       0.000000
Purchase Type          0.000000
Payment Method         0.000000
Railcard               0.000000
Ticket Class           0.000000
Ticket Type            0.000000
Price                  0.000000
Departure Station      0.000000
Arrival Destination    0.000000
Date of Journey        0.000000
Departure Time         0.000000
Arrival Time           0.000000
Actual Arrival Time    5.939405
Journey Status         0.000000
Reason for Delay       0.000000
Refund Request         0.000000
dtype: float64
```

```
df['Actual Arrival Time'].fillna(df['Arrival Time'])
print(f"Missing values in 'Actual Arrival Time' after handling: {df['Actual Arrival Time'].isnull().sum()}")
```

```
Missing values in 'Actual Arrival Time' after handling: 1880
```

```
print(df.isnull().mean() * 100)
```

```
Transaction ID         0.0
Date of Purchase       0.0
Time of Purchase       0.0
Purchase Type          0.0
Payment Method         0.0
Railcard               0.0
Ticket Class           0.0
Ticket Type            0.0
Price                  0.0
Departure Station      0.0
Arrival Destination    0.0
Date of Journey        0.0
Departure Time         0.0
Arrival Time           0.0
Actual Arrival Time    0.0
Journey Status         0.0
Reason for Delay       0.0
Refund Request         0.0
dtype: float64
```

Handle duplicates

```
df.duplicated().sum()
```

```
np.int64(0)
```

Data Transformation

```
df['Date of Purchase'] = pd.to_datetime(df['Date of Purchase'])
df['Date of Journey'] = pd.to_datetime(df['Date of Journey'])

df['Time of Purchase'] = pd.to_datetime(df['Time of Purchase'], format='%H:%M:%S')
df['Departure Time'] = pd.to_datetime(df['Departure Time'], format='%H:%M:%S')
df['Arrival Time'] = pd.to_datetime(df['Arrival Time'], format='%H:%M:%S')
df['Actual Arrival Time'] = pd.to_datetime(df['Actual Arrival Time'], format='%H:%M:%S', errors='coerce')
```

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 31653 entries, 0 to 31652
Data columns (total 18 columns):
 #   Column               Non-Null Count  Dtype
---  ------               --------------  -----
 0   Transaction ID       31653 non-null  object
 1   Date of Purchase     31653 non-null  datetime64[ns]
 2   Time of Purchase     31653 non-null  datetime64[ns]
 3   Purchase Type        31653 non-null  object
 4   Payment Method       31653 non-null  object
 5   Railcard             31653 non-null  object
 6   Ticket Class         31653 non-null  object
 7   Ticket Type          31653 non-null  object
 8   Price                31653 non-null  int64
 9   Departure Station    31653 non-null  object
 10  Arrival Destination  31653 non-null  object
 11  Date of Journey      31653 non-null  datetime64[ns]
 12  Departure Time       31653 non-null  datetime64[ns]
 13  Arrival Time         31653 non-null  datetime64[ns]
 14  Actual Arrival Time  29773 non-null  datetime64[ns]
 15  Journey Status       31653 non-null  object
 16  Reason for Delay     31653 non-null  object
 17  Refund Request       31653 non-null  object
dtypes: datetime64[ns](6), int64(1), object(11)
memory usage: 4.3+ MB
```

Calculate journey duration and delay and convert categorical columns to category data type.

```
df['Departure_datetime'] = pd.to_datetime(df['Date of Journey'].astype(str) + ' ' + df['Departure Time'].astype(str))
df['Scheduled_Arrival_datetime'] = pd.to_datetime(df['Date of Journey'].astype(str) + ' ' + df['Arrival Time'].astype(str))
```

```
# Handle potential NaT values in 'Actual Arrival Time' before combining with 'Date of Journey'
df['Actual_Arrival_datetime'] = pd.to_datetime(df['Date of Journey'].astype(str) + ' ' + df['Actual Arrival Time'].astype(str), errors=

df['Journey Duration'] = (df['Actual_Arrival_datetime'] - df['Departure_datetime']).dt.total_seconds() / 60
df.loc[df['Journey Duration'] < 0, 'Journey Duration'] += 24 * 60  # Add 24 hours if negative

df['Delay'] = (df['Scheduled_Arrival_datetime'] - df['Actual_Arrival_datetime']).dt.total_seconds() / 60

categorical_cols = ['Purchase Type', 'Payment Method', 'Railcard', 'Ticket Class', 'Ticket Type', 'Departure Station', 'Arrival Destinat
for col in categorical_cols:
    df[col] = df[col].astype('category')

df = df.drop(columns=['Departure_datetime', 'Actual_Arrival_datetime', 'Scheduled_Arrival_datetime'])
```

```
/tmp/ipython-input-266334473.py:1: UserWarning: Could not infer format, so each element will be parsed individually, falling back to
  df['Departure_datetime'] = pd.to_datetime(df['Date of Journey'].astype(str) + ' ' + df['Departure Time'].astype(str))
/tmp/ipython-input-266334473.py:2: UserWarning: Could not infer format, so each element will be parsed individually, falling back to
  df['Scheduled_Arrival_datetime'] = pd.to_datetime(df['Date of Journey'].astype(str) + ' ' + df['Arrival Time'].astype(str))
/tmp/ipython-input-266334473.py:5: UserWarning: Could not infer format, so each element will be parsed individually, falling back to
  df['Actual_Arrival_datetime'] = pd.to_datetime(df['Date of Journey'].astype(str) + ' ' + df['Actual Arrival Time'].astype(str), er
```

Handle potential inconsistencies in categorical data

```
categorical_cols = ['Purchase Type', 'Payment Method', 'Railcard', 'Ticket Class', 'Ticket Type', 'Departure Station', 'Arrival Destinat
for col in categorical_cols:
    encoded_cols = [c for c in df.columns if c.startswith(col + '_')]
    print(f"Unique values for one-hot encoded '{col}' columns:")
    for encoded_col in encoded_cols:
        print(f"- {encoded_col}: {df[encoded_col].unique()}")
```

```
Unique values for one-hot encoded 'Purchase Type' columns:
Unique values for one-hot encoded 'Payment Method' columns:
Unique values for one-hot encoded 'Railcard' columns:
Unique values for one-hot encoded 'Ticket Class' columns:
Unique values for one-hot encoded 'Ticket Type' columns:
Unique values for one-hot encoded 'Departure Station' columns:
Unique values for one-hot encoded 'Arrival Destination' columns:
Unique values for one-hot encoded 'Journey Status' columns:
Unique values for one-hot encoded 'Reason for Delay' columns:
Unique values for one-hot encoded 'Refund Request' columns:
```

Based on the previous output, there are inconsistencies in the 'Reason for Delay' column with both 'Signal Failure' and 'Signal failure', and 'Staff Shortage' and 'Staffing', and 'Weather' and 'Weather Conditions'. These need to be unified by renaming the one-hot encoded columns and then combining them.

```
# Perform one-hot encoding on the 'Reason for Delay' column
df = pd.get_dummies(df, columns=['Reason for Delay'], prefix='Reason for Delay')

# Now combine the columns with similar meanings
df['Reason for Delay_Signal Failure'] = df['Reason for Delay_Signal Failure'] | df['Reason for Delay_Signal failure']
df.drop(columns=['Reason for Delay_Signal failure'], inplace=True)
df['Reason for Delay_Staff Shortage'] = df['Reason for Delay_Staff Shortage'] | df['Reason for Delay_Staffing']
df.drop(columns=['Reason for Delay_Staffing'], inplace=True)
df['Reason for Delay_Weather'] = df['Reason for Delay_Weather'] | df['Reason for Delay_Weather Conditions']
df.drop(columns=['Reason for Delay_Weather Conditions'], inplace=True)

# Display the updated columns
display(df[['Reason for Delay_Signal Failure', 'Reason for Delay_Staff Shortage', 'Reason for Delay_Weather']].head())
```

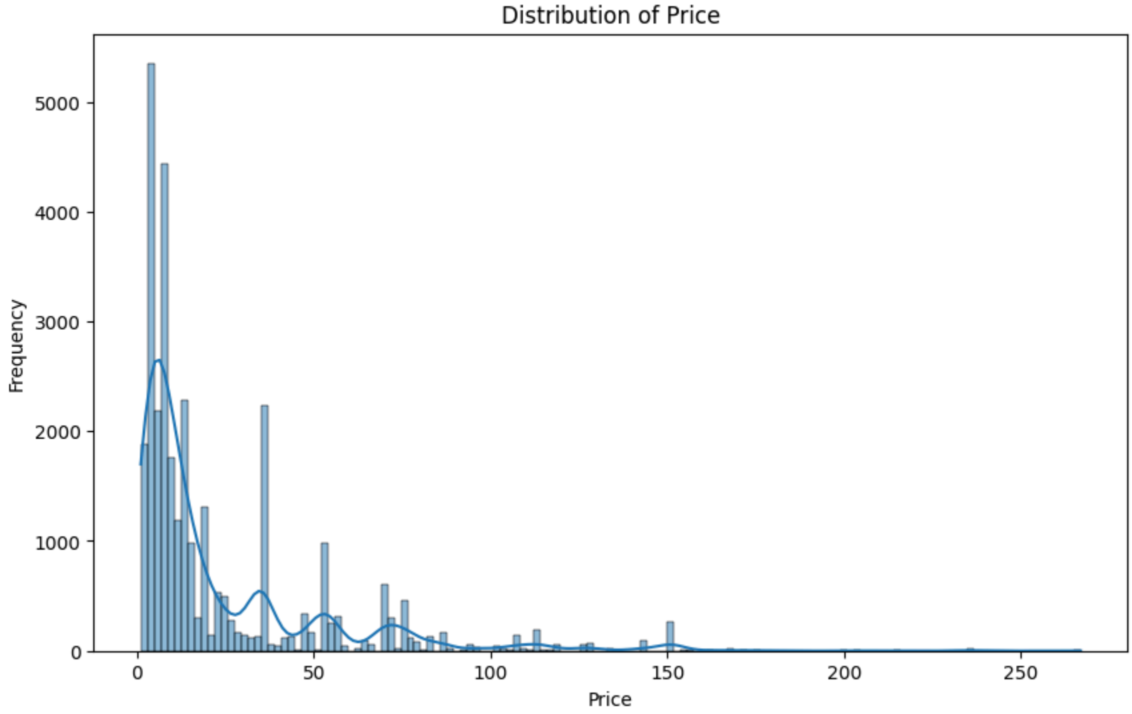
	Reason for Delay_Signal Failure	Reason for Delay_Staff Shortage	Reason for Delay_Weather
0	False	False	False
1	True	False	False
2	False	False	False
3	False	False	False
4	False	False	False

Part 3

Basic Analysis (Distribution , Outliers , Winsorizing)

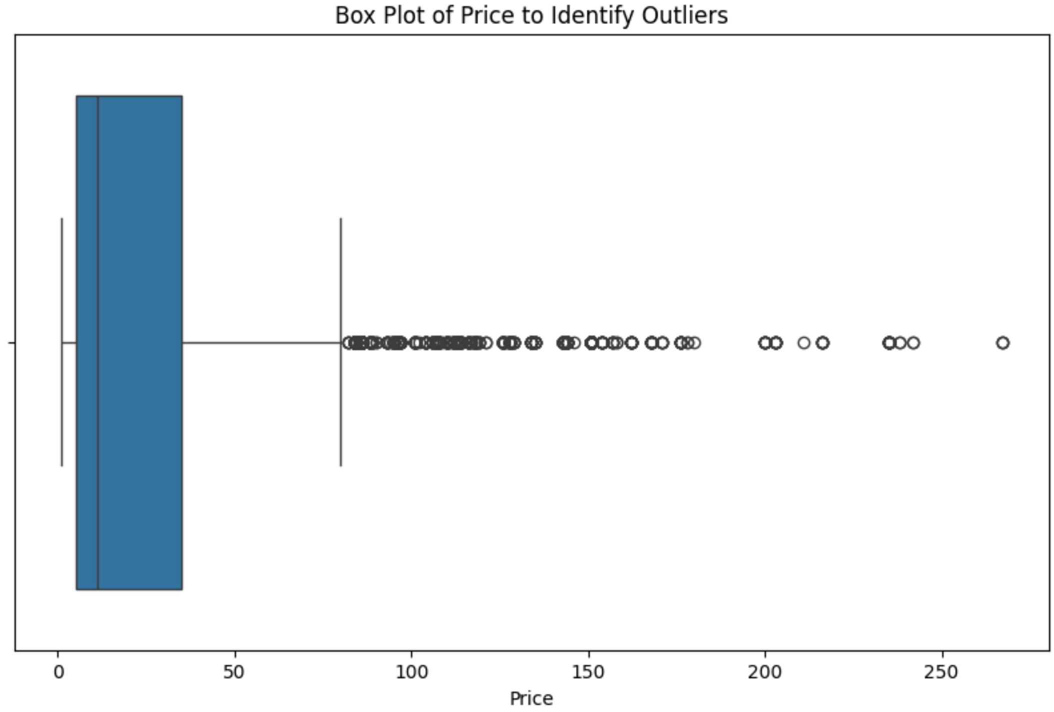
Price Distribution

```
plt.figure(figsize=(10, 6))
sns.histplot(df['Price'], kde=True)
plt.title('Distribution of Price')
plt.xlabel('Price')
plt.ylabel('Frequency')
plt.show()
```



Price Distribution (Outliers)

```
plt.figure(figsize=(10, 6))
sns.boxplot(x=df['Price'])
plt.title('Box Plot of Price to Identify Outliers')
plt.xlabel('Price')
plt.show()
```



Explore Outliers in Price

```
# Sort the DataFrame by 'Price' in descending order and display the top N rows
n_top_outliers = 10  # You can adjust this number
df_sorted_by_price = df.sort_values(by='Price', ascending=False)
display(df_sorted_by_price.head(n_top_outliers))
```

	Transaction ID	Date of Purchase	Time of Purchase	Purchase Type	Payment Method	Railcard	Ticket Class	Ticket Type	Price	Departure Station	...	Actual Arrival Time	Journey Status	R Re
11849	de723682-3979-4d69-9664	2024-02-09	1900-01-01 06:30:59	Station	Credit Card	Unknown	First Class	Anytime	267	Manchester Piccadilly	...	1900-01-01 10:16:00	Delayed	
711	092e5598-08de-42f5-b6b2	2024-01-04	1900-01-01 06:35:01	Station	Credit Card	Unknown	First Class	Anytime	267	Manchester Piccadilly	...	1900-01-01 10:23:00	Delayed	
2042	05193f47-2107-4bb8-8adb	2024-01-09	1900-01-01 06:33:37	Station	Credit Card	Unknown	First Class	Anytime	267	Manchester Piccadilly	...	1900-01-01 10:29:00	Delayed	
31434	a4bbac34-6ed7-4d71-b738	2024-04-29	1900-01-01 16:49:20	Station	Contactless	Unknown	First Class	Anytime	242	Reading	...	1900-01-01 20:45:00	On Time	
13367	d327e0ec-e1ac-436a-aa5c	2024-02-13	1900-01-01 16:51:59	Station	Contactless	Unknown	First Class	Anytime	242	Reading	...	1900-01-01 20:45:00	On Time	
22488	419bcd59-fbc0-48f5-b48c	2024-03-26	1900-01-01 07:29:28	Station	Contactless	Unknown	First Class	Anytime	238	Liverpool Lime Street	...	1900-01-01 11:15:00	On Time	
20279	8e02ccef-5f58-48ca-b8b8	2024-03-18	1900-01-01 07:25:54	Station	Contactless	Unknown	First Class	Anytime	238	Liverpool Lime Street	...	1900-01-01 11:15:00	On Time	
24788	520bc09b-f83f-404a-9bf5	2024-04-05	1900-01-01 06:32:18	Station	Credit Card	Unknown	First Class	Anytime	235	Liverpool Lime Street	...	1900-01-01 11:11:00	Delayed	
21094	82b3b685-a3d7-49d1-8ffb	2024-03-21	1900-01-01 06:38:11	Station	Credit Card	Unknown	First Class	Anytime	235	Liverpool Lime Street	...	1900-01-01 10:48:00	Delayed	
29422	a94dd34b-3925-4eab-8b27	2024-04-22	1900-01-01 06:37:31	Station	Credit Card	Unknown	First Class	Anytime	235	Liverpool Lime Street	...	NaT	Cancelled	

10 rows × 24 columns

Define a threshold for high prices:

Calculate Q1, Q3, and IQR for the 'Price' column, define the upper bound for outliers, and use it as the threshold for high prices.

```
Q1 = df['Price'].quantile(0.25)
Q3 = df['Price'].quantile(0.75)
IQR = Q3 - Q1
upper_bound = Q3 + 1.5 * IQR
print(f"Q1: {Q1}")
print(f"Q3: {Q3}")
print(f"IQR: {IQR}")
print(f"Upper bound (High Price Threshold): {upper_bound}")
```

```
Q1: 5.0
Q3: 35.0
IQR: 30.0
Upper bound (High Price Threshold): 80.0
```

Create a subset of high-priced tickets

Filter the DataFrame to create a new DataFrame containing only the rows where the 'Price' is above the defined threshold.

```
df_high_price = df[df['Price'] > upper_bound].copy()
display(df_high_price.head())
```

```
print(f"Number of high-priced tickets: {len(df_high_price)}")
```

	Transaction ID	Date of Purchase	Time of Purchase	Purchase Type	Payment Method	Railcard	Ticket Class	Ticket Type	Price	Departure Station	...	Actual Arrival Time	Journey Status	Refu Requ
25	842da93c-b820-42dc-ad4f	2023-12-31	1900-01-01 15:19:53	Online	Contactless	Unknown	Standard	Advance	86	Manchester Piccadilly	...	1900-01-01 16:00:00	On Time	
45	767314a0-f839-4607-a3d3	2024-01-01	1900-01-01 05:09:30	Station	Credit Card	Unknown	First Class	Advance	134	Manchester Piccadilly	...	1900-01-01 05:31:00	Delayed	
51	382d60f9-9fe0-4920-97e4	2024-01-01	1900-01-01 06:34:08	Station	Credit Card	Unknown	Standard	Anytime	151	Liverpool Lime Street	...	1900-01-01 10:39:00	Delayed	
61	711c08ba-eb61-44ba-821a	2024-01-01	1900-01-01 09:30:09	Station	Credit Card	Unknown	First Class	Advance	134	Manchester Piccadilly	...	1900-01-01 10:08:00	Delayed	
68	9082a416-480e-4ca4-bf9d	2024-01-01	1900-01-01 15:39:11	Station	Credit Card	Unknown	Standard	Anytime	151	Liverpool Lime Street	...	1900-01-01 19:15:00	On Time	

5 rows × 24 columns

```
Number of high-priced tickets: 1555
```

Compare characteristics

Create a subset for non-high-priced tickets, calculate and print descriptive statistics for numerical columns, and calculate and print value counts for categorical columns for both subsets.

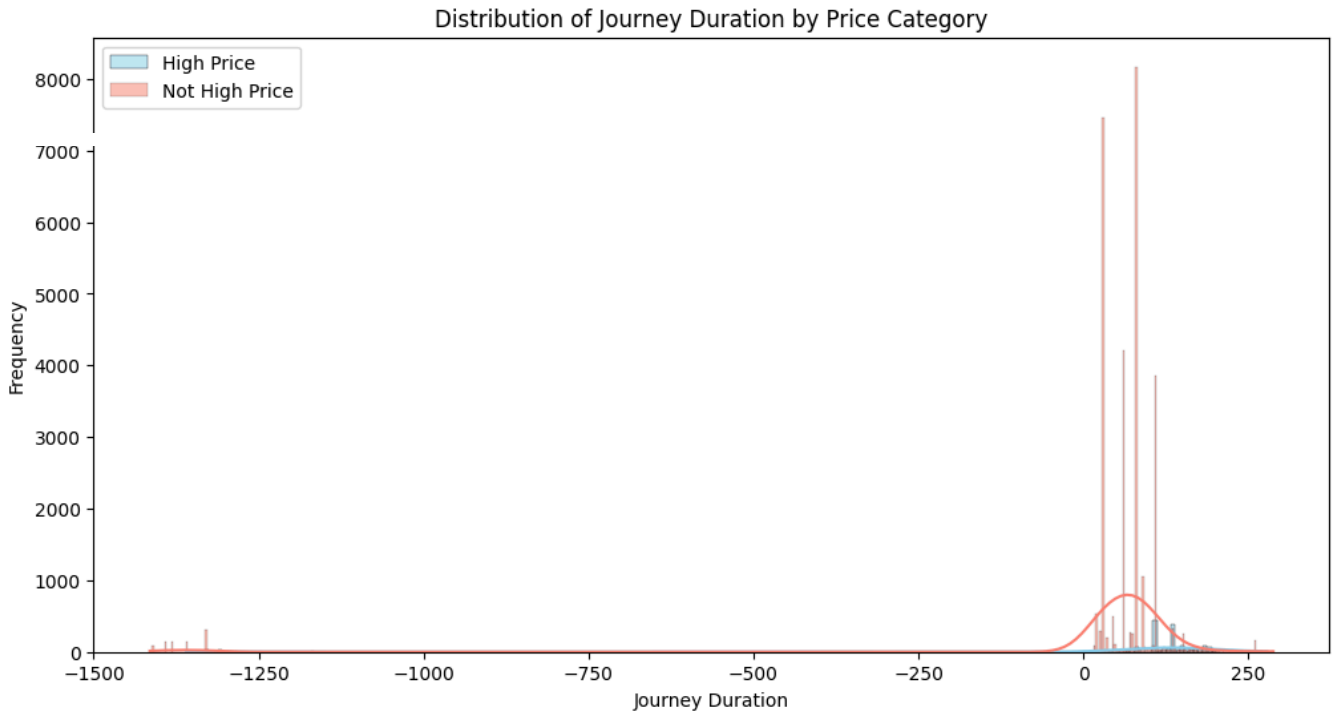
```
df_not_high_price = df[df['Price'] <= upper_bound].copy()
numerical_cols_compare = ['Journey Duration', 'Delay']
print("Descriptive statistics for High-Priced Tickets:")
display(df_high_price[numerical_cols_compare].describe())
print("\nDescriptive statistics for Not High-Priced Tickets:")
display(df_not_high_price[numerical_cols_compare].describe())
categorical_cols_compare = ['Purchase Type', 'Payment Method', 'Railcard', 'Ticket Class', 'Ticket Type', 'Departure Station', 'Arrival
print("\nValue counts for Categorical Columns (High-Priced Tickets):")
for col in categorical_cols_compare:
    print(f"\n{col}:")
    display(df_high_price[col].value_counts(normalize=True))
print("\nValue counts for Categorical Columns (Not High-Priced Tickets):")
for col in categorical_cols_compare:
    print(f"\n{col}:")
    display(df_not_high_price[col].value_counts(normalize=True))
```

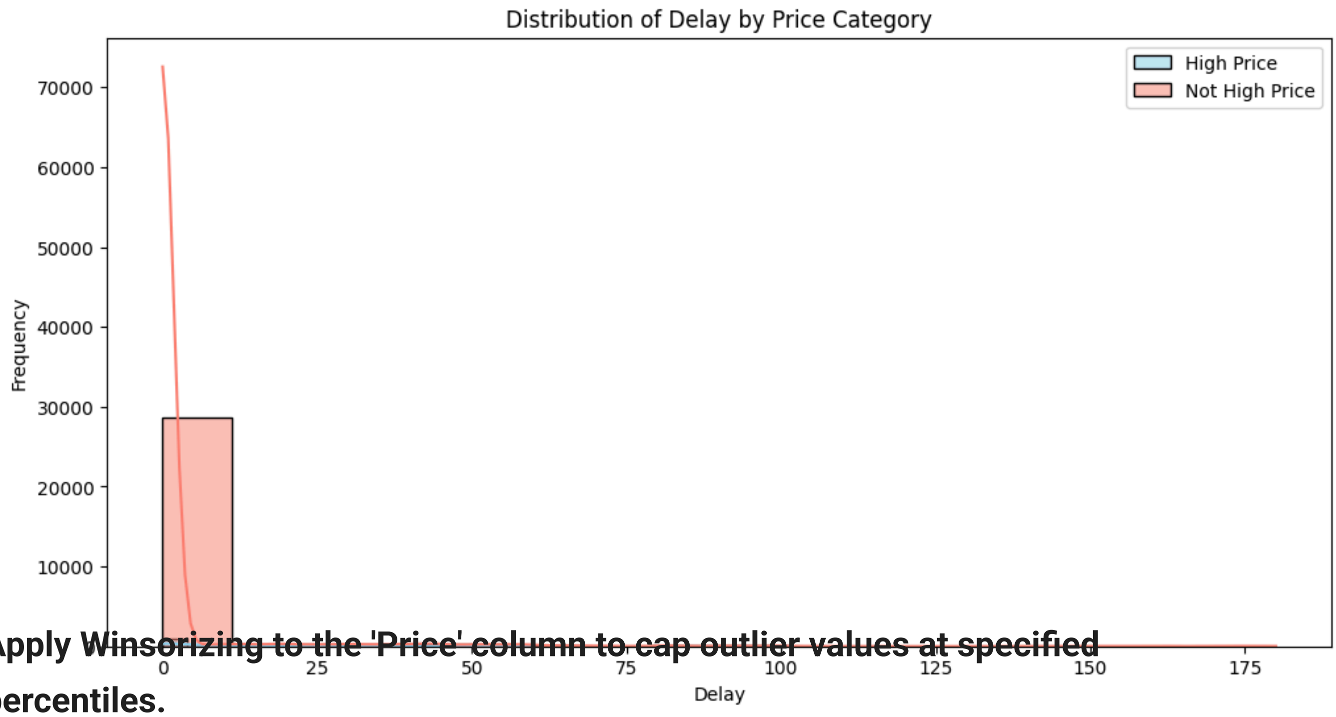
```
Descriptive statistics for High-Priced Tickets:
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
/tmp/ipython-input-332978330.py in <cell line: 0>()
      2 numerical_cols_compare = ['Journey Duration', 'Delay']
      3 print("Descriptive statistics for High-Priced Tickets:")
----> 4 display(df_high_price[numerical_cols_compare].describe())
      5 print("\nDescriptive statistics for Not High-Priced Tickets:")
      6 display(df_not_high_price[numerical_cols_compare].describe())

2 frames
/usr/local/lib/python3.11/dist-packages/pandas/core/indexes/base.py in _raise_if_missing(self, key, indexer, axis_name)
   6247         if nmissing:
   6248             if nmissing == len(indexer):
-> 6249                 raise KeyError(f"None of [{key}] are in the [{axis_name}]")
   6250 
   6251             not_found = list(ensure_index(key)[missing_mask.nonzero()[0]].unique())

KeyError: "None of [Index(['Journey Duration', 'Delay'], dtype='object')] are in the [columns]"
```

Visualize the distribution of key numerical and categorical columns for both high-priced and not high-priced tickets to compare their characteristics.

```
numerical_cols_to_plot = ['Journey Duration', 'Delay']
for col in numerical_cols_to_plot:
    plt.figure(figsize=(12, 6))
    sns.histplot(df_high_price[col], kde=True, color='skyblue', label='High Price')
    sns.histplot(df_not_high_price[col], kde=True, color='salmon', label='Not High Price')
    plt.title(f'Distribution of {col} by Price Category')
```

```
    plt.xlabel(col)
    plt.ylabel('Frequency')
    plt.legend()
    plt.show()

# Perform one-hot encoding on the categorical columns to be plotted
categorical_cols_to_encode = ['Purchase Type', 'Ticket Class', 'Ticket Type', 'Journey Status', 'Refund Request']
df_high_price_encoded = pd.get_dummies(df_high_price, columns=categorical_cols_to_encode, drop_first=False)
df_not_high_price_encoded = pd.get_dummies(df_not_high_price, columns=categorical_cols_to_encode, drop_first=False)

categorical_cols_to_plot = ['Purchase Type_Station', 'Ticket Class_Standard', 'Ticket Type_Anytime', 'Ticket Type_Off-Peak', 'Journey St
for col in categorical_cols_to_plot:
    plt.figure(figsize=(8, 5))
    high_price_counts = df_high_price_encoded[col].value_counts(normalize=True).reset_index()
    high_price_counts['Price Category'] = 'High Price'
    not_high_price_counts = df_not_high_price_encoded[col].value_counts(normalize=True).reset_index()
    not_high_price_counts['Price Category'] = 'Not High Price'
    combined_counts = pd.concat([high_price_counts, not_high_price_counts])
    sns.barplot(x=col, y='proportion', hue='Price Category', data=combined_counts)
    plt.title(f'Proportion of {col} by Price Category')
    plt.xlabel(col)
    plt.ylabel('Proportion')
    plt.show()
```





Apply Winsorizing to the 'Price' column to cap outlier values at specified percentiles.

Proportion of Purchase Type_Station by Price Category

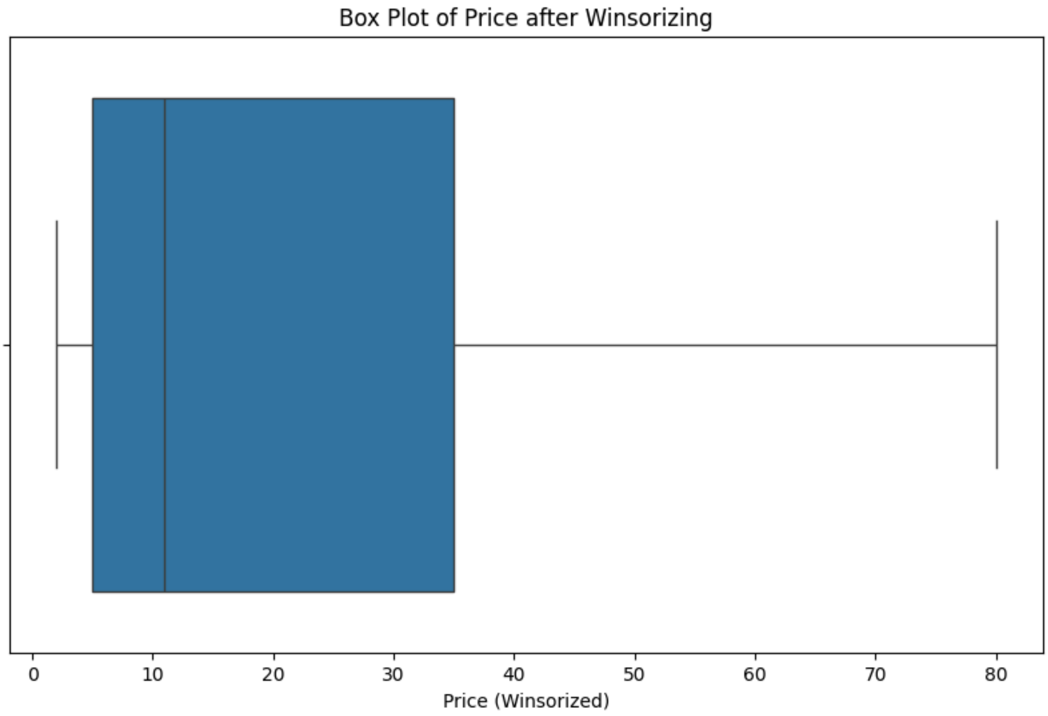
Calculate the lower and upper bounds for Winsorizing based on percentiles (e.g., 5th and 95th) and apply the capping to the 'Price' column. Display the descriptive statistics and a box plot after Winsorizing to verify the effect.

```
# Calculate lower and upper bounds based on percentiles
lower_bound = df['Price'].quantile(0.05)
upper_bound_winsorize = df['Price'].quantile(0.95)
# Apply Winsorizing
df['Price_winsorized'] = df['Price'].clip(lower=lower_bound, upper=upper_bound_winsorize)
# Display descriptive statistics of the new column
print("Descriptive statistics of 'Price' after Winsorizing:")
display(df['Price_winsorized'].describe())
# Display a box plot of the new column to visualize the effect
plt.figure(figsize=(10, 6))
sns.boxplot(x=df['Price_winsorized'])
plt.title('Box Plot of Price after Winsorizing')
plt.xlabel('Price (Winsorized)')
plt.show()
```

```
Descriptive statistics of 'Price' after Winsorizing:
```

	Price_winsorized
count	31653.000000
mean	21.431270
std	23.203243
min	2.000000
25%	5.000000
50%	11.000000
75%	35.000000
max	80.000000

dtype: float64



Part 4

KPIs

Key Performance Indicators: Revenue/Prices and Routes

```
# Display previously calculated KPIs for Revenue/Prices
print("Total Revenue:")
total_revenue = df['Price'].sum()
```

```
print(f"{total_revenue}")

print("\nAverage Ticket Price:")
average_price = df['Price'].mean()
print(f"{average_price:.2f}")
```

```
Total Revenue:
741921

Average Ticket Price:
23.44
```

Revenue by Stations

```
# Calculate total revenue per departure station
revenue_by_departure_station = df.groupby('Departure Station')['Price'].sum().sort_values(ascending=False)
print("Total Revenue by Departure Station:")
display(revenue_by_departure_station)

# Calculate total revenue per arrival station
revenue_by_arrival_station = df.groupby('Arrival Destination')['Price'].sum().sort_values(ascending=False)
print("\nTotal Revenue by Arrival Station:")
display(revenue_by_arrival_station)
```

Revenue by Ticket Type and Class

```
# Calculate total revenue by Ticket Type
revenue_by_ticket_type = df.groupby('Ticket Type')['Price'].sum().sort_values(ascending=False)
print("Total Revenue by Ticket Type:")
display(revenue_by_ticket_type)

# Calculate total revenue by Ticket Class
revenue_by_ticket_class = df.groupby('Ticket Class')['Price'].sum().sort_values(ascending=False)
print("\nTotal Revenue by Ticket Class:")
display(revenue_by_ticket_class)

# Calculate total revenue by Ticket Type and Ticket Class
revenue_by_type_and_class = df.groupby(['Ticket Type', 'Ticket Class'])['Price'].sum().unstack(fill_value=0)
print("\nTotal Revenue by Ticket Type and Ticket Class:")
display(revenue_by_type_and_class)
```

Revenue per Journey

```
# Calculate average revenue per journey (which is the average ticket price)
average_revenue_per_journey = df['Price'].mean()
print(f"Average Revenue per Journey: {average_revenue_per_journey:.2f}")
```

Key Performance Indicators: Time and Delay

```
# Calculate average delay
average_delay = df['Delay'].mean()
print(f"Average Delay (minutes): {average_delay:.2f}")

# Calculate proportion of delayed journeys
delayed_journeys_count = df[df['Journey Status'] == 'Delayed'].shape[0]
total_journeys_count = df.shape[0]
proportion_delayed = delayed_journeys_count / total_journeys_count
print(f"Proportion of Delayed Journeys: {proportion_delayed:.2%}")

# Calculate average delay for delayed journeys
average_delay_for_delayed = df[df['Journey Status'] == 'Delayed']['Delay'].mean()
print(f"Average Delay for Delayed Journeys Only (minutes): {average_delay_for_delayed:.2f}")
```

```
Average Delay (minutes): 3.06
Proportion of Delayed Journeys: 7.24%
Average Delay for Delayed Journeys Only (minutes): 42.21
```

Key Performance Indicators: Ticket Types and Purchase Methods

```
# Calculate proportion of each Ticket Type
print("\nProportion of each Ticket Type:")
display(df['Ticket Type'].value_counts(normalize=True))

# Calculate proportion of each Purchase Type
print("\nProportion of each Purchase Type:")
display(df['Purchase Type'].value_counts(normalize=True))

# Calculate proportion of each Payment Method
print("\nProportion of each Payment Method:")
display(df['Payment Method'].value_counts(normalize=True))
```

```
Proportion of each Ticket Type:
```

	proportion
Ticket Type	
Advance	0.554797
Off-Peak	0.276498
Anytime	0.168704

dtype: float64

```
Proportion of each Purchase Type:
```

	proportion
Purchase Type	
Online	0.585126
Station	0.414874

dtype: float64

```
Proportion of each Payment Method:
```

	proportion
Payment Method	
Credit Card	0.604556
Contactless	0.342274
Debit Card	0.053170

dtype: float64

Part 5

Deep analysis

Analyze the distribution of the price column

```
df_not_high_price = df[df['Price'] <= upper_bound].copy()

numerical_cols_compare = ['Journey Duration', 'Delay']
print("Descriptive statistics for High-Priced Tickets:")
display(df_high_price[numerical_cols_compare].describe())
print("\nDescriptive statistics for Not High-Priced Tickets:")
display(df_not_high_price[numerical_cols_compare].describe())

categorical_cols_compare = ['Purchase Type', 'Payment Method', 'Railcard', 'Ticket Class', 'Ticket Type', 'Departure Station', 'Arrival
print("\nValue counts for Categorical Columns (High-Priced Tickets):")
for col in categorical_cols_compare:
    print(f"\n{col}:")
    display(df_high_price[col].value_counts(normalize=True))

print("\nValue counts for Categorical Columns (Not High-Priced Tickets):")
for col in categorical_cols_compare:
    print(f"\n{col}:")
    display(df_not_high_price[col].value_counts(normalize=True))

numerical_cols_to_plot = ['Journey Duration', 'Delay']
for col in numerical_cols_to_plot:
    plt.figure(figsize=(12, 6))
    sns.histplot(df_high_price[col], kde=True, color='skyblue', label='High Price')
    sns.histplot(df_not_high_price[col], kde=True, color='salmon', label='Not High Price')
```

```
    plt.title(f'Distribution of {col} by Price Category')
    plt.xlabel(col)
    plt.ylabel('Frequency')
    plt.legend()
    plt.show()

categorical_cols_to_encode = ['Purchase Type', 'Ticket Class', 'Ticket Type', 'Journey Status', 'Refund Request']
df_high_price_encoded = pd.get_dummies(df_high_price, columns=categorical_cols_to_encode, drop_first=False)
df_not_high_price_encoded = pd.get_dummies(df_not_high_price, columns=categorical_cols_to_encode, drop_first=False)

categorical_cols_to_plot = [col for col in df_high_price_encoded.columns if any(cat in col for cat in categorical_cols_to_encode)]
for col in categorical_cols_to_plot:
    if df_high_price_encoded[col].nunique() > 1: # Only plot if there are at least two categories
        plt.figure(figsize=(8, 5))
        high_price_counts = df_high_price_encoded[col].value_counts(normalize=True).reset_index()
        high_price_counts['Price Category'] = 'High Price'
        not_high_price_counts = df_not_high_price_encoded[col].value_counts(normalize=True).reset_index()
        not_high_price_counts['Price Category'] = 'Not High Price'
        combined_counts = pd.concat([high_price_counts, not_high_price_counts])
        sns.barplot(x=col, y='proportion', hue='Price Category', data=combined_counts)
        plt.title(f'Proportion of {col} by Price Category')
        plt.xlabel(col)
        plt.ylabel('Proportion')
        plt.show()
```

Descriptive statistics for High-Priced Tickets:

	Journey Duration	Delay
count	1555.000000	1555.000000
mean	83.315113	12.932476
std	271.703414	19.448773
min	-1360.000000	0.000000
25%	110.000000	0.000000
50%	135.000000	0.000000
75%	150.000000	24.000000
max	277.000000	60.000000

Descriptive statistics for Not High-Priced Tickets:

	Journey Duration	Delay
count	30098.000000	30098.000000
mean	27.762742	2.546448
std	[illegible]	[illegible]
min	-1415.000000	0.000000
25%	30.000000	0.000000
50%	80.000000	0.000000
75%	80.000000	0.000000
max	288.000000	180.000000

Value counts for Categorical Columns (High-Priced Tickets):

Purchase Type:

Purchase Type	proportion
Station	0.62701
Online	0.37299

dtype: float64

Payment Method:

Payment Method	proportion
Credit Card	0.701608
Contactless	0.176849
Debit Card	0.121543

Analyze relationship between price and other variables

dtype: float64

Railcard:

Subtask:

proportion

Explore the correlation or relationship between 'Price' and other numerical and categorical variables to understand which factors might influence the ticket price.

Railcard	proportion
Unknown	0.865595
Adult	0.086817
Disabled	0.033441

Reasoning: Calculate and display the correlation matrix for the numerical columns and create a heatmap to visualize the relationships.

```
numerical_cols = ['Price', 'Journey Duration', 'Delay', 'Price_winsorized']
correlation_matrix = df[numerical_cols].corr()
print("Correlation Matrix for Numerical Columns:")
display(correlation_matrix)

plt.figure(figsize=(8, 6))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt=".2f")
plt.title('Correlation Heatmap of Numerical Columns')
plt.show()
```

Standard 0.704823

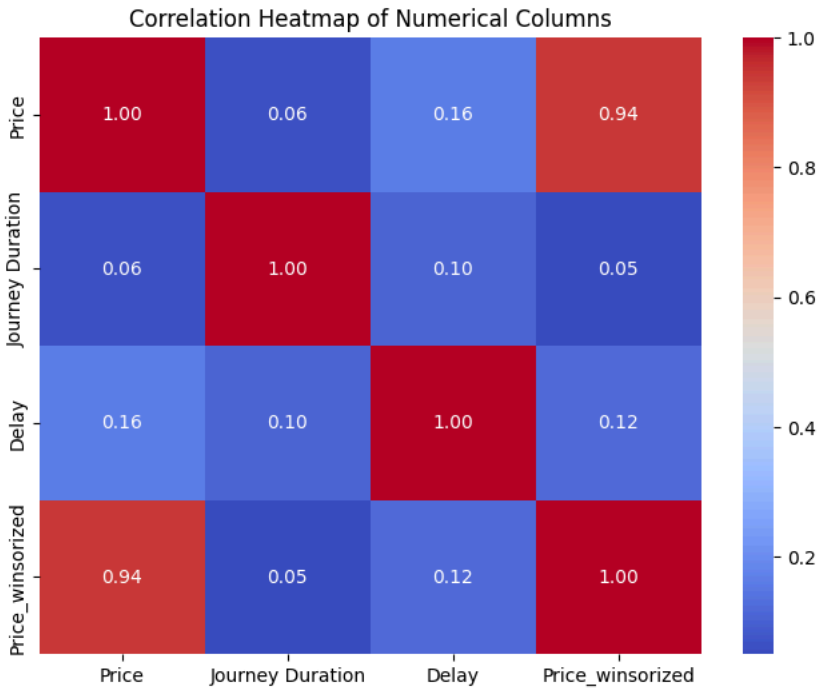
First Class 0.295177

Correlation Matrix for Numerical Columns:

	Price	Journey Duration	Delay	Price_winsorized
Price	1.000000	0.059552	0.158477	0.942523
Journey Duration	0.059552	1.000000	0.104464	0.050069
Delay	0.158477	0.104464	1.000000	0.117862
Price_winsorized	0.942523	0.050069	0.117862	1.000000

Ticket Type:

Ticket Type



Oxford 0.000643

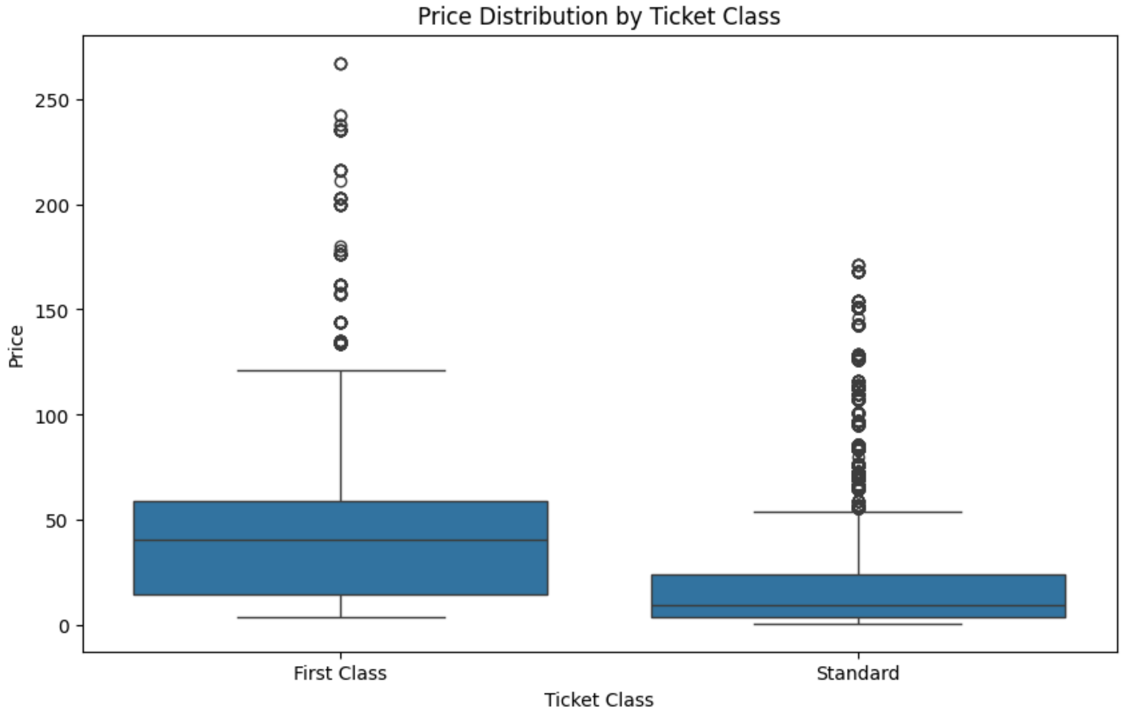
Reasoning: Create box plots to visualize the distribution of 'Price' for key categorical columns and calculate the average 'Price' for each category.

Bristol Temple Meads 0.000000

dtype: float64

```
categorical_cols_for_boxplot = ['Ticket Class', 'Ticket Type', 'Purchase Type', 'Journey Status', 'Payment Method']
for col in categorical_cols_for_boxplot:
    plt.figure(figsize=(10, 6))
    sns.boxplot(x=col, y='Price', data=df)
    plt.title(f'Price Distribution by {col}')
    plt.xlabel(col)
    plt.ylabel('Price')
    plt.show()

    print(f"\nAverage Price by {col}:")
    average_price_by_category = df.groupby(col, observed=True)['Price'].mean()
    display(average_price_by_category)
```

Liverpool Lime Street	0.030868
London St Pancras	0.028939
London Kings Cross	0.008360
Peterborough	0.005145
Edinburgh Waverley	0.002572

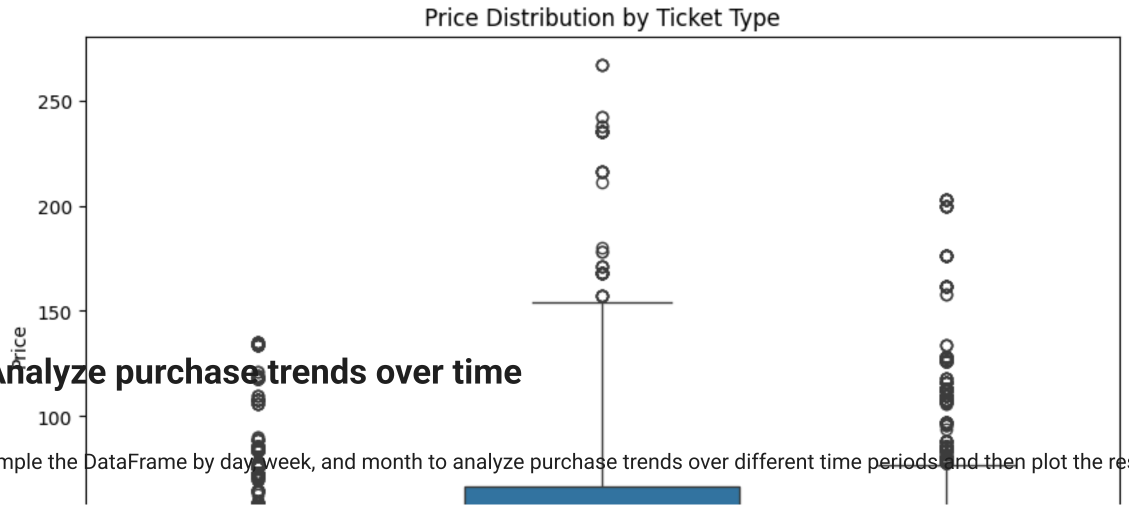


Stafford	0.000000
Tamworth	0.000000
Wakefield	0.000000
	0.000000
Wolverhampton	0.000000

```
Average Price by Ticket Class:
```

Ticket Class	Price
First Class	48.855134
Standard	20.721175

dtype: float64
dtype: float64



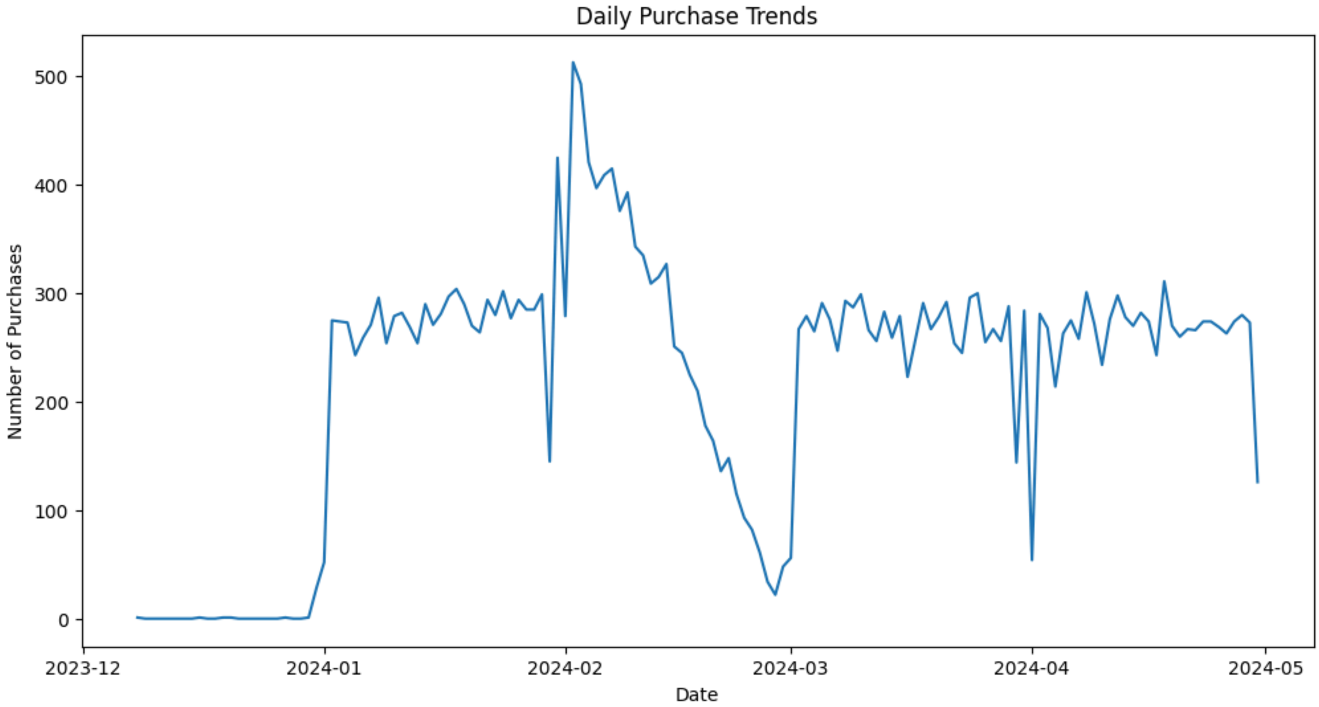
Analyze purchase trends over time

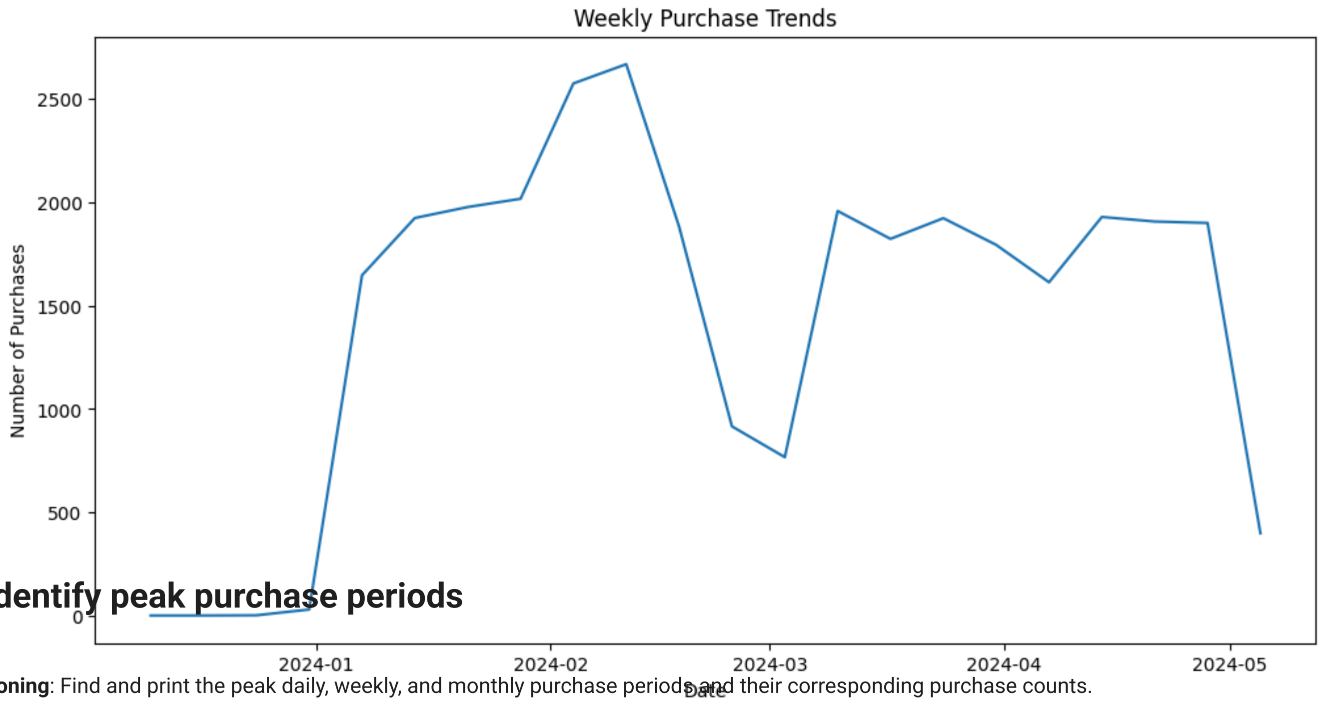
Resample the DataFrame by day, week, and month to analyze purchase trends over different time periods and then plot the results.

```
daily_purchases = df.set_index('Date of Purchase').resample('D').size().reset_index(name='purchase_count')
weekly_purchases = df.set_index('Date of Purchase').resample('W').size().reset_index(name='purchase_count')
monthly_purchases = df.set_index('Date of Purchase').resample('M').size().reset_index(name='purchase_count')

plt.figure(figsize=(12, 6))
sns.lineplot(x='Date of Purchase', y='purchase_count', data=daily_purchases)
plt.title('Daily Purchase Trends')
plt.xlabel('Date')
plt.ylabel('Number of Purchases')
plt.show()

plt.figure(figsize=(12, 6))
sns.lineplot(x='Date of Purchase', y='purchase_count', data=weekly_purchases)
plt.title('Weekly Purchase Trends')
plt.xlabel('Date')
plt.ylabel('Number of Purchases')
plt.show()

plt.figure(figsize=(12, 6))
sns.lineplot(x='Date of Purchase', y='purchase_count', data=monthly_purchases)
plt.title('Monthly Purchase Trends')
plt.xlabel('Date')
```

```
plt.ylabel('Number of Purchases')
plt.show()
```

```
/tmp/ipython-input-986164977.py:3: FutureWarning: 'M' is deprecated and will be removed in a future version, please use 'ME' inste
  monthly_purchases = df.set_index('Date of Purchase').resample('M').size().reset_index(name='purchase_count')
```





Identify peak purchase periods

Reasoning: Find and print the peak daily, weekly, and monthly purchase periods and their corresponding purchase counts.

```
peak_day = daily_purchases.loc[daily_purchases['purchase_count'].idxmax()]
peak_week = weekly_purchases.loc[weekly_purchases['purchase_count'].idxmax()]
peak_month = monthly_purchases.loc[monthly_purchases['purchase_count'].idxmax()]
print("Peak Daily Purchase Period:")
display(peak_day)
print("\nPeak Weekly Purchase Period:")
display(peak_week)
print("\nPeak Monthly Purchase Period:")
display(peak_month)
```

Peak Daily Purchase Period:

	56
Date of Purchase	2024-02-02 00:00:00
purchase_count	513

dtype: object

Peak Weekly Purchase Period:

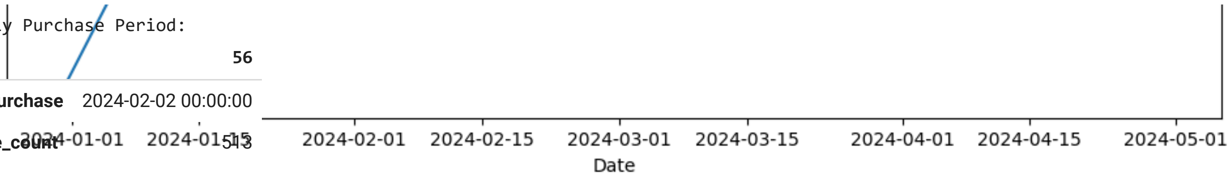
	9
Date of Purchase	2024-02-11 00:00:00
purchase_count	2668

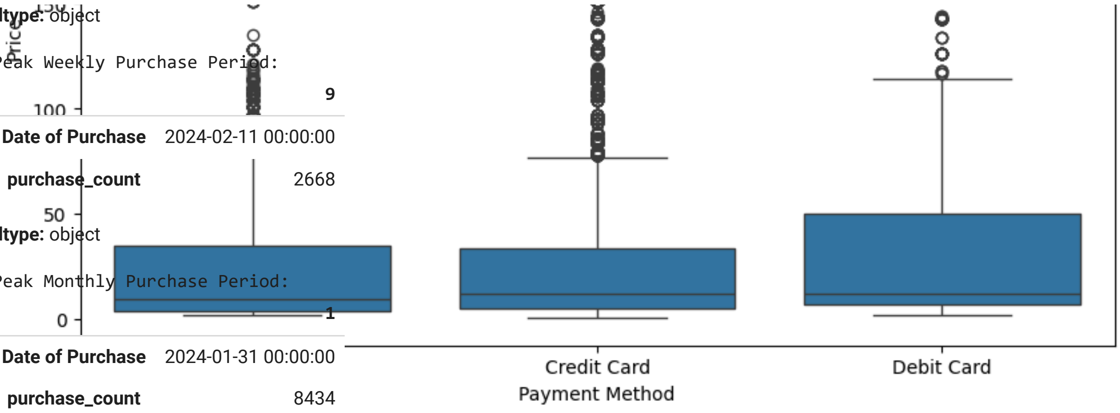
dtype: object

Peak Monthly Purchase Period:

	1
Date of Purchase	2024-01-31 00:00:00
purchase_count	8434

dtype: object





Average Price by Payment Method:

	Price
Payment Method	
Contactless	20.255123
Credit Card	24.535483
Debit Card	31.471182

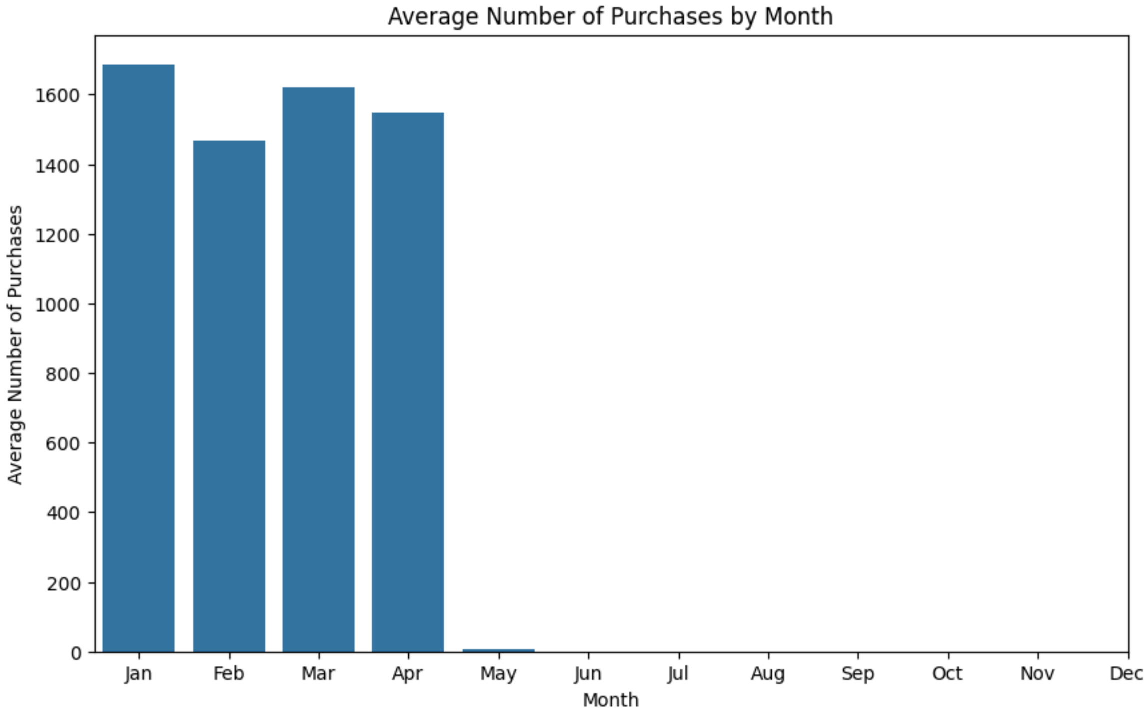
dtype: float64

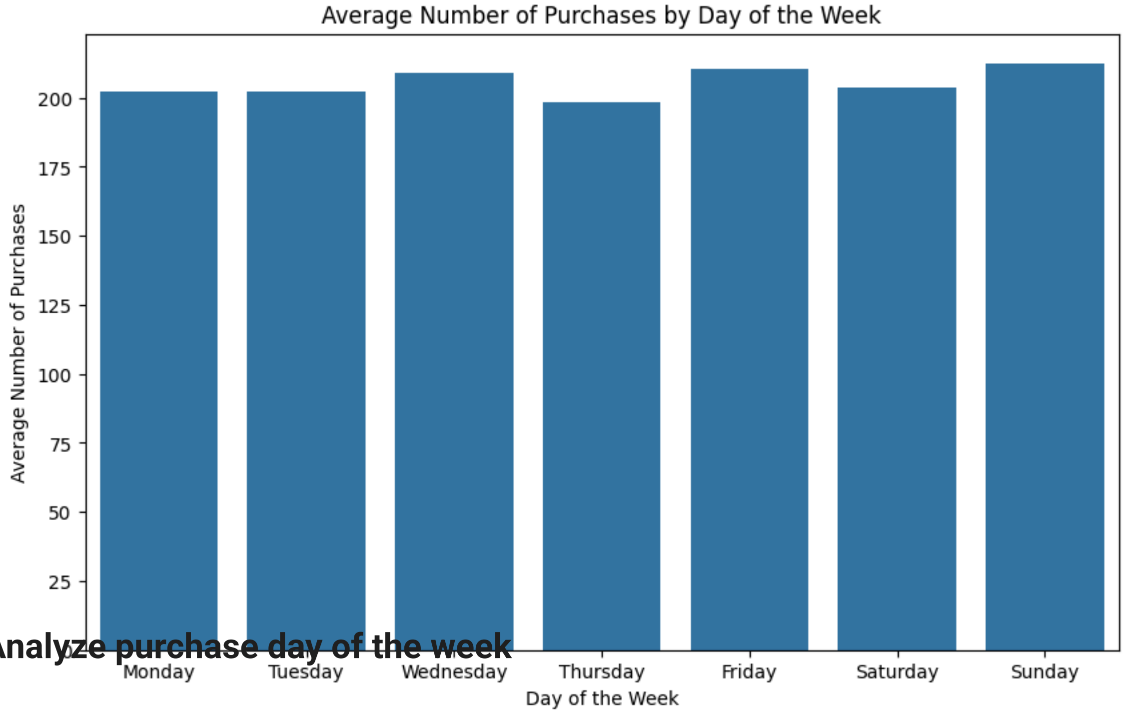
Manchester Piccadilly	0.173207
London Euston	0.155060
London Paddington	0.148947
	0.134361
	0.133065
London St Pancras	0.128082
	0.070304
York	0.030534
Leicester	0.011197
Sheffield	0.009037
Durham	0.008572
Leeds	0.008472
Peterborough	0.007775
Swindon	0.007575
Tamworth	0.007542
Nuneaton	0.007276
Doncaster	0.007010
London Paddington	0.006579
Crewe	0.006412
Stafford	0.006313

Explore seasonality

Extract the month and day of the week from the 'Date of Purchase' column and store them in new columns named 'purchase_month' and 'purchase_day_of_week'. Then calculate the average number of purchases for each month and each day of the week. Finally, create bar plots to visualize the average number of purchases by month and day of the week.

```
df['purchase_month'] = df['Date of Purchase'].dt.month
df['purchase_day_of_week'] = df['Date of Purchase'].dt.day_name()

average_purchases_by_month = df.groupby('purchase_month').size().reset_index(name='average_purchases')
average_purchases_by_month['average_purchases'] = average_purchases_by_month['average_purchases'] / len(df['Date of Purchase'].dt.to_per

average_purchases_by_day_of_week = df.groupby('purchase_day_of_week').size().reset_index(name='average_purchases')
days_order = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday']
average_purchases_by_day_of_week['purchase_day_of_week'] = pd.Categorical(average_purchases_by_day_of_week['purchase_day_of_week'], cate
average_purchases_by_day_of_week = average_purchases_by_day_of_week.sort_values('purchase_day_of_week')
average_purchases_by_day_of_week['average_purchases'] = average_purchases_by_day_of_week['average_purchases'] / len(df['Date of Purchase

plt.figure(figsize=(10, 6))
sns.barplot(x='purchase_month', y='average_purchases', data=average_purchases_by_month)
plt.title('Average Number of Purchases by Month')
plt.xlabel('Month')
plt.ylabel('Average Number of Purchases')
plt.xticks(ticks=range(12), labels=['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'])
plt.show()

plt.figure(figsize=(10, 6))
sns.barplot(x='purchase_day_of_week', y='average_purchases', data=average_purchases_by_day_of_week)
plt.title('Average Number of Purchases by Day of the Week')
plt.xlabel('Day of the Week')
plt.ylabel('Average Number of Purchases')
plt.show()
```





Analyze purchase day of the week

Calculate the number of purchases for each day of the week, sort the results, print the counts, and identify the day with the highest number of purchases.

Reason for Delay_Technical Issue	
False	0.979268
True	0.020732

```
purchases_by_day_of_week = df['purchase_day_of_week'].value_counts()
print("Number of purchases for each day of the week:")
display(purchases_by_day_of_week)
most_popular_day = purchases_by_day_of_week.idxmax()
print(f"\nThe day of the week with the most purchases is: {most_popular_day}")
```

Reason for Delay_Traffic	
False	0.990099
True	0.009901

dtype: float64

```
Reason for Delay_Unknown Reason:
```

Reason for Delay_Unknown Reason	proportion
True	0.883979
False	0.116021

Number of purchases for each day of the week:

purchase_day_of_week	count
Sunday	4676
Friday	4627
Wednesday	4602
Saturday	4477
Monday	4455
Tuesday	4454
Thursday	4362

dtype: int64

The day of the week with the most purchases is: Sunday

dtype: float64

Reason for Delay_Weather:

Reason for Delay_Weather	proportion
False	0.969267
True	0.030733

Refund Request:

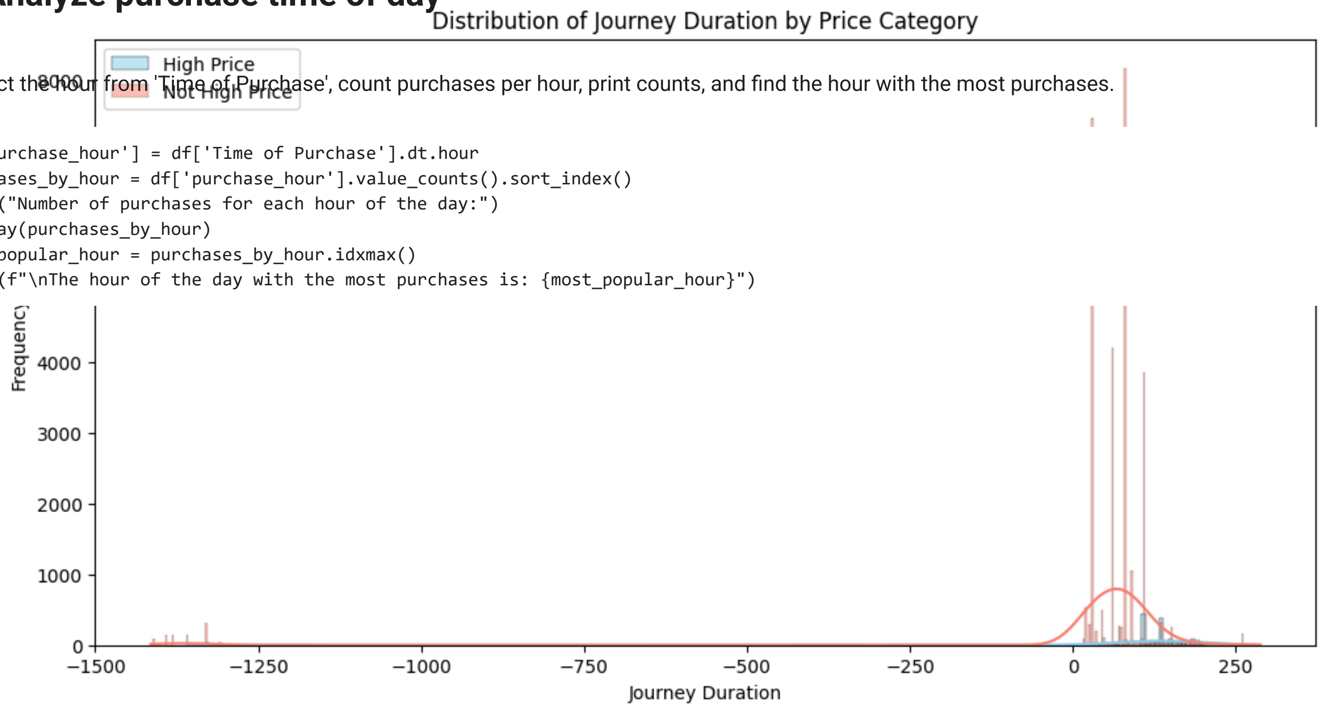
Refund Request	
No	0.96734
Yes	0.03266

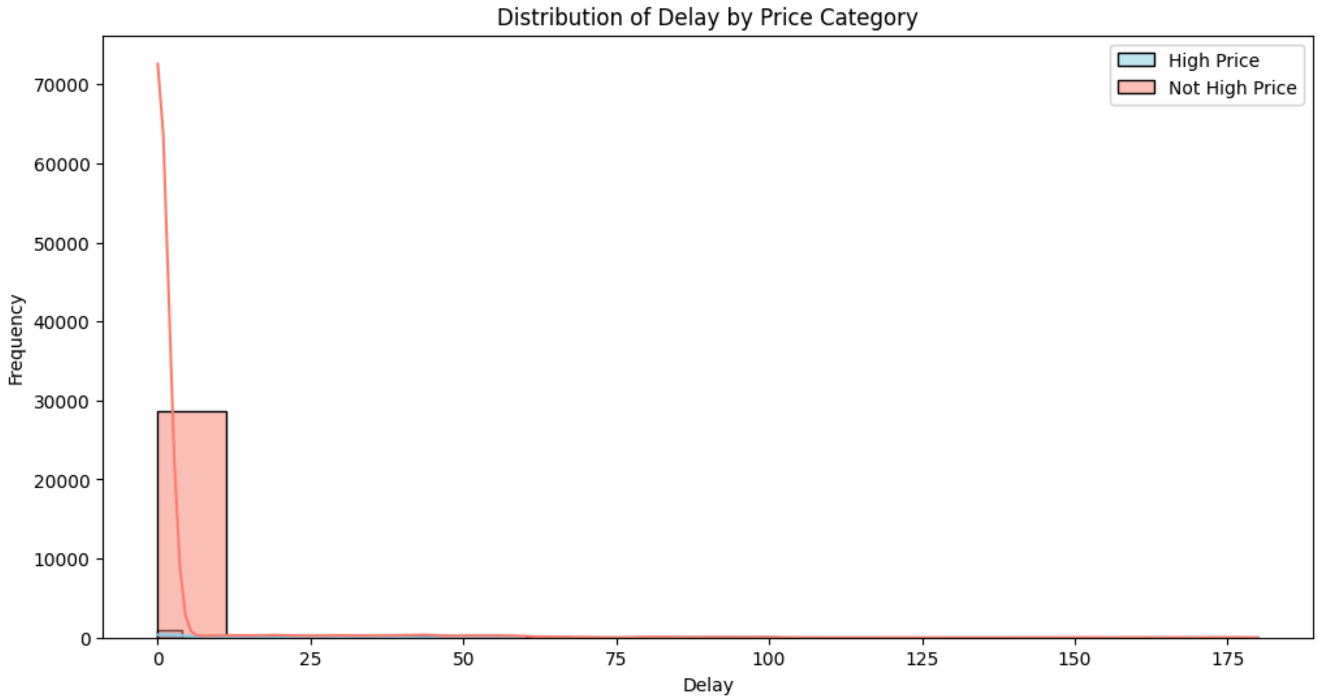
dtype: float64

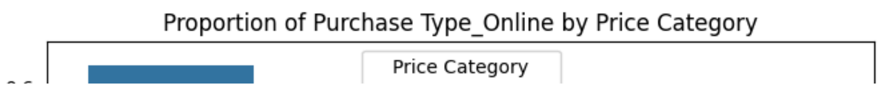
Analyze purchase time of day

Extract the hour from 'Time of Purchase', count purchases per hour, print counts, and find the hour with the most purchases.

```
df['purchase_hour'] = df['Time of Purchase'].dt.hour
purchases_by_hour = df['purchase_hour'].value_counts().sort_index()
print("Number of purchases for each hour of the day:")
display(purchases_by_hour)
most_popular_hour = purchases_by_hour.idxmax()
print(f"\nThe hour of the day with the most purchases is: {most_popular_hour}")
```





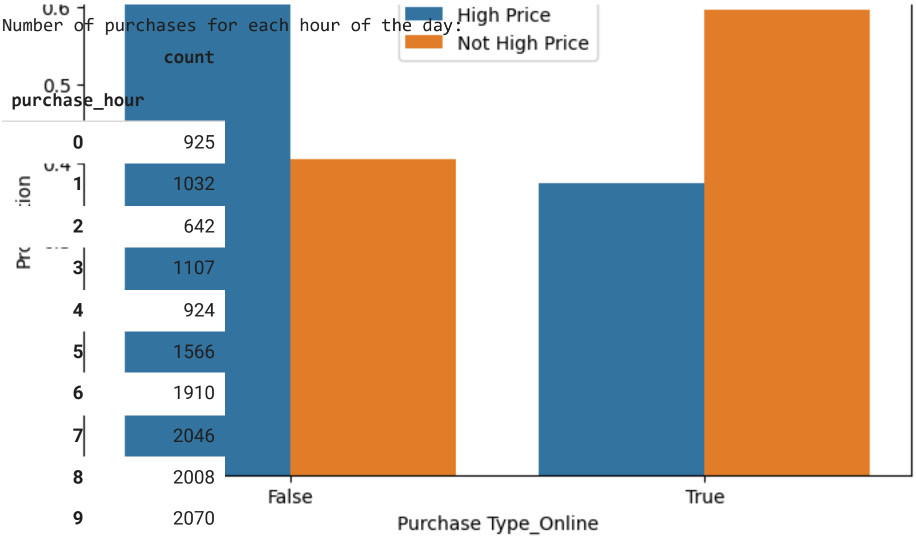


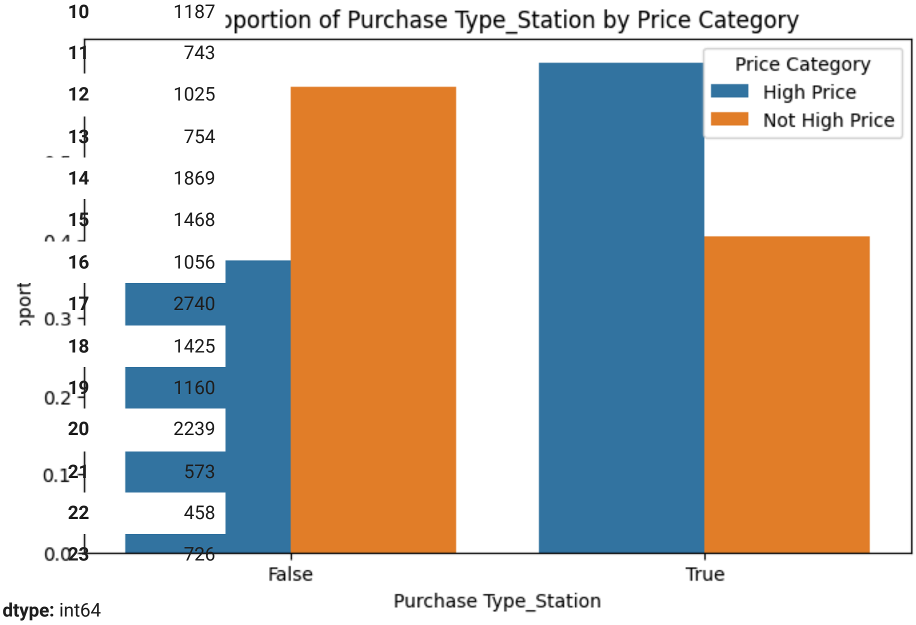
Number of purchases for each hour of the day:

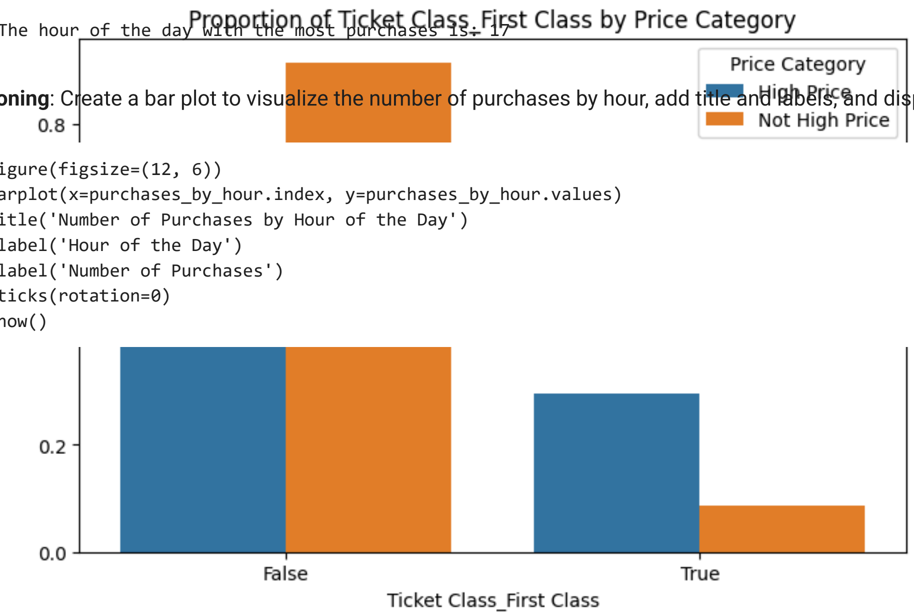
purchase_hour	count
0	925
1	1032
2	642
3	1107
4	924
5	1566
6	1910
7	2046
8	2008
9	2070
10	1187
11	743
12	1025
13	754
14	1869
15	1468
16	1056
17	2740
18	1425
19	1160
20	2239
21	573
22	458
23	726

dtype: int64

The hour of the day with the most purchases is: 17

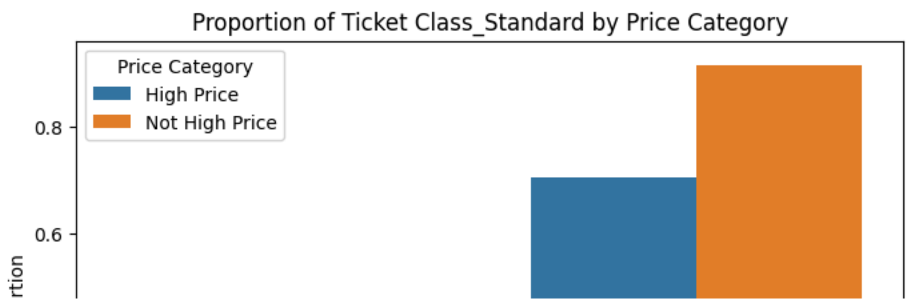


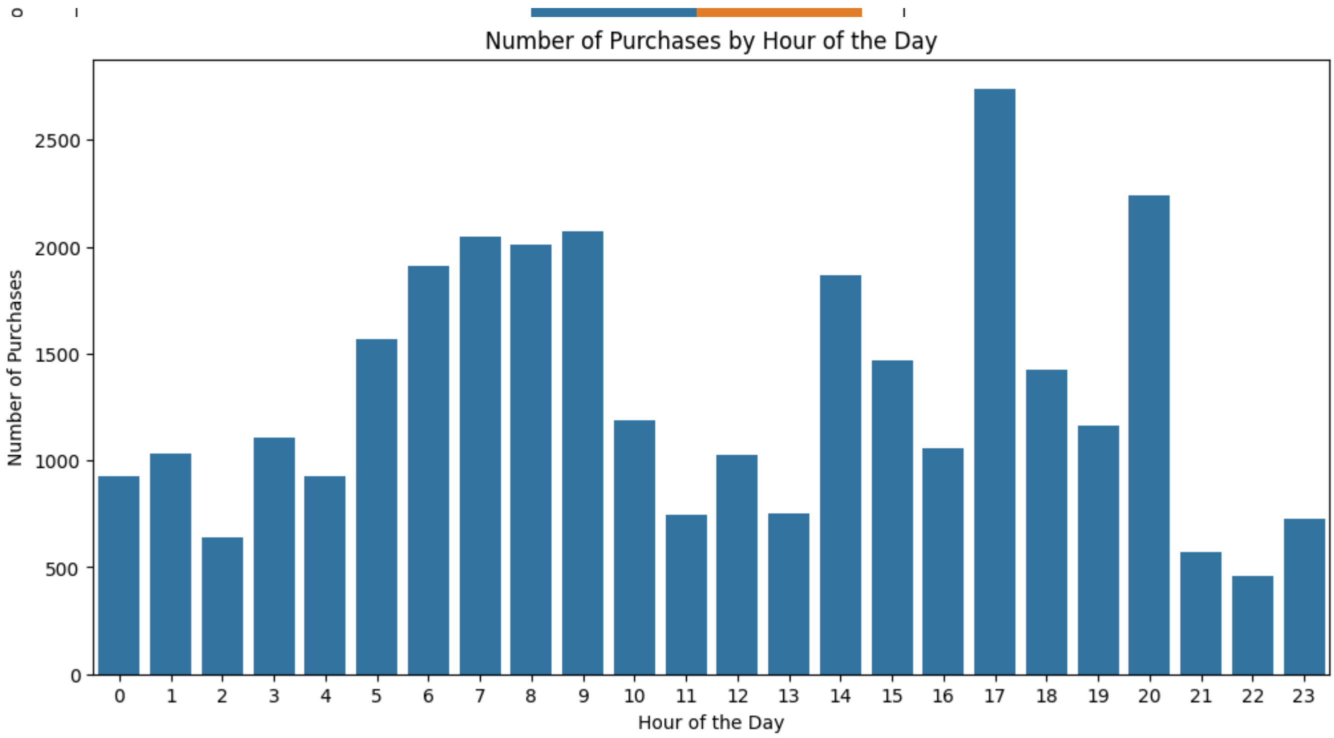


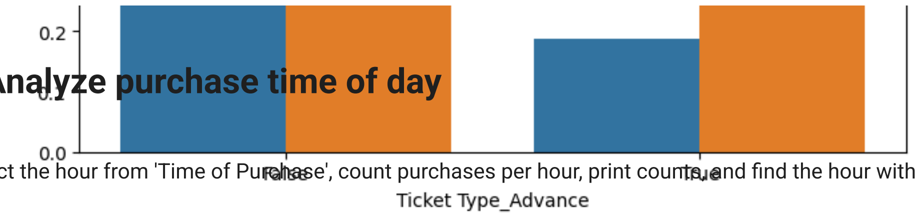


Reasoning: Create a bar plot to visualize the number of purchases by hour, add title and labels, and display the plot.

```
plt.figure(figsize=(12, 6))
sns.barplot(x=purchases_by_hour.index, y=purchases_by_hour.values)
plt.title('Number of Purchases by Hour of the Day')
plt.xlabel('Hour of the Day')
plt.ylabel('Number of Purchases')
plt.xticks(rotation=0)
plt.show()
```



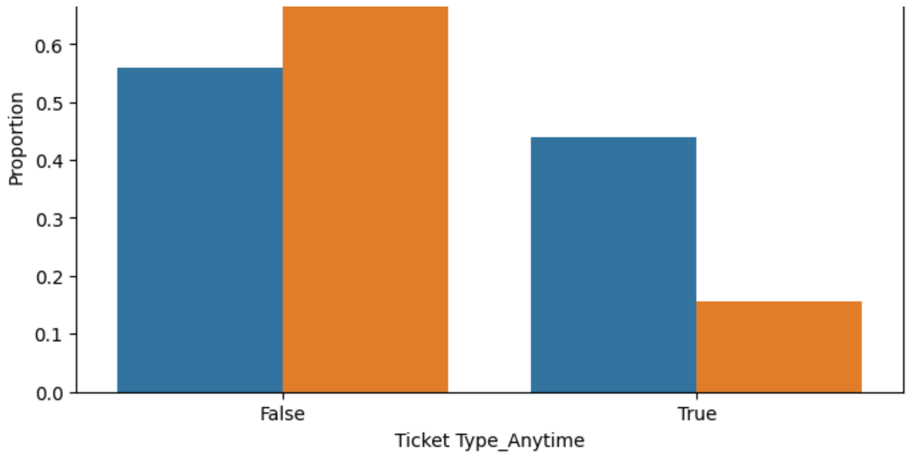


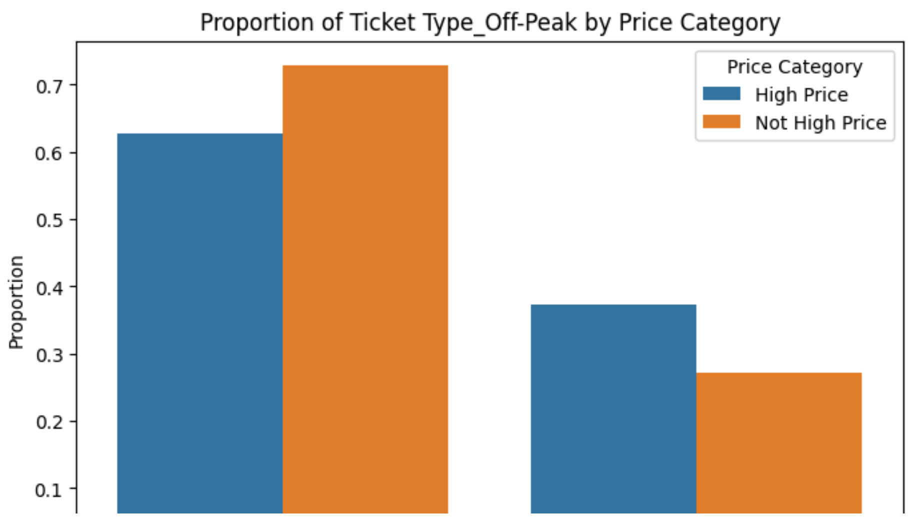


Analyze purchase time of day

Extract the hour from 'Time of Purchase', count purchases per hour, print counts, and find the hour with the most purchases.

```
df['purchase_hour'] = df['Time of Purchase'].dt.hour
purchases_by_hour = df['purchase_hour'].value_counts().sort_index()
print("Number of purchases for each hour of the day:")
display(purchases_by_hour)
most_popular_hour = purchases_by_hour.idxmax()
print(f"\nThe hour of the day with the most purchases is: {most_popular_hour}")
```



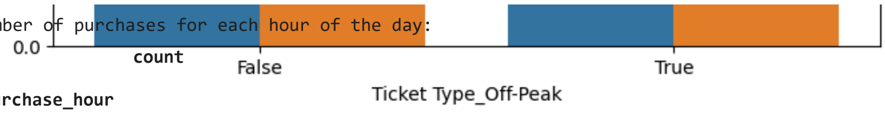


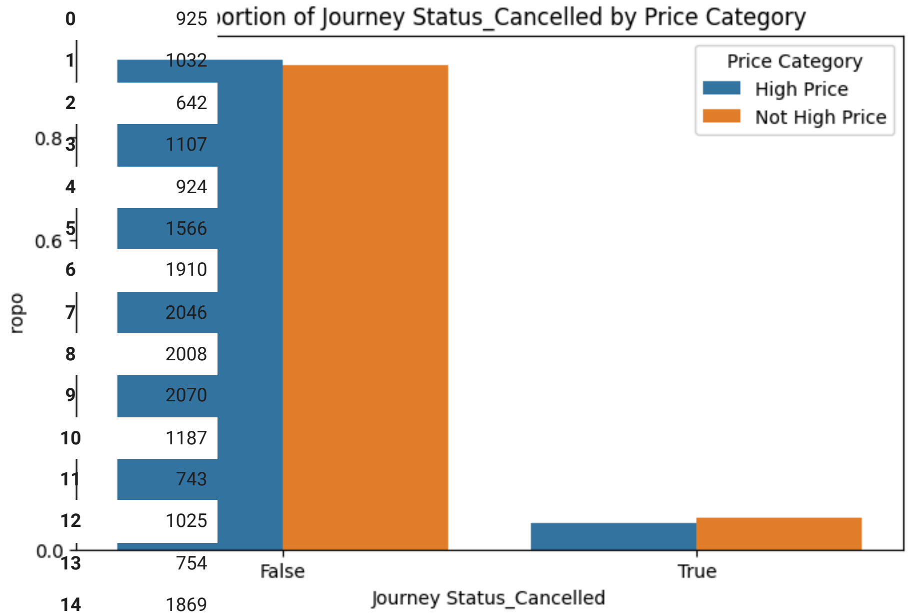
Number of purchases for each hour of the day:

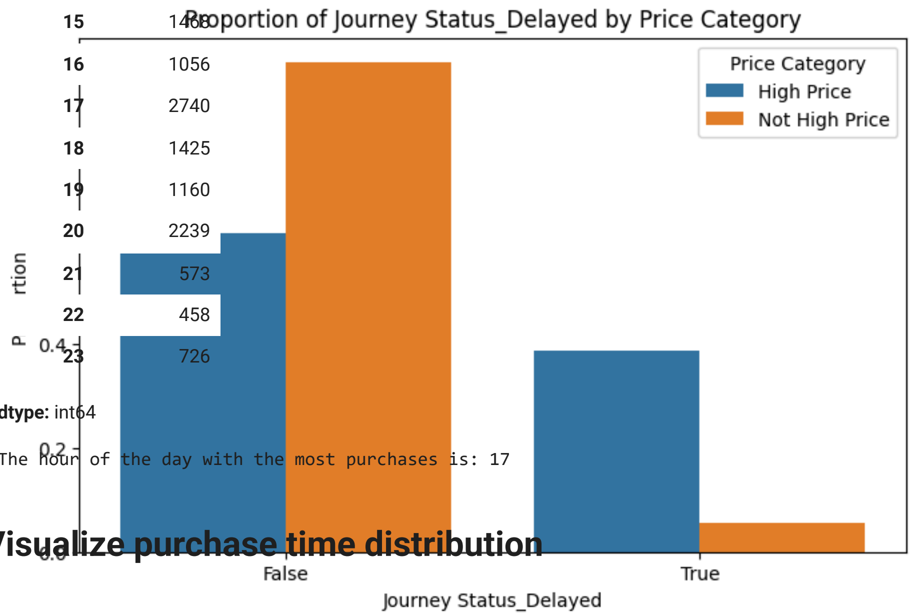
purchase_hour	count
0	925
1	1032
2	642
3	1107
4	924
5	1566
6	1910
7	2046
8	2008
9	2070
10	1187
11	743
12	1025
13	754
14	1869
15	1468
16	1056
17	2740
18	1425
19	1160
20	2239
21	573
22	458
23	726

dtype: int64

The hour of the day with the most purchases is: 17



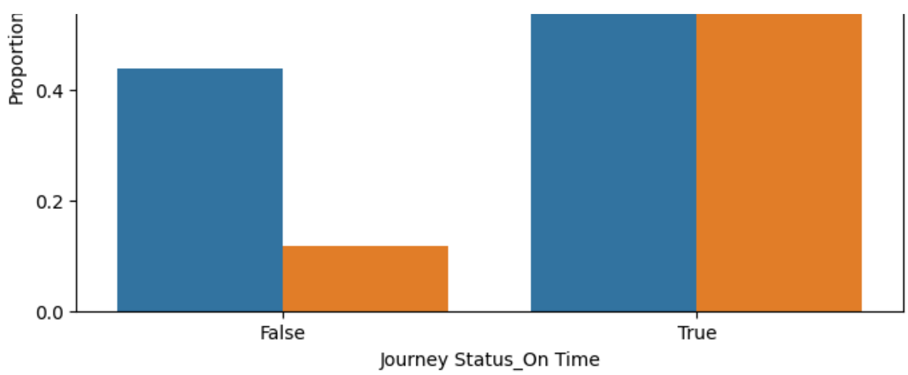


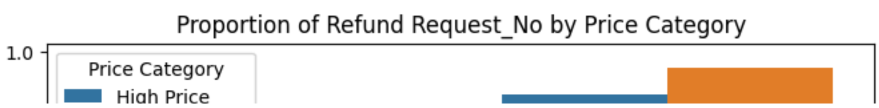


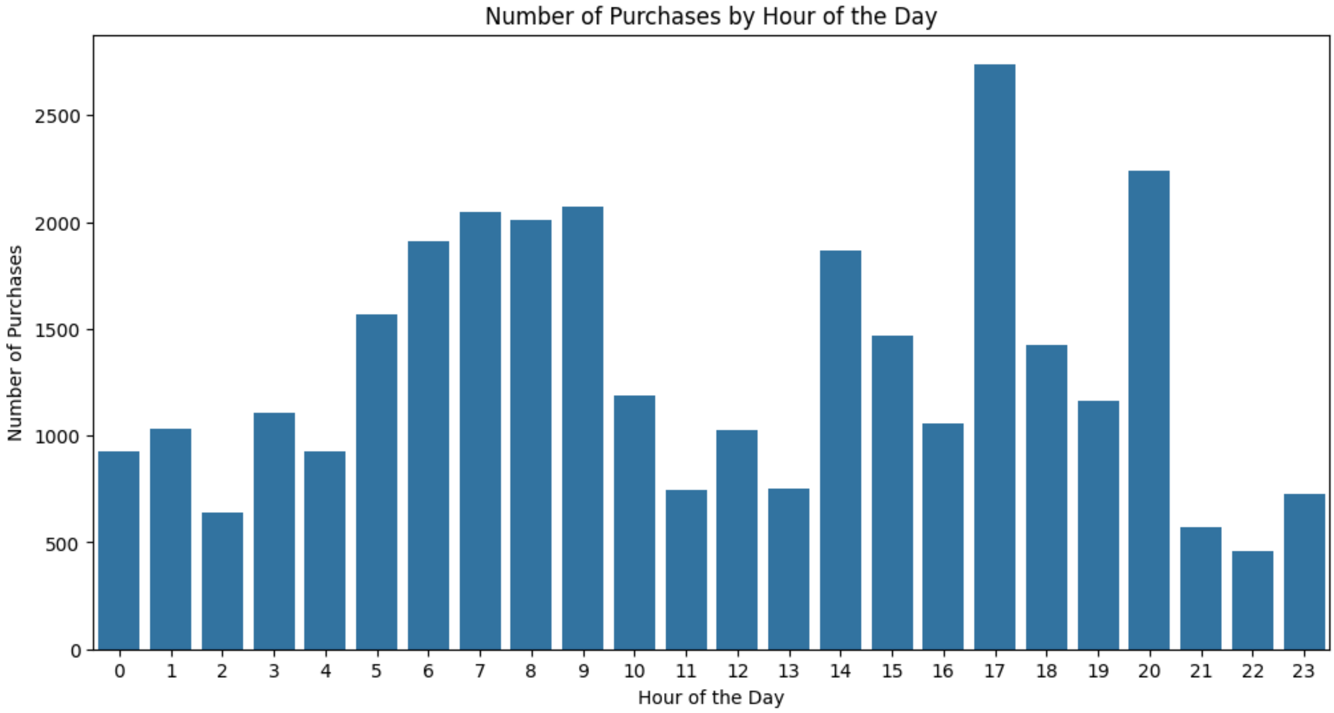
Visualize purchase time distribution

Create a bar plot to visualize the number of purchases by hour, add title and labels, and display the plot.

```
plt.figure(figsize=(12, 6))
sns.barplot(x=purchases_by_hour.index, y=purchases_by_hour.values)
plt.title('Number of Purchases by Hour of the Day')
plt.xlabel('Hour of the Day')
plt.ylabel('Number of Purchases')
plt.xticks(rotation=0)
plt.show()
```







Analyze departure time distribution

Extract the hour from 'Departure Time', count occurrences per hour, sort by hour, print counts, find the hour with the most occurrences, and print the peak hour.

```
df['departure_hour'] = df['Departure Time'].dt.hour
departures_by_hour = df['departure_hour'].value_counts().sort_index()
print("Number of departures for each hour of the day:")
display(departures_by_hour)
most_popular_departure_hour = departures_by_hour.idxmax()
print(f"\nThe hour of the day with the most departures is: {most_popular_departure_hour}")
```

```
Number of departures for each hour of the day:
```

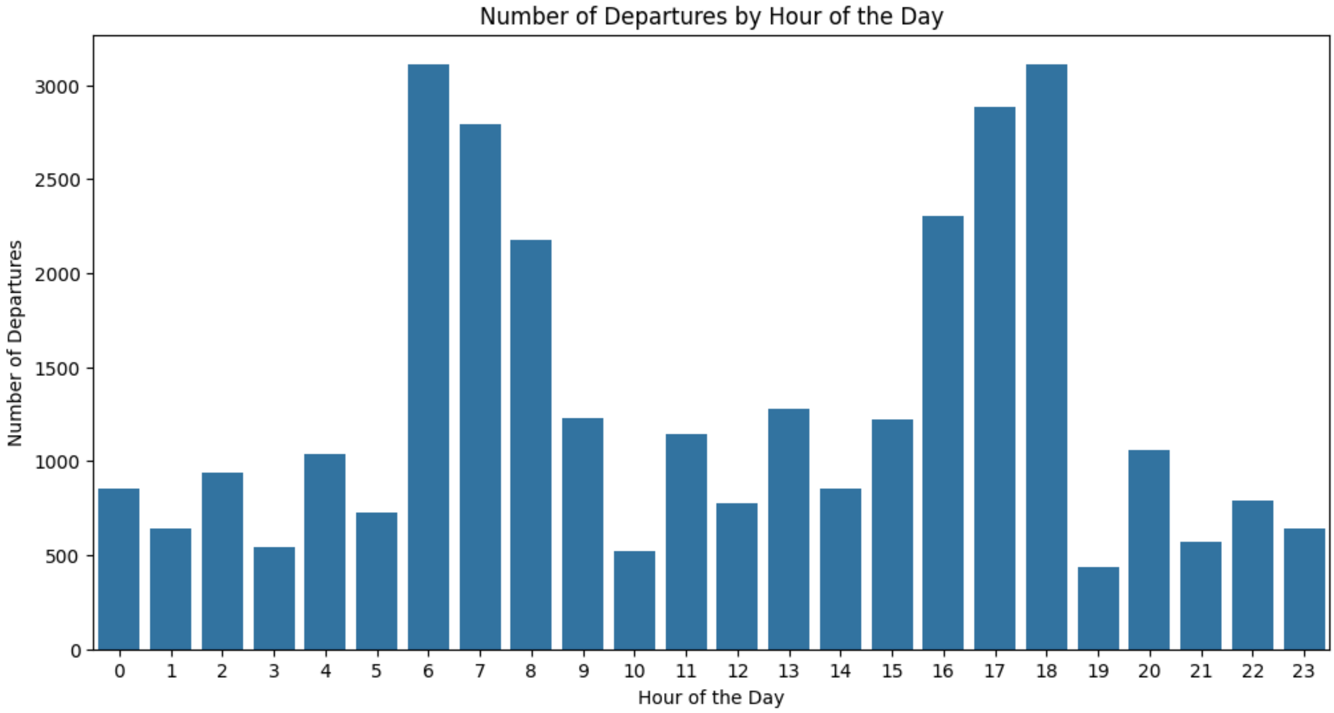
departure_hour	count
0	853
1	644
2	942
3	543
4	1041
5	725
6	3112
7	2795
8	2179
9	1230
10	525
11	1143
12	773
13	1276
14	855
15	1220
16	2301
17	2888
18	3113
19	438
20	1058
21	570
22	788
23	641

dtype: int64

```
The hour of the day with the most departures is: 18
```

Visualize the number of departures by hour using a bar plot to better understand the distribution and visually confirm the peak hours.

```
plt.figure(figsize=(12, 6))
sns.barplot(x=departures_by_hour.index, y=departures_by_hour.values)
plt.title('Number of Departures by Hour of the Day')
plt.xlabel('Hour of the Day')
plt.ylabel('Number of Departures')
plt.xticks(rotation=0)
plt.show()
```



Analyze arrival time distribution

Extract the hour from the 'Arrival Time' column, count the occurrences of each hour, sort the results, print the counts, and find the hour with the most arrivals.

```
df['arrival_hour'] = df['Arrival Time'].dt.hour
arrivals_by_hour = df['arrival_hour'].value_counts().sort_index()
print("Number of arrivals for each hour of the day:")
display(arrivals_by_hour)
most_popular_arrival_hour = arrivals_by_hour.idxmax()
print(f"\nThe hour of the day with the most arrivals is: {most_popular_arrival_hour}")
```

```
Number of arrivals for each hour of the day:
```

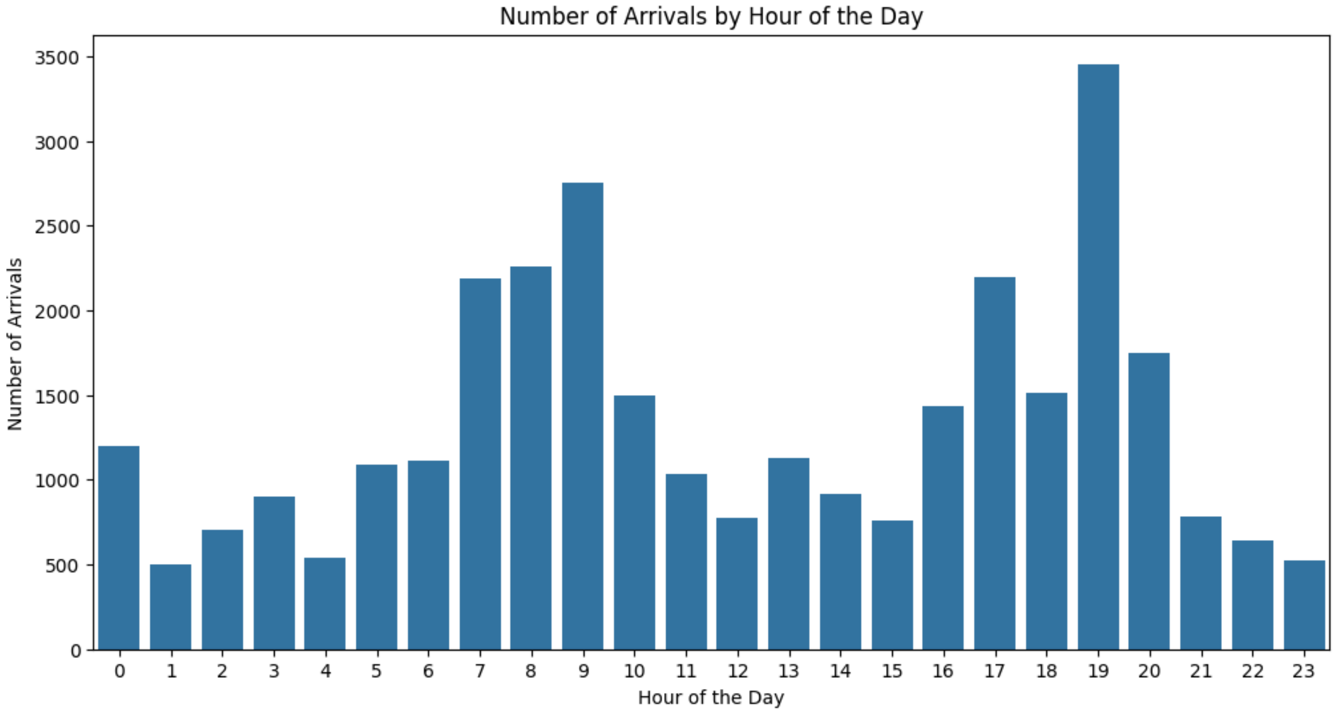
arrival_hour	count
0	1200
1	503
2	704
3	898
4	543
5	1090
6	1111
7	2185
8	2258
9	2752
10	1495
11	1037
12	775
13	1132
14	916
15	757
16	1436
17	2200
18	1510
19	3455
20	1751
21	780
22	645
23	520

dtype: int64

```
The hour of the day with the most arrivals is: 19
```

Create a bar plot to visualize the number of arrivals by hour, add title and labels, and display the plot.

```
plt.figure(figsize=(12, 6))
sns.barplot(x=arrivals_by_hour.index, y=arrivals_by_hour.values)
plt.title('Number of Arrivals by Hour of the Day')
plt.xlabel('Hour of the Day')
plt.ylabel('Number of Arrivals')
plt.xticks(rotation=0)
plt.show()
```



Identify peak travel times

Based on the outputs of the previous subtasks, identify and display the peak travel periods.

```
print(f"Peak Departure Hour: {most_popular_departure_hour}")
print(f"Peak Arrival Hour: {most_popular_arrival_hour}")
print("\nBased on the analysis of departure and arrival times, the overall peak travel periods are in the early morning (around 7-9 AM),
```

```
Peak Departure Hour: 18
Peak Arrival Hour: 19

Based on the analysis of departure and arrival times, the overall peak travel periods are in the early morning (around 7-9 AM), late
```

Geographical Analysis: Identifying Busiest Stations

```
# Calculate the number of departures from each station
departure_counts = df['Departure Station'].value_counts()

print("Number of departures from each station:")
display(departure_counts)
```

```
Number of departures from each station:
```

Departure Station	count
Manchester Piccadilly	5650
London Euston	4954
Liverpool Lime Street	4561
London Paddington	4500
London Kings Cross	4229
London St Pancras	3891
Birmingham New Street	2136
York	927
Reading	594
Oxford	144
Edinburgh Waverley	51
Bristol Temple Meads	16

dtype: int64

```
# Calculate the number of arrivals at each station
arrival_counts = df['Arrival Destination'].value_counts()

print("\nNumber of arrivals at each station:")
display(arrival_counts)
```

```
Number of arrivals at each station:
```

Arrival Destination	count
Birmingham New Street	7742
Liverpool Lime Street	5022
York	4019
Manchester Piccadilly	3968
Reading	3920
London Euston	1567
London St Pancras	749
Oxford	623
London Paddington	351
Leicester	337
Sheffield	272
Durham	258
Leeds	255
Peterborough	242
Swindon	228
Tamworth	227
Nuneaton	219
Doncaster	211
Crewe	193
Stafford	190
Edinburgh Waverley	178
Nottingham	158
Edinburgh	154
Bristol Temple Meads	144
Wolverhampton	115
London Kings Cross	84
London Waterloo	68
Coventry	65
Didcot	48
Cardiff Central	16
Warrington	15
Wakefield	15

Summary of Busiest Stations:

Based on the departure and arrival counts, we can identify the stations with the highest traffic.

```
print("\nTop 5 Busiest Departure Stations:")
display(departure_counts.head())

print("\nTop 5 Busiest Arrival Stations:")
display(arrival_counts.head())
```

```
Top 5 Busiest Departure Stations:
```

Departure Station	count
Manchester Piccadilly	5650
London Euston	4954
Liverpool Lime Street	4561
London Paddington	4500
London Kings Cross	4229

dtype: int64

```
Top 5 Busiest Arrival Stations:
```

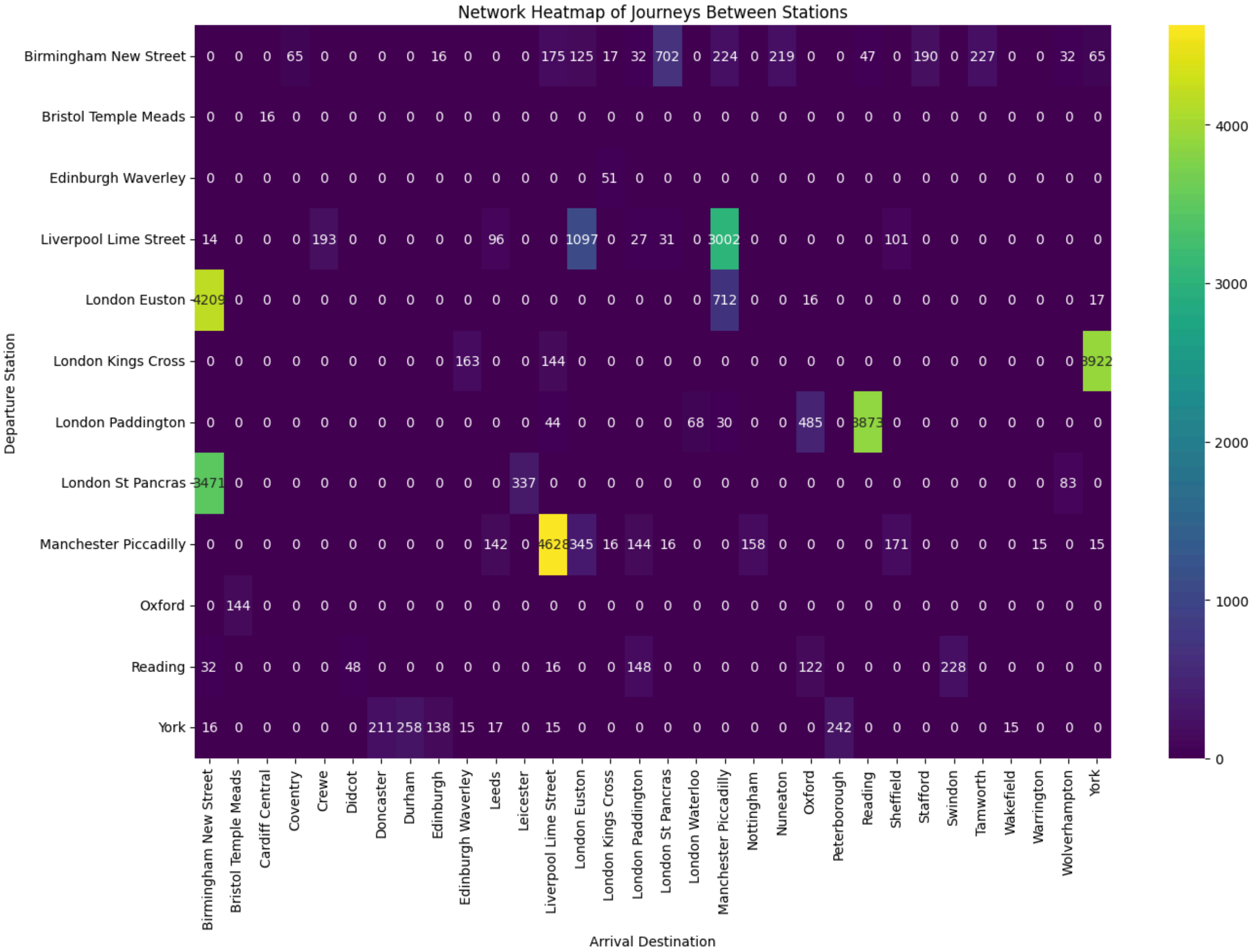
Arrival Destination	count
Birmingham New Street	7742
Liverpool Lime Street	5022
York	4019
Manchester Piccadilly	3968
Reading	3920

dtype: int64

Visualize Important Routes

```
# Create a pivot table to show the number of journeys between departure and arrival stations
route_matrix = df.pivot_table(index='Departure Station', columns='Arrival Destination', values='Transaction ID', aggfunc='count', fill_v

plt.figure(figsize=(14, 10))
sns.heatmap(route_matrix, annot=True, fmt='d', cmap='viridis')
plt.title('Network Heatmap of Journeys Between Stations')
plt.xlabel('Arrival Destination')
plt.ylabel('Departure Station')
plt.xticks(rotation=90)
plt.yticks(rotation=0)
plt.tight_layout()
plt.show()
```

```
/tmp/ipython-input-377728584.py:2: FutureWarning: The default value of observed=False is deprecated and will change to observed=True
  route_matrix = df.pivot_table(index='Departure Station', columns='Arrival Destination', values='Transaction ID', aggfunc='count',
```



Define routes

Concatenate the 'Departure Station' and 'Arrival Destination' columns to create the 'Route' column and display the head of the dataframe to confirm the creation of the new column.

```
df['Route'] = df['Departure Station'].astype(str) + ' to ' + df['Arrival Destination'].astype(str)
display(df.head())
```

	Transaction ID	Date of Purchase	Time of Purchase	Purchase Type	Payment Method	Railcard	Ticket Class	Ticket Type	Price	Departure Station	...	Reason for Delay_Traffic	Reasor Delay_Unk Re
0	da8a6ba8-b3dc-4677-b176	2023-12-08	1900-01-01 12:41:11	Online	Contactless	Adult	Standard	Advance	43	London Paddington	...	False	
1	b0cdd1b0-f214-4197-be53	2023-12-16	1900-01-01 11:23:01	Station	Credit Card	Adult	Standard	Advance	23	London Kings Cross	...	False	
2	f3ba7a96-f713-40d9-9629	2023-12-19	1900-01-01 19:51:27	Online	Credit Card	Unknown	Standard	Advance	3	Liverpool Lime Street	...	False	
3	b2471f11-4fe7-4c87-8ab4	2023-12-20	1900-01-01 23:00:36	Station	Credit Card	Unknown	Standard	Advance	13	London Paddington	...	False	
4	2be00b45-0762-485e-a7a3	2023-12-27	1900-01-01 18:22:56	Online	Contactless	Unknown	Standard	Advance	76	Liverpool Lime Street	...	False	

5 rows × 32 columns

The error indicates that the 'Departure Station' and 'Arrival Destination' columns are of categorical type and cannot be directly concatenated with a string. Convert these columns to string type before concatenation.

```
df['Route'] = df['Departure Station'].astype(str) + ' to ' + df['Arrival Destination'].astype(str)
display(df.head())
```

	Transaction ID	Date of Purchase	Time of Purchase	Purchase Type	Payment Method	Railcard	Ticket Class	Ticket Type	Price	Departure Station	...	Reason for Delay_Traffic	Reasor Delay_Unk Re
0	da8a6ba8-b3dc-4677-b176	2023-12-08	1900-01-01 12:41:11	Online	Contactless	Adult	Standard	Advance	43	London Paddington	...	False	
1	b0cdd1b0-f214-4197-be53	2023-12-16	1900-01-01 11:23:01	Station	Credit Card	Adult	Standard	Advance	23	London Kings Cross	...	False	
2	f3ba7a96-f713-40d9-9629	2023-12-19	1900-01-01 19:51:27	Online	Credit Card	Unknown	Standard	Advance	3	Liverpool Lime Street	...	False	
3	b2471f11-4fe7-4c87-8ab4	2023-12-20	1900-01-01 23:00:36	Station	Credit Card	Unknown	Standard	Advance	13	London Paddington	...	False	
4	2be00b45-0762-485e-a7a3	2023-12-27	1900-01-01 18:22:56	Online	Contactless	Unknown	Standard	Advance	76	Liverpool Lime Street	...	False	

5 rows × 32 columns

Identify popular routes

Calculate and print the value counts for the 'Route' column to identify the most popular routes and find and print the route with the highest number of purchases.

```
route_counts = df['Route'].value_counts()
print("Number of purchases for each route:")
display(route_counts)
most_popular_route = route_counts.idxmax()
print(f"\nThe most popular route is: {most_popular_route}")
```

```
Number of purchases for each route:
```

Route	count
Manchester Piccadilly to Liverpool Lime Street	4628
London Euston to Birmingham New Street	4209
London Kings Cross to York	3922
London Paddington to Reading	3873
London St Pancras to Birmingham New Street	3471
...	...
York to Edinburgh Waverley	15
York to Wakefield	15
York to Liverpool Lime Street	15
Manchester Piccadilly to Warrington	15
Liverpool Lime Street to Birmingham New Street	14

65 rows × 1 columns

dtype: int64

```
The most popular route is: Manchester Piccadilly to Liverpool Lime Street
```

Analyze popular routes

Calculate the average price, journey duration, and delay for each route and print them sorted.

```
average_price_by_route = df.groupby('Route')['Price'].mean()
average_journey_duration_by_route = df.groupby('Route')['Journey Duration'].mean()
average_delay_by_route = df.groupby('Route')['Delay'].mean()

print("Average Price by Route (Descending):")
display(average_price_by_route.sort_values(ascending=False))

print("\nAverage Journey Duration by Route (Descending):")
display(average_journey_duration_by_route.sort_values(ascending=False))

print("\nAverage Delay by Route (Descending):")
display(average_delay_by_route.sort_values(ascending=False))
```

Average Price by Route (Descending):

Route	Price
Manchester Piccadilly to London Paddington	114.111111
Liverpool Lime Street to London St Pancras	104.774194
Liverpool Lime Street to London Euston	103.280766
Liverpool Lime Street to London Paddington	99.962963
Manchester Piccadilly to London St Pancras	99.562500
...	...
Liverpool Lime Street to Manchester Piccadilly	3.980680
Manchester Piccadilly to Liverpool Lime Street	3.740277
Manchester Piccadilly to Warrington	3.533333
London Euston to Oxford	2.562500
Birmingham New Street to Wolverhampton	1.875000

65 rows × 1 columns

dtype: float64

Average Journey Duration by Route (Descending):

Route	Journey Duration
Edinburgh Waverley to London Kings Cross	275.274510
London Kings Cross to Edinburgh Waverley	260.000000
Liverpool Lime Street to London Paddington	168.481481
Liverpool Lime Street to London St Pancras	150.000000
London Paddington to Liverpool Lime Street	150.000000
...	...
London Euston to Manchester Piccadilly	-120.561798
Reading to London Paddington	-130.675676
Liverpool Lime Street to Leeds	-150.000000
York to Durham	-779.740310
Birmingham New Street to Edinburgh	-1170.000000

65 rows × 1 columns

dtype: float64

Average Delay by Route (Descending):

Synthesize the findings from the previous steps regarding popular routes and summarize the key insights about their characteristics.

Delay

```
print("Summary of Key Insights about Popular Routes:")
print("-" * 50)
print(f"The most popular route by number of purchases is: {most_popular_route} with {route_counts.loc[most_popular_route]} purchases.")
print("\nCharacteristics of Popular Routes (Top 10 by Purchase Count):")
top_10_routes = route_counts.head(10).index
for route in top_10_routes:
    avg_price = average_price_by_route.get(route, 'N/A')
    avg_duration = average_journey_duration_by_route.get(route, 'N/A')
    avg_delay = average_delay_by_route.get(route, 'N/A')
    print(f"\nRoute: {route}")
    print(f"  Average Price: {avg_price:.2f}" if isinstance(avg_price, (int, float)) else f"  Average Price: {avg_price}")
    print(f"  Average Journey Duration (minutes): {avg_duration:.2f}" if isinstance(avg_duration, (int, float)) else f"  Average Journey
    print(f"  Average Delay (minutes): {avg_delay:.2f}" if isinstance(avg_delay, (int, float)) else f"  Average Delay (minutes): {avg_de

print("\nComparison of Popular Routes:")
print("While Manchester Piccadilly to Liverpool Lime Street is the most popular route by volume, other routes may have different charact
print("For example, routes to/from London stations often have higher average prices.")
print("Routes with longer distances tend to have longer journey durations and potentially higher average delays.")
print("The 'Journey Status' analysis showed that popular routes like Manchester Piccadilly to Liverpool Lime Street have a significant p

print("\nOverall Takeaways:")
print("- Route popularity is not directly correlated with average price, journey duration, or delay.")
print("- High-volume routes can still experience significant delays.")
print("- Factors like distance, station location (e.g., London), and ticket class likely play a larger role in determining price and dur
```

```
Summary of Key Insights about Popular Routes:
--------------------------------------------------
The most popular route by number of purchases is: Manchester Piccadilly to Liverpool Lime Street with 4628 purchases.

Characteristics of Popular Routes (Top 10 by Purchase Count):

Route: Manchester Piccadilly to Liverpool Lime Street
  Average Price: 3.74
  Average Journey Duration (minutes): 35.14
  Average Delay (minutes): 5.14

Route: London Euston to Birmingham New Street
  Average Price: 11.96
  Average Journey Duration (minutes): 49.29
  Average Delay (minutes): 3.16

Route: London Kings Cross to York
  Average Price: 46.71
  Average Journey Duration (minutes): 40.41
  Average Delay (minutes): 0.54

Route: London Paddington to Reading
  Average Price: 16.88
  Average Journey Duration (minutes): 6.70
  Average Delay (minutes): 0.61

Route: London St Pancras to Birmingham New Street
  Average Price: 15.23
  Average Journey Duration (minutes): 65.89
  Average Delay (minutes): 0.00

Route: Liverpool Lime Street to Manchester Piccadilly
  Average Price: 3.98
  Average Journey Duration (minutes): -9.04
  Average Delay (minutes): 0.78

Route: Liverpool Lime Street to London Euston
  Average Price: 103.28
  Average Journey Duration (minutes): 138.69
  Average Delay (minutes): 26.01

Route: London Euston to Manchester Piccadilly
  Average Price: 85.68
  Average Journey Duration (minutes): -120.56
  Average Delay (minutes): 0.00

Route: Birmingham New Street to London St Pancras
  Average Price: 27.08
  Average Journey Duration (minutes): 80.00
  Average Delay (minutes): 0.00

Route: London Paddington to Oxford
  Average Price: 26.51
  Average Journey Duration (minutes): 90.00
  Average Delay (minutes): 0.00

Comparison of Popular Routes:
While Manchester Piccadilly to Liverpool Lime Street is the most popular route by volume, other routes may have different characte
```

Analyze journey status distribution

Calculate and print the value counts and proportions of the 'Journey Status' column and create a bar plot to visualize the distribution.

```
journey_status_counts = df['Journey Status'].value_counts()
print("Number of journeys for each status:")
display(journey_status_counts)
journey_status_proportions = df['Journey Status'].value_counts(normalize=True)
print("\nProportion of journeys for each status:")
display(journey_status_proportions)
plt.figure(figsize=(8, 5))
sns.barplot(x=journey_status_proportions.index, y=journey_status_proportions.values)
plt.title('Distribution of Journey Status')
plt.xlabel('Journey Status')
plt.ylabel('Proportion')
plt.show()
```

Number of journeys for each status:

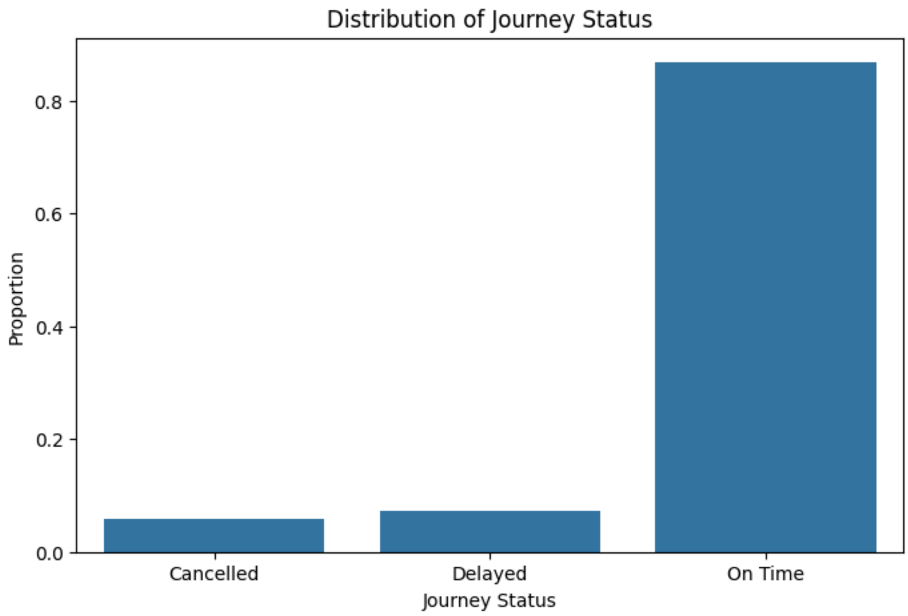
Journey Status	count
On Time	27481
Delayed	2292
Cancelled	1880

dtype: int64

Proportion of journeys for each status:

Journey Status	proportion
On Time	0.868196
Delayed	0.072410
Cancelled	0.059394

dtype: float64



Investigate reasons for delay

Calculate value counts and proportions for 'Reason for Delay' and visualize the proportions with a bar plot to understand the most common causes of delays.

```
reason_cols = [col for col in df.columns if col.startswith('Reason for Delay_')]
reason_for_delay_counts = df[reason_cols].sum().sort_values(ascending=False)
print("Number of occurrences for each reason for delay:")
display(reason_for_delay_counts)

reason_for_delay_proportions = reason_for_delay_counts / len(df)
print("\nProportion of each reason for delay:")
display(reason_for_delay_proportions)

plt.figure(figsize=(10, 6))
sns.barplot(x=reason_for_delay_proportions.index, y=reason_for_delay_proportions.values)
plt.title('Proportion of Journeys by Reason for Delay')
plt.xlabel('Reason for Delay')
plt.ylabel('Proportion')
plt.xticks(rotation=45, ha='right')
plt.tight_layout()
plt.show()
```

Number of occurrences for each reason for delay:

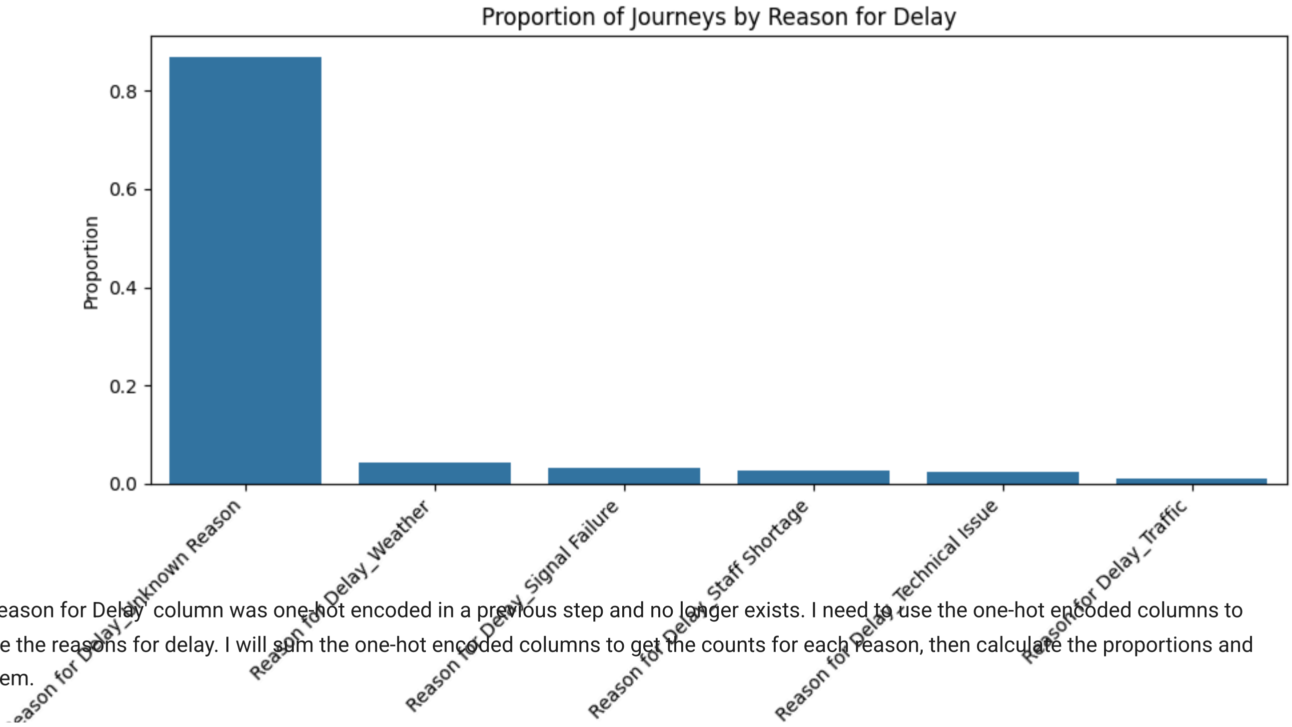
	0
Reason for Delay_Unknown Reason	27481
Reason for Delay_Weather	1372
Reason for Delay_Signal Failure	970
Reason for Delay_Staff Shortage	809
Reason for Delay_Technical Issue	707
Reason for Delay_Traffic	314

dtype: int64

Proportion of each reason for delay:

	0
Reason for Delay_Unknown Reason	0.868196
Reason for Delay_Weather	0.043345
Reason for Delay_Signal Failure	0.030645
Reason for Delay_Staff Shortage	0.025558
Reason for Delay_Technical Issue	0.022336
Reason for Delay_Traffic	0.009920

dtype: float64



The 'Reason for Delay' column was one-hot encoded in a previous step and no longer exists. I need to use the one-hot encoded columns to analyze the reasons for delay. I will sum the one-hot encoded columns to get the counts for each reason, then calculate the proportions and plot them.

```
reason_cols = [col for col in df.columns if col.startswith('Reason for Delay_')]
reason_for_delay_counts = df[reason_cols].sum().sort_values(ascending=False)
print("Number of occurrences for each reason for delay:")
display(reason_for_delay_counts)
reason_for_delay_proportions = reason_for_delay_counts / len(df)
print("\nProportion of each reason for delay:")
display(reason_for_delay_proportions)
plt.figure(figsize=(10, 6))
sns.barplot(x=reason_for_delay_proportions.index, y=reason_for_delay_proportions.values)
plt.title('Proportion of Journeys by Reason for Delay')
plt.xlabel('Reason for Delay')
plt.ylabel('Proportion')
plt.xticks(rotation=45, ha='right')
plt.tight_layout()
plt.show()
```

Number of occurrences for each reason for delay:

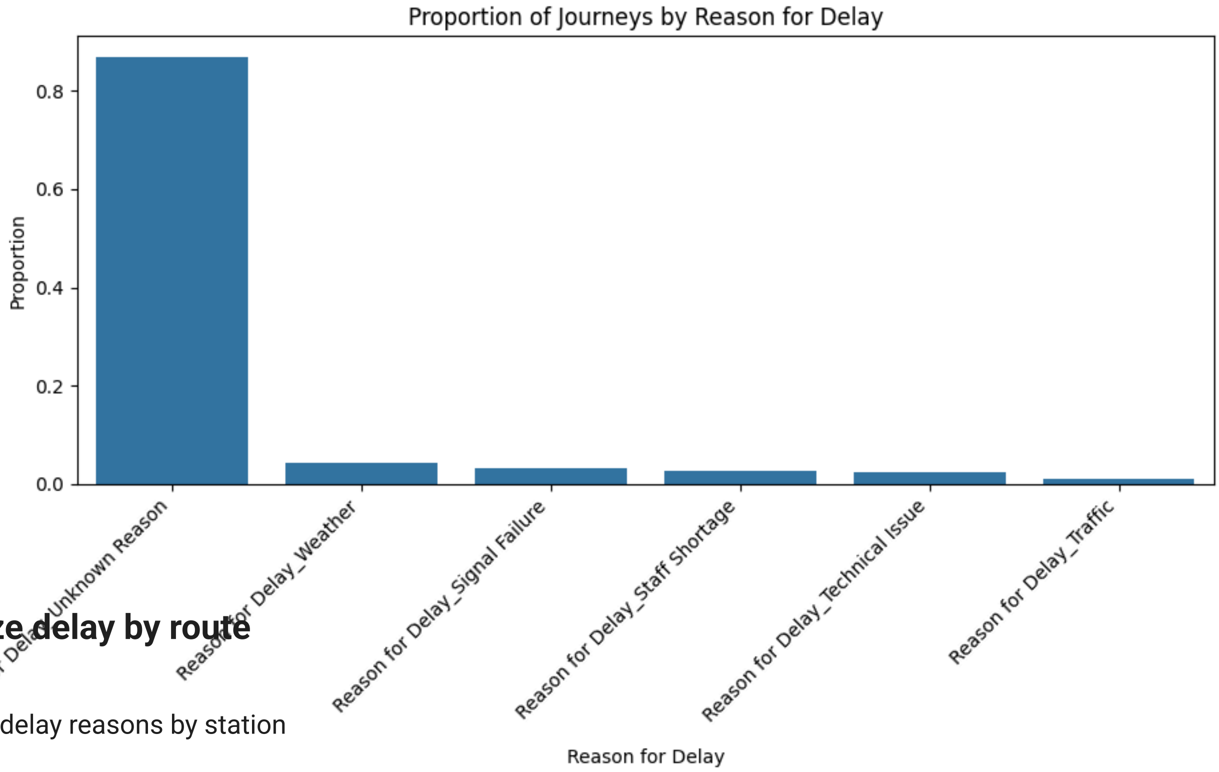
	0
Reason for Delay_Unknown Reason	27481
Reason for Delay_Weather	1372
Reason for Delay_Signal Failure	970
Reason for Delay_Staff Shortage	809
Reason for Delay_Technical Issue	707
Reason for Delay_Traffic	314

dtype: int64

Proportion of each reason for delay:

	0
Reason for Delay_Unknown Reason	0.868196
Reason for Delay_Weather	0.043345
Reason for Delay_Signal Failure	0.030645
Reason for Delay_Staff Shortage	0.025558
Reason for Delay_Technical Issue	0.022336
Reason for Delay_Traffic	0.009920

dtype: float64



Analyze delay by route

Analyze delay reasons by station

```
# Calculate the proportion of each delay reason for each departure station
delay_reason_by_departure_station = df.groupby('Departure Station')[reason_cols].mean()
print("Proportion of each delay reason by Departure Station:")
display(delay_reason_by_departure_station)

# Calculate the proportion of each delay reason for each arrival station
delay_reason_by_arrival_station = df.groupby('Arrival Destination')[reason_cols].mean()
print("\nProportion of each delay reason by Arrival Station:")
display(delay_reason_by_arrival_station)

# Visualize the proportion of each delay reason by Departure Station for the top N stations
N = 10 # You can adjust this number
top_departure_stations = departure_counts.head(N).index
delay_reason_by_departure_station_top_N = delay_reason_by_departure_station.loc[top_departure_stations]
delay_reason_by_departure_station_top_N = delay_reason_by_departure_station_top_N.transpose()

plt.figure(figsize=(14, 8))
delay_reason_by_departure_station_top_N.plot(kind='bar', stacked=True, figsize=(14,8))
plt.title('Proportion of Delay Reasons by Top 10 Departure Stations')
plt.xlabel('Reason for Delay')
plt.ylabel('Proportion')
```

```
plt.xticks(rotation=45, ha='right')
plt.legend(title='Departure Station', bbox_to_anchor=(1.05, 1), loc='upper left')
plt.tight_layout()
plt.show()

# Visualize the proportion of each delay reason by Arrival Station for the top N stations
top_arrival_stations = arrival_counts.head(N).index
delay_reason_by_arrival_station_top_N = delay_reason_by_arrival_station.loc[top_arrival_stations]
delay_reason_by_arrival_station_top_N = delay_reason_by_arrival_station_top_N.transpose()

plt.figure(figsize=(14, 8))
delay_reason_by_arrival_station_top_N.plot(kind='bar', stacked=True, figsize=(14,8))
plt.title('Proportion of Delay Reasons by Top 10 Arrival Stations')
plt.xlabel('Reason for Delay')
plt.ylabel('Proportion')
plt.xticks(rotation=45, ha='right')
plt.legend(title='Arrival Station', bbox_to_anchor=(1.05, 1), loc='upper left')
plt.tight_layout()
plt.show()
```

```
Proportion of each delay reason by Departure Station:
/tmp/ipython-input-4183821481.py:2: FutureWarning: The default of observed=False is deprecated and will be changed to True in a futu
  delay_reason_by_departure_station = df.groupby('Departure Station')[reason_cols].mean()
```

Departure Station	Reason for Delay_Signal Failure	Reason for Delay_Staff Shortage	Reason for Delay_Technical Issue	Reason for Delay_Traffic	Reason for Delay_Unknown Reason	Reason for Delay_Weather
Birmingham New Street	0.011236	0.026217	0.050562	0.001404	0.874532	0.036049
Bristol Temple Meads	0.000000	0.000000	0.000000	0.000000	1.000000	0.000000
Edinburgh Waverley	0.000000	1.000000	0.000000	0.000000	0.000000	0.000000
Liverpool Lime Street	0.021487	0.026749	0.054155	0.016005	0.751370	0.130235
London Euston	0.044207	0.015543	0.008680	0.005854	0.896044	0.029673
London Kings Cross	0.034524	0.007567	0.016316	0.008040	0.919839	0.013715
London Paddington	0.016444	0.030889	0.018222	0.010222	0.906000	0.018222
London St Pancras	0.028527	0.012593	0.010280	0.007967	0.925469	0.015163
Manchester Piccadilly	0.044425	0.039646	0.018584	0.015398	0.825310	0.056637
Oxford	0.111111	0.000000	0.013889	0.006944	0.854167	0.013889
Reading	0.015152	0.013468	0.006734	0.003367	0.951178	0.010101
York	0.023732	0.055016	0.007551	0.008630	0.875944	0.029126

```
Proportion of each delay reason by Arrival Station:
/tmp/ipython-input-4183821481.py:7: FutureWarning: The default of observed=False is deprecated and will be changed to True in a futu
  delay_reason_by_arrival_station = df.groupby('Arrival Destination')[reason_cols].mean()
```

Arrival Destination	Reason for Delay_Signal Failure	Reason for Delay_Staff Shortage	Reason for Delay_Technical Issue	Reason for Delay_Traffic	Reason for Delay_Unknown Reason	Reason for Delay_Weather
Birmingham New Street	0.038491	0.015242	0.008783	0.006458	0.905709	0.025316
Bristol Temple Meads	0.111111	0.000000	0.013889	0.006944	0.854167	0.013889
Cardiff Central	0.000000	0.000000	0.000000	0.000000	1.000000	0.000000
Coventry	0.000000	0.046154	0.000000	0.030769	0.923077	0.000000
Crewe	0.000000	0.000000	0.010363	0.000000	0.974093	0.015544

Calculate the proportion of delayed journeys for each route, sort the results, and print the top N routes with the highest proportion of delays.

```
delayed_journeys = df[df['Journey Status'] == 'Delayed']
delayed_proportion_by_route = delayed_journeys.groupby('Route').size() / df.groupby('Route').size()
delayed_proportion_by_route = delayed_proportion_by_route.sort_values(ascending=False)
N = 15
print(f"Top {N} Routes with Highest Proportion of Delayed Journeys:")
display(delayed_proportion_by_route.head(N))
```

Top 15 Routes with Highest Proportion of Delayed Journeys:

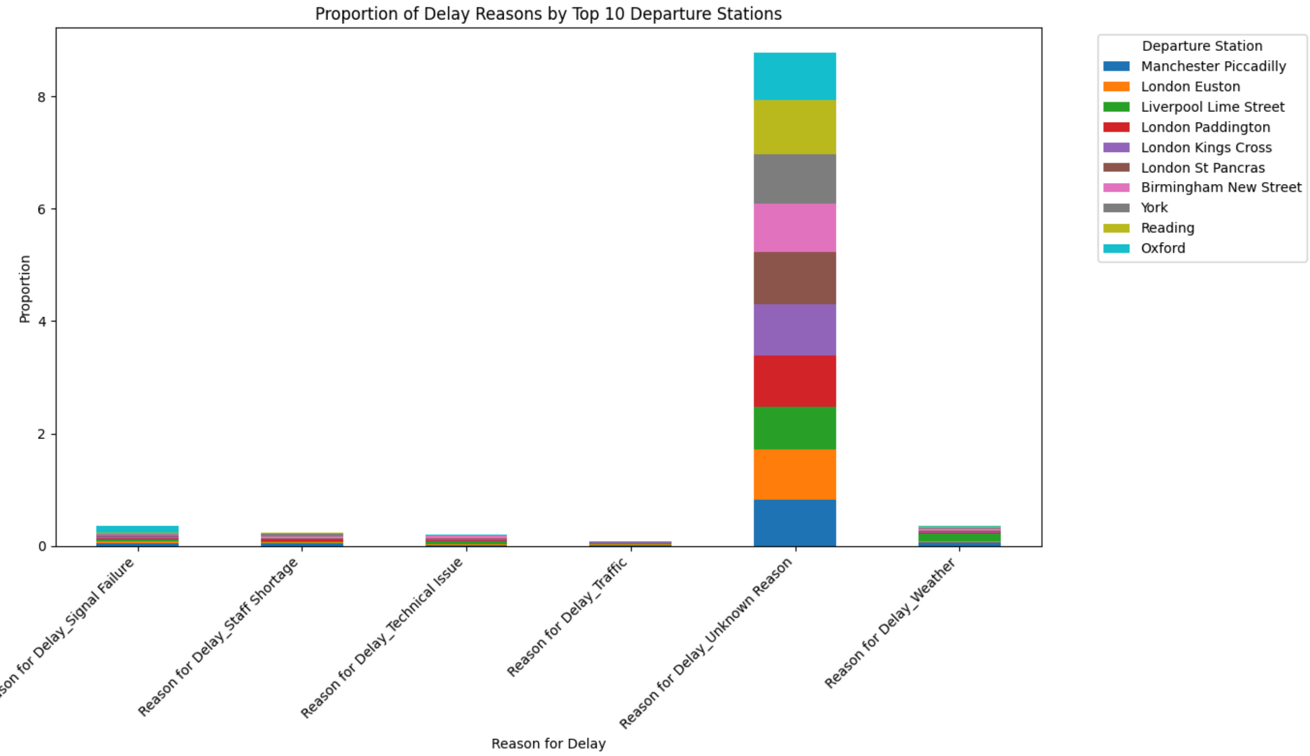
Route	0
London Euston to York	1.000000
Edinburgh Waverley to London Kings Cross	1.000000
York to Wakefield	1.000000
Liverpool Lime Street to London Euston	0.711030
Manchester Piccadilly to London Euston	0.695652
Liverpool Lime Street to London Paddington	0.481481
Manchester Piccadilly to Leeds	0.450704
Birmingham New Street to Manchester Piccadilly	0.428571
Birmingham New Street to London Euston	0.352000
York to Doncaster	0.127962
Oxford to Bristol Temple Meads	0.104167
Manchester Piccadilly to Nottingham	0.088608
Manchester Piccadilly to Liverpool Lime Street	0.076491
York to Durham	0.062016
London Euston to Birmingham New Street	0.057496

dtype: float64

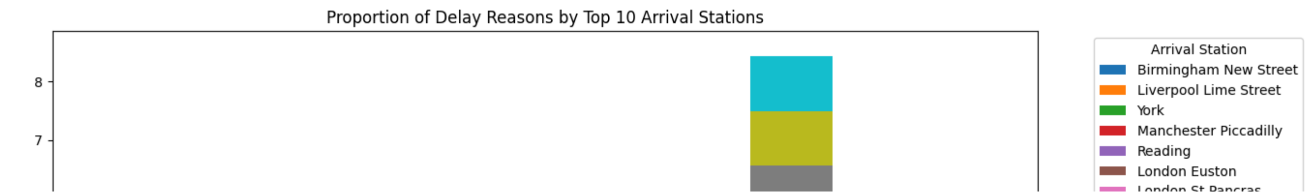
Edinburgh Waverley	0.005618	0.011236	0.022472	0.000000	0.949438	0.011236
Leeds	0.250980	0.000000	0.000000	0.003922	0.745098	0.000000
Leicester	[illegible]	[illegible]	0.017804	0.005935	0.952522	0.020772
Liverpool Lime Street	0.033453	0.037634	0.011151	0.013540	0.870569	0.033652
[illegible]	[illegible]	[illegible]	0.121889	0.038290	0.252712	0.469049
London Kings Cross	0.000000	0.607143	0.000000	0.000000	0.392857	0.000000
London Paddington	[illegible]	0.045584	0.005698	0.005698	0.914530	0.011396
London St Pancras	[illegible]	[illegible]	0.008011	0.000000	0.927904	0.024032
[illegible]	[illegible]	[illegible]	0.000000	0.000000	0.911765	0.000000
Manchester Piccadilly	0.018145	[illegible]	0.052671	0.009073	0.901210	0.010837
[illegible]	[illegible]	[illegible]	0.000000	0.012658	0.835443	0.018987
Nuneaton	0.000000	0.013699	0.004566	0.004566	0.936073	0.041096
[illegible]	[illegible]	[illegible]	0.014446	0.016051	0.889246	0.035313
Peterborough	0.000000	0.057851	0.000000	0.028926	0.909091	0.004132
[illegible]	[illegible]	[illegible]	0.018878	0.009184	0.909439	0.015306
Sheffield	0.007353	0.000000	0.000000	0.000000	0.992647	0.000000
Stafford	0.000000	0.000000	0.000000	0.000000	0.978947	0.021053
Swindon	0.000000	0.017544	0.008772	0.004386	0.947368	0.021930

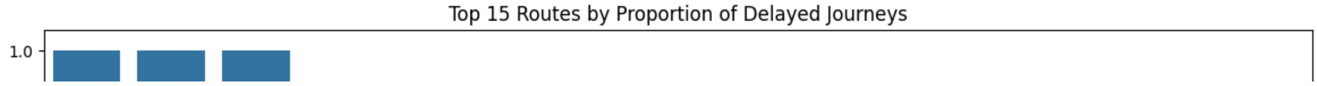
Create a bar plot to visualize the proportion of delayed journeys for the top N routes.

```
plt.figure(figsize=(12, 8))
sns.barplot(x=delayed_proportion_by_route.head(N).index, y=delayed_proportion_by_route.head(N).values)
plt.title(f'Top {N} Routes by Proportion of Delayed Journeys')
plt.xlabel('Route')
plt.ylabel('Proportion of Delayed Journeys')
plt.xticks(rotation=90, ha='right')
plt.tight_layout()
plt.show()
```



<Figure size 1400x800 with 0 Axes>




Top 15 Routes by Proportion of Delayed Journeys
1.0