Module 2: NCR Ride Bookings - Complete Project Guide

Data Manipulation & Exploration with Pandas and NumPy

DS Club | Hands-On Project
Duration: 4 Sessions (90 minutes each)
Dataset: 150,000 ride booking records from NCR region

📋 Table of Contents

Project Overview

Business Context

You are a **Data Analyst** at a ride-hailing company operating in the NCR (National Capital Region). The company has been experiencing:

- Increasing ride cancellations
- Fluctuating revenues across different locations
- Unclear patterns in customer and driver behavior

Your Mission: Analyze 150,000 ride booking records to uncover insights that will help the business:

1. Reduce cancellation rates
2. Identify profitable locations
3. Understand peak demand times
4. Improve customer and driver satisfaction

Dataset Information

File: `ncr_ride_bookings.csv`
Size: 150,000 rows × 21 columns
Period: January - March 2022
Regions: Delhi, Gurgaon, Noida, and surrounding NCR areas

Columns:

1. **Date** - Date of booking
2. **Time** - Time of booking
3. **Booking ID** - Unique identifier for each booking
4. **Booking Status** - Status of the ride (Success, Cancelled, etc.)
5. **Customer ID** - Unique customer identifier

6. **Vehicle Type** - Type of vehicle (Auto, Bike, Prime Sedan, Prime SUV)
7. **Pickup Location** - Starting point of ride
8. **Drop Location** - Destination
9. **Avg VTAT** - Average Vehicle Turn Around Time
10. **Avg CTAT** - Average Customer Turn Around Time
11. **Cancelled Rides by Customer** - Number of cancellations by customer
12. **Reason for cancelling by Customer** - Why customer cancelled
13. **Cancelled Rides by Driver** - Number of cancellations by driver
14. **Driver Cancellation Reason** - Why driver cancelled
15. **Incomplete Rides** - Number of incomplete rides
16. **Incomplete Rides Reason** - Why ride was incomplete
17. **Booking Value** - Fare amount in local currency
18. **Ride Distance** - Distance in kilometers
19. **Driver Ratings** - Rating given to driver (1-5)
20. **Customer Rating** - Rating given to customer (1-5)
21. **Payment Method** - How customer paid (Cash, Card, Wallet, UPI)

Session 1: Data Loading & Initial Exploration

Duration: 90 minutes
Goal: Load the data and understand its structure

Part 1: Setting Up Your Workspace (10 minutes)

Step 1.1: Import Required Libraries

python

```python
# Import libraries for data manipulation
import pandas as pd          # For working with tabular data
import numpy as np           # For numerical operations
import matplotlib.pyplot as plt  # For creating visualizations
import seaborn as sns        # For statistical visualizations

# Display settings
pd.set_option('display.max_columns', None)  # Show all columns
pd.set_option('display.width', None)        # Don't truncate display
pd.set_option('display.max_rows', 100)      # Show up to 100 rows

print("✅ Libraries imported successfully!")
```

What Each Library Does:

- **pandas (pd):** Think of it as Excel on steroids - lets you work with tables of data (called DataFrames)
- **numpy (np):** For math operations, especially on arrays of numbers
- **matplotlib (plt):** Creates charts and graphs
- **seaborn (sns):** Makes matplotlib prettier and easier to use

The `pd.set_option()` commands:

- Help us see more data when we print DataFrames
- Makes debugging easier
- You can adjust these based on your screen size

Question 2: Revenue by Vehicle Type

python

```python
# Analyze revenue patterns by vehicle type
vehicle_revenue = df_clean.groupby('Vehicle Type').agg({
    'Booking Value': ['sum', 'mean', 'median', 'count'],
    'Ride Distance': 'mean'
}).round(2)

# Flatten column names
vehicle_revenue.columns = ['Total_Revenue', 'Avg_Booking', 'Median_Booking', 'Num_Rides', 'Avg_Distance']
vehicle_revenue = vehicle_revenue.sort_values('Total_Revenue', ascending=False)

print("Revenue Analysis by Vehicle Type:")
print(vehicle_revenue)

# Visualize - Multiple subplots
fig, axes = plt.subplots(1, 2, figsize=(15, 6))

# Plot 1: Total Revenue
vehicle_revenue['Total_Revenue'].plot(kind='bar', ax=axes[0], color='teal')
axes[0].set_title('Total Revenue by Vehicle Type', fontsize=14, fontweight='bold')
axes[0].set_ylabel('Revenue (₹)')
axes[0].set_xlabel('Vehicle Type')
axes[0].tick_params(axis='x', rotation=45)
axes[0].grid(True, alpha=0.3, axis='y')

# Plot 2: Average Booking Value
vehicle_revenue['Avg_Booking'].plot(kind='bar', ax=axes[1], color='coral')
axes[1].set_title('Average Booking Value by Vehicle Type', fontsize=14, fontweight='bold')
axes[1].set_ylabel('Avg Booking (₹)')
axes[1].set_xlabel('Vehicle Type')
axes[1].tick_params(axis='x', rotation=45)
axes[1].grid(True, alpha=0.3, axis='y')

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

**Business Step 1.2: Load the Dataset

python

```python
# Load the CSV file into a DataFrame
df = pd.read_csv('data/ncr_ride_bookings.csv')

# Display success message
print("✅ Dataset loaded successfully!")
print(f"Shape: {df.shape}")
```

Understanding `pd.read_csv()`:

- **What it does:** Reads a CSV (Comma Separated Values) file and converts it to a DataFrame
- **Parameters:**
  - `'data/ncr_ride_bookings.csv'` - Path to your file
  - You can add: `encoding='utf-8'` if you get encoding errors
  - You can add: `parse_dates=['Date']` to automatically convert dates

Understanding `df.shape`:

- Returns a tuple: `(number_of_rows, number_of_columns)`
- Example output: `(150000, 21)` means 150,000 rows and 21 columns

Part 2: First Look at the Data (20 minutes)

Step 2.1: View Sample Data

python

```python
# Display first 5 rows
print("First 5 rows of the dataset:")
df.head()
```

Understanding `df.head()`:

- **What it does:** Shows the first 5 rows of your DataFrame
- **Why use it:** Quickly see what your data looks like
- **Variations:**
  - `df.head(10)` - Show first 10 rows
  - `df.tail()` - Show last 5 rows
  - `df.tail(20)` - Show last 20 rows
- **Returns:** A DataFrame with n rows

python

```python
# Display last 5 rows
print("Last 5 rows of the dataset:")
df.tail()
```

Why look at both head and tail?

- Check if data is consistent throughout
- See if there are any patterns in how data is organized
- Detect if data collection changed over time

Step 2.2: Understanding Data Structure

python

```python
# Get detailed information about the DataFrame
df.info()
```

Understanding `df.info()` Output:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 150000 entries, 0 to 149999
Data columns (total 21 columns):
 #   Column                    Non-Null Count   Dtype
---  ------                    --------------   -----
 0   Date                      150000 non-null  object
 1   Time                      150000 non-null  object
 2   Booking ID                150000 non-null  object
 ...
```

What this tells us:

- **RangeIndex:** Row numbers (0 to 149999)
- **150000 entries:** Total number of rows
- **21 columns:** Total number of columns
- **Non-Null Count:** How many non-missing values in each column
  - If less than 150000, that column has missing data
- **Dtype:** Data type of each column
  - `object` = Text/String
  - `int64` = Whole numbers
  - `float64` = Decimal numbers

    - `datetime64` = Date/Time

Key Things to Look For:

1. **Missing values:** Columns with Non-Null Count < 150000
2. **Data types:** Are they correct? (dates should be datetime, numbers should be numeric)
3. **Memory usage:** How much RAM the DataFrame uses

Step 2.3: Get Column Names

python

```python
# List all column names
print("Column names in the dataset:")
print(df.columns.tolist())
```

Understanding `df.columns`:

- **What it is:** An Index object containing all column names
- **`.tolist()`:** Converts it to a regular Python list
- **Why useful:**
    - Check exact spelling of columns (no typos!)
    - Copy-paste column names for filtering
    - Understand data structure

python

```python
# Better formatted view
print("\nColumns grouped by category:")
print("\n📅 Time Information:")
print("- Date, Time")
print("\n🚗 Booking Information:")
print("- Booking ID, Booking Status, Vehicle Type")
print("\n📍 Location Information:")
print("- Pickup Location, Drop Location")
print("\n💰 Financial Information:")
print("- Booking Value, Payment Method")
print("\n⭐ Rating Information:")
print("- Driver Ratings, Customer Rating")
print("\n❌ Cancellation Information:")
print("- Cancelled Rides by Customer, Reason for cancelling by Customer")
print("- Cancelled Rides by Driver, Driver Cancellation Reason")
print("- Incomplete Rides, Incomplete Rides Reason")
```

Step 2.4: Basic Statistics

python

```python
# Get statistical summary of numerical columns
df.describe()
```

Understanding `df.describe()` Output:

```
       Booking Value  Ride Distance  Driver Ratings  Customer Rating
count     150000.000     150000.000     145000.000      148000.000
mean         387.45         15.32          4.12            4.23
std         125.67          8.45          0.89            0.75
min          50.00          0.50          1.00            1.00
25%         295.00          9.00          3.50            4.00
50%         380.00         14.00          4.20            4.30
75%         475.00         20.00          4.80            4.70
max        1250.00         95.00          5.00            5.00
```

What Each Row Means:

- **count:** Number of non-null values (if less than total, some are missing)
- **mean:** Average value
- **std:** Standard deviation (how spread out the values are)
  - High std = values are very different from each other
  - Low std = values are similar
- **min:** Smallest value
- **25%:** First quartile - 25% of values are below this
- **50%:** Median (middle value) - 50% of values are below this
- **75%:** Third quartile - 75% of values are below this
- **max:** Largest value

What to Look For:

1. **Unrealistic values:**
   - Negative prices?
   - Ratings above 5?
   - Distance of 0 km?
2. **Outliers:**
   - Max value way larger than 75%?
   - Min value way smaller than 25%?
3. **Missing data:**
   - Count less than total rows?

python

```python
# Get statistics for specific columns
print("\nBooking Value Statistics:")
print(df['Booking Value'].describe())

print("\nDriver Ratings Statistics:")
print(df['Driver Ratings'].describe())
```

Step 2.5: Understanding Data Types

python

```python
# Check data types of all columns
print("Data types of each column:")
print(df.dtypes)
```

Common Data Types in Pandas:

Dtype	Meaning	Example
object	Text/String	"Delhi", "Cash", "SUV"
int64	Integer (whole number)	1, 42, 150000
float64	Decimal number	3.14, 387.45, 4.2
bool	True/False	True, False
datetime64	Date and time	2022-01-15 14:30:00

Why Data Types Matter:

- You can't do math on `object` type (even if it looks like a number)
- Dates stored as `object` can't be used for time-based analysis
- Wrong data types cause errors and prevent analysis

python

```python
# Check specific column type
print(f"Date column type: {df['Date'].dtype}")
print(f"Booking Value column type: {df['Booking Value'].dtype}")
```

Part 3: Initial Data Quality Check (15 minutes)

Step 3.1: Check for Missing Values

python

```python
# Count missing values in each column
print("Missing values per column:")
missing_values = df.isnull().sum()
print(missing_values)

# Show only columns with missing values
print("\nColumns with missing values:")
missing_cols = missing_values[missing_values > 0]
print(missing_cols)

# Calculate percentage of missing values
print("\nPercentage of missing values:")
missing_percent = (missing_values / len(df)) * 100
print(missing_percent[missing_percent > 0])
```

Understanding `df.isnull()`:

- **What it does:** Returns a DataFrame of True/False values
  - `True` where value is missing (NaN, None, null)
  - `False` where value exists
- **`.sum()`:** Counts the True values (missing data) per column
- **Why chain them:** `df.isnull().sum()` gives count of missing values per column

Example Output:

```
Date                          0
Time                          0
Booking ID                    0
Cancelled Rides by Customer      85000
Reason for cancelling by Customer 85000
Driver Ratings                5000
```

What This Tells Us:

- Some columns have NO missing values (Date, Time, Booking ID)
- Cancellation columns have many missing values (this is expected - not all rides are cancelled!)
- Driver Ratings has 5000 missing values (3.33% of data)

Step 3.2: Check for Duplicate Rows

python

```python
# Check for duplicate rows
duplicate_count = df.duplicated().sum()
print(f"Number of duplicate rows: {duplicate_count}")

# Show duplicate rows if any
if duplicate_count > 0:
    print("\nDuplicate rows:")
    duplicates = df[df.duplicated()]
    print(duplicates.head())
```

Understanding `df.duplicated()`:

- **What it does:** Returns True for each duplicate row (after the first occurrence)
- **`.sum()`:** Counts how many duplicates exist
- **Example:**

```
Row 1: A, B, C  <- Not marked as duplicate
Row 2: X, Y, Z  <- Not marked as duplicate
Row 3: A, B, C  <- Marked as duplicate (same as Row 1)
```

Parameters you can use:

python

```python
# Check duplicates based on specific columns
df.duplicated(subset=['Booking ID'])  # Only check if Booking ID is duplicate

# Keep different occurrences
df.duplicated(keep='first')  # Mark 2nd, 3rd, etc. as duplicates (default)
df.duplicated(keep='last')   # Mark 1st, 2nd, etc. as duplicates
df.duplicated(keep=False)    # Mark ALL duplicates (including first occurrence)
```

Step 3.3: Unique Values Check

python

```python
# Check unique values in categorical columns
categorical_columns = [
    'Booking Status',
    'Vehicle Type',
    'Payment Method'
]

print("Unique values in categorical columns:\n")
for col in categorical_columns:
    unique_count = df[col].nunique()
    unique_values = df[col].unique()
    print(f"{col}:")
    print(f"  - Number of unique values: {unique_count}")
    print(f"  - Values: {unique_values}")
    print()
```

Understanding `.nunique()` vs `.unique()`:

- **`.nunique()`:** Returns COUNT of unique values (just a number)
- **`.unique()`:** Returns ARRAY of actual unique values

Example Output:

```
Booking Status:
  - Number of unique values: 3
  - Values: ['Success' 'Cancelled by Customer' 'Cancelled by Driver']

Vehicle Type:
  - Number of unique values: 4
  - Values: ['Auto' 'Bike' 'Prime Sedan' 'Prime SUV']
```

Why This Matters:

- Helps identify typos or inconsistent naming ("sedan" vs "Sedan" vs "SEDAN")
- Reveals all possible categories for analysis
- Helps spot data quality issues

python

```python
# Value counts - how many times each unique value appears
print("Frequency of each Booking Status:")
print(df['Booking Status'].value_counts())

print("\nFrequency of each Vehicle Type:")
print(df['Vehicle Type'].value_counts())
```

Understanding `.value_counts()`:

- **What it does:** Counts how many times each unique value appears
- **Returns:** A Series sorted by frequency (most common first)
- **Useful for:** Understanding distribution of categorical data

Example Output:

```
Booking Status
Success                120000
Cancelled by Customer      20000
Cancelled by Driver        10000
Name: Booking Status, dtype: int64
```

Part 4: Basic Analysis Questions (25 minutes)

Now let's answer some simple questions about our data!

Question 1: How many total bookings do we have?

python

```python
# Method 1: Using shape
total_bookings = df.shape[0]
print(f"Total bookings: {total_bookings:,}")

# Method 2: Using len()
total_bookings = len(df)
print(f"Total bookings: {total_bookings:,}")

# Method 3: Counting non-null Booking IDs
total_bookings = df['Booking ID'].count()
print(f"Total bookings: {total_bookings:,}")
```

Which Method to Use?

- **`df.shape[0]`:** Fastest, always use this for row count
- **`len(df)`:** Also fast, more Pythonic
- **`.count()`:** Only use when you want to count non-null values

Question 2: What's the total revenue generated?

python

```python
# Sum all booking values
total_revenue = df['Booking Value'].sum()
print(f"Total revenue: ₹{total_revenue:,.2f}")

# Average booking value
avg_booking = df['Booking Value'].mean()
print(f"Average booking value: ₹{avg_booking:.2f}")

# Median booking value
median_booking = df['Booking Value'].median()
print(f"Median booking value: ₹{median_booking:.2f}")
```

Understanding `.sum()`, `.mean()`, `.median()`:

Function	What it calculates	When to use
.sum()	Total (adds all values)	Revenue, total distance, count
.mean()	Average (sum / count)	General "typical" value
.median()	Middle value (50th percentile)	When data has outliers

Mean vs Median Example:

```
Values: [10, 20, 30, 40, 1000]
Mean: 220 (skewed by the 1000)
Median: 30 (the middle value - more representative)
```

Formatting Numbers:

- **`:,`:** Adds thousand separators (58116750 → 58,116,750)
- **`.2f`:** Shows 2 decimal places (387.456789 → 387.46)
- **`:,.2f`:** Both! (58116750.456 → 58,116,750.46)

Question 3: How many unique customers do we have?

python

```python
# Count unique customers
unique_customers = df['Customer ID'].nunique()
print(f"Number of unique customers: {unique_customers:,}")

# Average bookings per customer
avg_bookings_per_customer = len(df) / unique_customers
print(f"Average bookings per customer: {avg_bookings_per_customer:.2f}")
```

Business Insight:

- If avg bookings per customer is high (>5), customers are loyal/frequent users
- If low (<2), mostly one-time users - need retention strategies

---

Question 4: What are the most popular vehicle types?

python

```python
# Count bookings by vehicle type
vehicle_counts = df['Vehicle Type'].value_counts()
print("Bookings by Vehicle Type:")
print(vehicle_counts)

# As percentages
vehicle_percentages = df['Vehicle Type'].value_counts(normalize=True) * 100
print("\nPercentage distribution:")
print(vehicle_percentages)
```

Understanding `.value_counts()` Parameters:

- **`normalize=True`:** Returns proportions (0.0 to 1.0) instead of counts
- **`dropna=False`:** Include missing values in count
- **`sort=False`:** Don't sort by frequency

Example Output:

```
Vehicle Type
Prime Sedan    60000  (40%)
Auto           45000  (30%)
Prime SUV      30000  (20%)
Bike           15000  (10%)
```

Question 5: What's the completion rate?

python

```python
# Count each booking status
status_counts = df['Booking Status'].value_counts()
print("Booking Status Distribution:")
print(status_counts)

# Calculate completion rate
total_rides = len(df)
completed_rides = status_counts['Success']  # or however success is labeled
completion_rate = (completed_rides / total_rides) * 100

print(f"\nCompletion Rate: {completion_rate:.2f}%")
print(f"Cancellation Rate: {100 - completion_rate:.2f}%")
```

Business Metrics:

- **Completion Rate:** % of rides that finished successfully
- **Cancellation Rate:** % of rides cancelled (customer + driver)
- **Industry Standard:** Usually aim for >85% completion rate

---

Part 5: Simple Visualizations (15 minutes)

Visualization 1: Distribution of Booking Values

python

```python
# Create histogram
plt.figure(figsize=(10, 6))  # Set size: width=10 inches, height=6 inches
plt.hist(df['Booking Value'], bins=50, edgecolor='black', color='skyblue')
plt.title('Distribution of Booking Values', fontsize=16, fontweight='bold')
plt.xlabel('Booking Value (₹)', fontsize=12)
plt.ylabel('Frequency (Number of Rides)', fontsize=12)
plt.grid(True, alpha=0.3)  # Add grid with 30% opacity
plt.tight_layout()  # Adjust spacing
plt.show()
```

Understanding `plt.hist()` Parameters:

- **`bins=50`:** Number of bars in histogram (more bins = more detail)
- **`edgecolor='black'`:** Border color around each bar
- **`color='skyblue'`:** Fill color of bars
- **`alpha=0.7`:** Transparency (0=invisible, 1=solid)

Understanding `plt.figure()` Parameters:

- **`figsize=(10, 6)`:** Size in inches (width, height)
- Standard sizes:
  - Small: `(8, 6)`
  - Medium: `(10, 6)`
  - Large: `(12, 8)`
  - Wide: `(15, 6)`

What This Chart Shows:

- Most common booking values
- Whether prices are clustered or spread out
- If there are any unusual price points

Visualization 2: Vehicle Type Distribution

python

```python
# Create bar chart
vehicle_counts = df['Vehicle Type'].value_counts()

plt.figure(figsize=(10, 6))
vehicle_counts.plot(kind='bar', color='coral', edgecolor='black')
plt.title('Number of Rides by Vehicle Type', fontsize=16, fontweight='bold')
plt.xlabel('Vehicle Type', fontsize=12)
plt.ylabel('Number of Rides', fontsize=12)
plt.xticks(rotation=45, ha='right')  # Rotate x-axis labels 45 degrees
plt.grid(True, alpha=0.3, axis='y')  # Only horizontal grid lines
plt.tight_layout()
plt.show()
```

Understanding `plot()` Parameters:

- **`kind='bar'`:** Type of plot
  - `'bar'` - Vertical bars
  - `'barh'` - Horizontal bars
  - `'line'` - Line chart
  - `'hist'` - Histogram
  - `'scatter'` - Scatter plot
  - `'pie'` - Pie chart

Understanding `plt.xticks()` Parameters:

- **`rotation=45`:** Rotate labels 45 degrees (prevents overlap)
- **`ha='right'`:** Horizontal alignment (left, center, right)
- Common rotation values:

- 0 = Horizontal
- 45 = Diagonal (most common)
- 90 = Vertical

Visualization 3: Booking Status Pie Chart

python

```python
# Create pie chart
status_counts = df['Booking Status'].value_counts()

plt.figure(figsize=(8, 8))
plt.pie(
    status_counts,
    labels=status_counts.index,
    autopct='%1.1f%%',  # Show percentages with 1 decimal
    startangle=90,      # Start from top (90 degrees)
    colors=['#2ecc71', '#e74c3c', '#f39c12'],  # Custom colors
    explode=(0.05, 0, 0)  # "Explode" first slice slightly
)
plt.title('Distribution of Booking Status', fontsize=16, fontweight='bold')
plt.axis('equal')  # Equal aspect ratio = circular pie
plt.tight_layout()
plt.show()
```

Understanding `plt.pie()` Parameters:

- **`autopct='%1.1f%%'`:** Format for percentages
  - `%1.1f` = 1 decimal place (25.5%)
  - `%1.0f` = No decimals (25%)
  - `%1.2f` = 2 decimals (25.50%)
- **`startangle=90`:** Where to start drawing (0-360 degrees)
- **`explode=(0.05, 0, 0)`:** Pull slices out
  - One value per slice
  - 0 = Not pulled out
  - 0.1 = Pulled out 10%
- **`colors`:** List of colors for each slice
  - Can use names: 'red', 'blue'
  - Or hex codes: '`#2ecc71`', '`#e74c3c`'

Part 6: Wrap-up & Homework (5 minutes)

Key Takeaways from Session 1:

✅ What We Learned:

1. How to load data with `pd.read_csv()`
2. How to explore data with `.head()`, `.tail()`, `.info()`, `.describe()`
3. How to check data quality (missing values, duplicates)
4. How to get basic statistics (`.sum()`, `.mean()`, `.value_counts()`)
5. How to create simple visualizations

✅ Key Functions Mastered:

- `df.head()` / `df.tail()` - View sample data
- `df.info()` - Get column information
- `df.describe()` - Get statistical summary
- `df.isnull().sum()` - Check missing values
- `df['column'].value_counts()` - Count unique values
- `df['column'].sum()` / `.mean()` / `.median()` - Basic math
- `plt.hist()` / `.plot()` / `.pie()` - Basic visualizations

Homework for Next Session:

python

```python
# 1. Find the most expensive ride
most_expensive = df['Booking Value'].max()
print(f"Most expensive ride: ₹{most_expensive}")

# 2. Find the longest distance traveled
longest_distance = df['Ride Distance'].max()
print(f"Longest distance: {longest_distance} km")

# 3. What's the most common payment method?
most_common_payment = df['Payment Method'].value_counts().index[0]
print(f"Most common payment method: {most_common_payment}")

# 4. How many rides were longer than 20 km?
long_rides = df[df['Ride Distance'] > 20].shape[0]
print(f"Rides longer than 20 km: {long_rides}")

# 5. What's the average rating given to drivers?
avg_driver_rating = df['Driver Ratings'].mean()
print(f"Average driver rating: {avg_driver_rating:.2f}")
```

Challenge Questions:

1. Which pickup location had the most bookings?
2. What percentage of rides used each payment method?
3. Create a bar chart showing top 10 pickup locations by number of rides

---

Session 2: Data Cleaning & Preparation

Duration: 90 minutes
Goal: Clean the data and prepare it for analysis

Part 1: Understanding Data Quality Issues (10 minutes)

What Makes Data "Dirty"?

python

```python
# Let's investigate our data quality systematically
print("=" * 50)
print("DATA QUALITY REPORT")
print("=" * 50)

# 1. Missing Values Summary
print("\n1. MISSING VALUES:")
missing_summary = df.isnull().sum()
print(missing_summary[missing_summary > 0])

# 2. Data Type Issues
print("\n2. DATA TYPES:")
print(df.dtypes)

# 3. Duplicate Check
print(f"\n3. DUPLICATES: {df.duplicated().sum()} duplicate rows found")

# 4. Inconsistent Values
print("\n4. UNIQUE VALUES IN KEY COLUMNS:")
for col in ['Booking Status', 'Vehicle Type', 'Payment Method']:
    print(f"{col}: {df[col].unique()}")
```

Common Data Quality Issues:

1. **Missing Values** - Blank cells, NaN, None
2. **Wrong Data Types** - Dates stored as text, numbers as strings
3. **Duplicates** - Same row appears multiple times
4. **Inconsistent Formatting** - "sedan" vs "Sedan", extra spaces
5. **Outliers** - Unrealistic values (negative prices, 1000 km rides in city)
6. **Invalid Values** - Ratings above 5, negative distances

Part 2: Handling Missing Data (25 minutes)

Step 2.1: Analyze Missing Data Patterns

python

```python
# Create a visual summary of missing data
import matplotlib.pyplot as plt

# Calculate missing percentages
missing_percent = (df.isnull().sum() / len(df)) * 100
missing_data = missing_percent[missing_percent > 0].sort_values(ascending=False)

# Plot missing data
plt.figure(figsize=(12, 6))
missing_data.plot(kind='bar', color='salmon', edgecolor='black')
plt.title('Percentage of Missing Values by Column', fontsize=16, fontweight='bold')
plt.xlabel('Column Name', fontsize=12)
plt.ylabel('Percentage Missing (%)', fontsize=12)
plt.xticks(rotation=45, ha='right')
plt.grid(True, alpha=0.3, axis='y')
plt.tight_layout()
plt.show()

# Print detailed summary
for col in missing_data.index:
    count = df[col].isnull().sum()
    percent = missing_percent[col]
    print(f"{col}: {count:,} missing ({percent:.2f}%)")
```

Step 2.2: Understanding WHY Data is Missing

Three Types of Missing Data:

1. **Missing Completely at Random (MCAR)**
   - No pattern to missingness
   - Example: Device randomly failed to record data
   - Strategy: Usually safe to drop or impute
2. **Missing at Random (MAR)**
   - Missingness depends on other variables
   - Example: High-value rides less likely to report distance
   - Strategy: Impute based on related variables
3. **Missing Not at Random (MNAR)**
   - Missingness is related to the missing value itself
   - Example: Unhappy customers don't give ratings
   - Strategy: Careful! Might need special handling

python

```python
# Let's investigate cancellation columns
print("Understanding Cancellation Data:")
print(f"Total rides: {len(df)}")
print(f"Missing 'Cancelled by Customer': {df['Cancelled Rides by Customer'].isnull().sum()}")
print(f"Missing 'Reason for cancelling': {df['Reason for cancelling by Customer'].isnull().sum()}")

# These nulls are INFORMATIVE - they mean "no cancellation happened"
# This is MNAR - the data is missing BECAUSE there was no cancellation
```

Key Insight About Cancellation Data:

- Null values in cancellation columns mean "didn't cancel"
- These are NOT errors - they're informative!
- We should fill them with 0 or "Not Cancelled", not delete them

---

Step 2.3: Strategy for Handling Missing Data

python

```python
# Create a copy to work with (always preserve original!)
df_clean = df.copy()

print("Original shape:", df.shape)
```

Why Create a Copy?

- `df.copy()` creates a completely separate DataFrame
- Changes to `df_clean` won't affect original `df`
- Lets you compare before/after
- Easy to restart if you make mistakes

Without .copy():

python

```python
df_clean = df  # This is just a reference, not a copy!
df_clean['column'] = 0  # This ALSO changes df!
```

Step 2.4: Handling Cancellation Columns

python

```python
# Strategy: Fill cancellation nulls with 0 (meaning "didn't happen")
cancellation_columns = [
    'Cancelled Rides by Customer',
    'Cancelled Rides by Driver',
    'Incomplete Rides'
]

for col in cancellation_columns:
    # Before
    before_nulls = df_clean[col].isnull().sum()

    # Fill nulls with 0
    df_clean[col] = df_clean[col].fillna(0)

    # After
    after_nulls = df_clean[col].isnull().sum()

    print(f"{col}:")
    print(f"  Before: {before_nulls:,} nulls")
    print(f"  After: {after_nulls:,} nulls")
    print(f"  ✅ Filled {before_nulls:,} missing values with 0\n")
```

Understanding `.fillna()`:

- **What it does:** Replaces NaN/null values with specified value
- **Syntax:** `df['column'].fillna(value)`
- **Common fill values:**
  - `0` - For counts, flags
  - `'Unknown'` - For categorical text
  - `.mean()` - Average of column
  - `.median()` - Middle value of column
  - `.mode()[0]` - Most common value
- **Returns:** New Series/DataFrame (doesn't change original unless `inplace=True`)

python

```
# Handle reason columns - fill with 'Not Applicable'
reason_columns = [
    'Reason for cancelling by Customer',
    'Driver Cancellation Reason',
    'Incomplete Rides Reason'
]

for col in reason_columns:
    before_nulls = df_clean[col].isnull().sum()
    df_clean[col] = df_clean[col].fillna('Not Applicable')
    after_nulls = df_clean[col].isnull().sum()

    print(f"{col}:")
    print(f"  Before: {before_nulls:,} nulls")
    print(f"  After: {after_nulls:,} nulls")
    print(f"  ✅ Filled with 'Not Applicable'\n")
```

Step 2.5: Handling Rating Columns

python

```python
# Check rating distributions before imputation
print("Driver Ratings Distribution:")
print(df_clean['Driver Ratings'].describe())

print("\nCustomer Rating Distribution:")
print(df_clean['Customer Rating'].describe())

# Strategy: Fill with MEDIAN (more robust than mean for ratings)
print("\nFilling missing ratings with median...")

# Driver Ratings
driver_median = df_clean['Driver Ratings'].median()
print(f"Driver Ratings median: {driver_median}")
df_clean['Driver Ratings'] = df_clean['Driver Ratings'].fillna(driver_median)

# Customer Rating
customer_median = df_clean['Customer Rating'].median()
print(f"Customer Rating median: {customer_median}")
df_clean['Customer Rating'] = df_clean['Customer Rating'].fillna(customer_median)

# Verify
print(f"\n✅ Driver Ratings nulls remaining: {df_clean['Driver Ratings'].isnull().sum()}")
print(f"✅ Customer Rating nulls remaining: {df_clean['Customer Rating'].isnull().sum()}")
```

Mean vs Median for Imputation:

Use MEAN when:

- Data is normally distributed (bell curve)
- No extreme outliers
- Example: Heights, test scores

Use MEDIAN when:

- Data has outliers
- Skewed distribution
- Ratings (1-5 scale with clustering)

Example:

```
Ratings: [4.5, 4.7, 4.8, 4.2, 4.6, 1.0, 4.3]
Mean: 4.0 (pulled down by the 1.0)
Median: 4.5 (more representative)
```

Step 2.6: Final Missing Value Check

python

```python
# Verify all missing values are handled
print("=" * 50)
print("FINAL MISSING VALUES CHECK")
print("=" * 50)

remaining_nulls = df_clean.isnull().sum()
print(remaining_nulls[remaining_nulls > 0])

if remaining_nulls.sum() == 0:
    print("\n✅ SUCCESS! All missing values handled!")
else:
    print(f"\n⚠️ Still have {remaining_nulls.sum()} missing values to address")
```

Part 3: Data Type Conversions (20 minutes)

Step 3.1: Converting Date Column

python

```python
# Check current date type
print(f"Current Date type: {df_clean['Date'].dtype}")
print(f"Sample values:\n{df_clean['Date'].head()}")

# Convert to datetime
df_clean['Date'] = pd.to_datetime(df_clean['Date'])

# Verify conversion
print(f"\nNew Date type: {df_clean['Date'].dtype}")
print(f"Sample values:\n{df_clean['Date'].head()}")
```

Understanding `pd.to_datetime()`:

- **What it does:** Converts strings to datetime objects
- **Why important:** Enables time-based operations
  - Extract day, month, year
  - Filter by date ranges
  - Calculate time differences
  - Sort chronologically

Common Parameters:

python

```python
# Specify format for faster parsing
pd.to_datetime(df['Date'], format='%Y-%m-%d')

# Handle errors
pd.to_datetime(df['Date'], errors='coerce')  # Invalid dates become NaT
pd.to_datetime(df['Date'], errors='ignore')  # Keep as string if fails

# Handle different date formats
pd.to_datetime(df['Date'], dayfirst=True)  # DD/MM/YYYY
pd.to_datetime(df['Date'], yearfirst=True)  # YYYY/MM/DD
```

Date Format Codes:

- `%Y` - 4-digit year (2022)
- `%y` - 2-digit year (22)
- `%m` - Month as number (01-12)
- `%d` - Day as number (01-31)
- `%H` - Hour (00-23)
- `%M` - Minute (00-59)
- `%S` - Second (00-59)

Step 3.2: Converting Time Column

python

```python
# Check current time type
print(f"Current Time type: {df_clean['Time'].dtype}")
print(f"Sample values:\n{df_clean['Time'].head()}")

# Convert to datetime (even though it's just time)
df_clean['Time'] = pd.to_datetime(df_clean['Time'], format='%H:%M:%S').dt.time

# Verify
print(f"\nNew Time type: {df_clean['Time'].dtype}")
print(f"Sample values:\n{df_clean['Time'].head()}")
```

Understanding `.dt` accessor:

- **What it is:** Special accessor for datetime operations
- **Only works on:** datetime64 columns
- **Allows you to extract:** date components, format dates, etc.

Common `.dt` operations:

python

```python
df['Date'].dt.year        # Extract year
df['Date'].dt.month       # Extract month (1-12)
df['Date'].dt.day         # Extract day (1-31)
df['Date'].dt.dayofweek   # Day of week (0=Monday, 6=Sunday)
df['Date'].dt.day_name()  # Day name ('Monday', 'Tuesday', etc.)
df['Date'].dt.month_name()# Month name ('January', 'February', etc.)
df['Date'].dt.quarter     # Quarter (1-4)
df['Date'].dt.weekofyear  # Week number (1-52)
```

Step 3.3: Extracting Date Components

python

```
# Extract useful date features for analysis
print("Extracting date components...")

# Day of week
df_clean['Day_of_Week'] = df_clean['Date'].dt.day_name()
print("✅ Created Day_of_Week column")

# Month
df_clean['Month'] = df_clean['Date'].dt.month_name()
print("✅ Created Month column")

# Day of month
df_clean['Day'] = df_clean['Date'].dt.day
print("✅ Created Day column")

# Hour from time
# Need to convert time back to datetime for extraction
df_clean['Hour'] = pd.to_datetime(df_clean['Time'].astype(str), format='%H:%M:%S').dt.hour
print("✅ Created Hour column")

# Is weekend?
df_clean['Is_Weekend'] = df_clean['Date'].dt.dayofweek >= 5  # Saturday=5, Sunday=6
print("✅ Created Is_Weekend column")

# Show sample
print("\nSample of new columns:")
print(df_clean[['Date', 'Time', 'Day_of_Week', 'Month', 'Hour', 'Is_Weekend']].head())
```

Why Extract Date Components?

1. **Analysis by time periods:** "Which day has most bookings?"
2. **Seasonal patterns:** "Are Mondays busier than Fridays?"
3. **Time-based grouping:** "Revenue by month"
4. **Feature engineering:** Use in machine learning later

Understanding Boolean Columns:

python

```python
df_clean['Is_Weekend'] = df_clean['Date'].dt.dayofweek >= 5
```

- Returns True/False for each row
- `dayofweek`: 0=Monday, 1=Tuesday, ..., 5=Saturday, 6=Sunday
- `>= 5` means Saturday or Sunday
- Can use in filtering: `df_clean[df_clean['Is_Weekend']]`

Step 3.4: Creating Ride Outcome Column

python

```python
# Create a categorical column for ride outcome
# This makes analysis easier than checking multiple columns

print("Creating Ride Outcome column...")

# Initialize with 'Completed'
df_clean['Ride_Outcome'] = 'Completed'

# Update based on cancellation columns
df_clean.loc[df_clean['Cancelled Rides by Customer'] > 0, 'Ride_Outcome'] = 'Cancelled by Customer'
df_clean.loc[df_clean['Cancelled Rides by Driver'] > 0, 'Ride_Outcome'] = 'Cancelled by Driver'
df_clean.loc[df_clean['Incomplete Rides'] > 0, 'Ride_Outcome'] = 'Incomplete'

# Verify
print("\nRide Outcome Distribution:")
print(df_clean['Ride_Outcome'].value_counts())
print(f"\n✅ Created Ride_Outcome column")
```

Understanding `.loc[]`:

- **What it does:** Accesses rows and columns by labels
- **Syntax:** `df.loc[row_condition, column_name]`
- **Use cases:**
  - Filter and update: `df.loc[condition, 'column'] = value`
  - Select subset: `df.loc[rows, columns]`
  - Boolean indexing: `df.loc[df['Age'] > 25]`

Why Use .loc for Assignment?

python

```python
# ❌ BAD - Can cause SettingWithCopyWarning
df[df['value'] > 100]['category'] = 'High'

# ✅ GOOD - Clear and explicit
df.loc[df['value'] > 100, 'category'] = 'High'
```

Part 4: Handling Inconsistent Data (15 minutes)

Step 4.1: Standardizing Text Columns

python

```python
# Check for inconsistencies in categorical columns
print("Checking for inconsistencies...\n")

# Vehicle Type
print("Vehicle Type unique values:")
print(df_clean['Vehicle Type'].unique())
print(f"Count: {df_clean['Vehicle Type'].nunique()}")

# Check for common issues
print("\nChecking for:")
print("- Leading/trailing spaces")
print("- Inconsistent capitalization")
print("- Typos")

# Clean up spaces and standardize
df_clean['Vehicle Type'] = df_clean['Vehicle Type'].str.strip()  # Remove leading/trailing spaces
df_clean['Vehicle Type'] = df_clean['Vehicle Type'].str.title()  # Title Case

# Same for other categorical columns
text_columns = ['Pickup Location', 'Drop Location', 'Payment Method', 'Booking Status']

for col in text_columns:
    df_clean[col] = df_clean[col].str.strip()
    print(f"✅ Cleaned {col}")
```

Understanding `.str` accessor:

- **What it is:** Special accessor for string operations
- **Only works on:** Object (string) columns
- **Like:** Python string methods but for entire columns

Common `.str` operations:

python

```python
df['column'].str.lower()      # Convert to lowercase
df['column'].str.upper()      # Convert to UPPERCASE
df['column'].str.title()      # Convert To Title Case
df['column'].str.strip()      # Remove leading/trailing spaces
df['column'].str.replace('old', 'new')  # Replace text
df['column'].str.contains('text')  # Check if contains text
df['column'].str.len()        # Length of each string
df['column'].str.split(',')   # Split by delimiter
```

Step 4.2: Fixing Data Entry Errors

python

```python
# Example: Standardize location names
print("\nStandardizing location names...")

# Create mapping for common variations
location_mapping = {
    'Gurgaon': 'Gurugram',  # Official name
    'Delhi Airport': 'Indira Gandhi International Airport',
    'CP': 'Connaught Place',
    'Noida Sec': 'Noida Sector'
}

# Apply mapping
for old_name, new_name in location_mapping.items():
    df_clean['Pickup Location'] = df_clean['Pickup Location'].str.replace(old_name, new_name)
    df_clean['Drop Location'] = df_clean['Drop Location'].str.replace(old_name, new_name)

print("✅ Location names standardized")

# Verify
print("\nTop Pickup Locations after cleaning:")
print(df_clean['Pickup Location'].value_counts().head(10))
```

Understanding `.replace()`:

python

```python
# Single value replacement
df['column'].replace('old', 'new')

# Multiple replacements with dictionary
df['column'].replace({'old1': 'new1', 'old2': 'new2'})

# Regex patterns
df['column'].str.replace(r'\d+', '', regex=True)  # Remove all numbers
```

Part 5: Handling Outliers (15 minutes)

Step 5.1: Detecting Outliers

python

```python
# Statistical method: Values beyond 3 standard deviations
print("Detecting outliers in numerical columns...\n")

numerical_cols = ['Booking Value', 'Ride Distance']

for col in numerical_cols:
    # Calculate statistics
    mean = df_clean[col].mean()
    std = df_clean[col].std()

    # Define outlier thresholds
    lower_bound = mean - 3 * std
    upper_bound = mean + 3 * std

    # Find outliers
    outliers = df_clean[(df_clean[col] < lower_bound) | (df_clean[col] > upper_bound)]

    print(f"{col}:")
    print(f"  Mean: {mean:.2f}")
    print(f"  Std Dev: {std:.2f}")
    print(f"  Lower Bound: {lower_bound:.2f}")
    print(f"  Upper Bound: {upper_bound:.2f}")
    print(f"  Number of outliers: {len(outliers):,} ({len(outliers)/len(df_clean)*100:.2f}%)")
    print(f"  Min outlier: {outliers[col].min():.2f}")
    print(f"  Max outlier: {outliers[col].max():.2f}")
    print()
```

Understanding Outlier Detection Methods:

1. Standard Deviation Method (3-sigma rule):

python

```
lower = mean - 3*std
upper = mean + 3*std
outliers = data outside [lower, upper]
```

- Works well for normally distributed data
- 99.7% of data should be within 3 standard deviations

2. IQR Method (Interquartile Range):

python

```python
Q1 = data.quantile(0.25)
Q3 = data.quantile(0.75)
IQR = Q3 - Q1
lower = Q1 - 1.5*IQR
upper = Q3 + 1.5*IQR
```

- More robust to extreme outliers
- Standard for box plots

3. Domain Knowledge:

python

```python
# Business rules
outliers = df[df['Booking Value'] < 0]  # Negative prices impossible
outliers = df[df['Rating'] > 5]  # Ratings above 5 impossible
```

Step 5.2: Visualizing Outliers

python

```python
# Box plot to visualize outliers
fig, axes = plt.subplots(1, 2, figsize=(15, 6))

# Booking Value
axes[0].boxplot(df_clean['Booking Value'].dropna())
axes[0].set_title('Booking Value Distribution', fontsize=14, fontweight='bold')
axes[0].set_ylabel('Booking Value (₹)')
axes[0].grid(True, alpha=0.3)

# Ride Distance
axes[1].boxplot(df_clean['Ride Distance'].dropna())
axes[1].set_title('Ride Distance Distribution', fontsize=14, fontweight='bold')
axes[1].set_ylabel('Distance (km)')
axes[1].grid(True, alpha=0.3)

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

Understanding Box Plots:

```
    Maximum (upper whisker)
        |
    ----+----  Q3 (75th percentile)
    |       |
    |  BOX  |  Median (50th percentile)
    |       |
    ----+----  Q1 (25th percentile)
        |
    Minimum (lower whisker)

    ○ ○      Outliers (dots beyond whiskers)
```

- **Box:** Contains middle 50% of data (Q1 to Q3)
- **Line in box:** Median
- **Whiskers:** Extend to 1.5*IQR beyond box
- **Dots:** Outliers beyond whiskers

Step 5.3: Handling Outliers

python

```python
# Decision: Keep outliers but flag them for investigation
print("Flagging outliers for investigation...")

# Create outlier flags
df_clean['Is_High_Value'] = df_clean['Booking Value'] > df_clean['Booking Value'].quantile(0.95)
df_clean['Is_Long_Distance'] = df_clean['Ride Distance'] > df_clean['Ride Distance'].quantile(0.95)

# Check flagged rides
print(f"\nHigh value rides (top 5%): {df_clean['Is_High_Value'].sum():,}")
print(f"Long distance rides (top 5%): {df_clean['Is_Long_Distance'].sum():,}")

# Investigate one outlier
print("\nSample high-value ride:")
sample_outlier = df_clean[df_clean['Is_High_Value']].iloc[0]
print(f"Booking Value: ₹{sample_outlier['Booking Value']}")
print(f"Distance: {sample_outlier['Ride Distance']} km")
print(f"Vehicle Type: {sample_outlier['Vehicle Type']}")
print(f"Pickup: {sample_outlier['Pickup Location']}")
print(f"Drop: {sample_outlier['Drop Location']}")
```

Strategies for Handling Outliers:

1. Keep them (recommended if legitimate):

python

```python
# Just flag for analysis
df['is_outlier'] = (df['value'] > threshold)
```

2. Remove them (careful! losing data):

python

```python
# Only if clearly errors
df_clean = df[(df['value'] >= lower) & (df['value'] <= upper)]
```

3. Cap them (winsorization):

python

```python
# Replace extreme values with threshold
df['value'] = df['value'].clip(lower=lower, upper=upper)
```

4. Transform them (log, square root):

python

```python
# Reduce impact of extremes
df['value_log'] = np.log(df['value'] + 1)
```

Part 6: Data Validation & Export (5 minutes)

Step 6.1: Final Data Quality Check

python

```python
print("=" * 60)
print("FINAL DATA QUALITY REPORT")
print("=" * 60)

# 1. Shape
print(f"\n1. Dataset Shape:")
print(f"   Original: {df.shape}")
print(f"   Cleaned: {df_clean.shape}")

# 2. Missing Values
print(f"\n2. Missing Values:")
total_nulls = df_clean.isnull().sum().sum()
if total_nulls == 0:
    print("   ✅ No missing values!")
else:
    print(f"   ⚠️ {total_nulls} missing values remain")

# 3. Data Types
print(f"\n3. Data Types:")
print(f"   Datetime columns: {df_clean.select_dtypes(include='datetime64').columns.tolist()}")
print(f"   Numeric columns: {df_clean.select_dtypes(include=['int64', 'float64']).columns.tolist()}")
print(f"   Text columns: {df_clean.select_dtypes(include='object').columns.tolist()}")

# 4. New Columns Created
new_columns = [col for col in df_clean.columns if col not in df.columns]
print(f"\n4. New Columns Created ({len(new_columns)}):")
for col in new_columns:
    print(f"   - {col}")

# 5. Duplicates
print(f"\n5. Duplicate Rows:")
print(f"   {df_clean.duplicated().sum()} duplicates")

print("\n" + "=" * 60)
print("✅ DATA CLEANING COMPLETE!")
print("=" * 60)
```

Step 6.2: Save Cleaned Data

python

```python
# Save cleaned dataset
output_file = 'data/ncr_ride_bookings_cleaned.csv'
df_clean.to_csv(output_file, index=False)

print(f"✅ Cleaned data saved to: {output_file}")
print(f"   Rows: {df_clean.shape[0]:,}")
print(f"   Columns: {df_clean.shape[1]}")
print(f"   File size: {os.path.getsize(output_file) / (1024*1024):.2f} MB")
```

Understanding `to_csv()` Parameters:

- **`index=False`:** Don't save row numbers as a column
- **`sep=','`:** Use comma as delimiter (default)
- **`encoding='utf-8'`:** Character encoding
- **`na_rep='NA'`:** How to represent missing values
- **`columns=['col1', 'col2']`:** Save only specific columns
- **`header=True`:** Include column names (default)

Part 7: Session 2 Wrap-up

Key Takeaways:

✅ Data Cleaning Skills Mastered:

1. Identifying and understanding missing data patterns
2. Filling missing values appropriately (`.fillna()`)
3. Converting data types (`.astype()`, `pd.to_datetime()`)
4. Extracting date components (`.dt` accessor)
5. Standardizing text data (`.str` methods)
6. Detecting and handling outliers
7. Creating new useful columns

✅ Key Functions:

- `df.fillna()` - Fill missing values
- `pd.to_datetime()` - Convert to datetime
- `.dt.day_name()`, `.dt.month_name()`, `.dt.hour` - Extract date parts
- `.str.strip()`, `.str.title()`, `.str.replace()` - Clean text
- `.loc[]` - Conditional selection and assignment
- `.quantile()` - Find percentiles
- `df.to_csv()` - Save cleaned data

Homework:

python

```python
# 1. Check average booking value by vehicle type
avg_by_vehicle = df_clean.groupby('Vehicle Type')['Booking Value'].mean()
print(avg_by_vehicle)

# 2. Find completion rate by day of week
completion_by_day = df_clean.groupby('Day_of_Week')['Ride_Outcome'].value_counts(normalize=True)
print(completion_by_day)

# 3. Compare weekend vs weekday revenue
weekend_revenue = df_clean.groupby('Is_Weekend')['Booking Value'].sum()
print(weekend_revenue)

# 4. Identify peak booking hours
peak_hours = df_clean['Hour'].value_counts().sort_index()
print(peak_hours)
```

Session 3: Exploratory Data Analysis (EDA)

Duration: 90 minutes
Goal: Discover patterns and insights in the data

Part 1: Business Questions Framework (10 minutes)

The Questions We'll Answer:

1. Revenue Analysis:

- Which locations generate most revenue?
- What's the average booking value by vehicle type?
- How does revenue vary by time of day?

2. Cancellation Analysis:

- What's the cancellation rate?
- Why do customers cancel?
- Why do drivers cancel?
- Which locations have highest cancellations?

3. Time-Based Patterns:

- Which days are busiest?
- What are peak booking hours?
- Weekend vs weekday patterns?

4. Customer Satisfaction:

- Average ratings by vehicle type?
- Correlation between distance and ratings?
- Do longer rides get better ratings?

5. Geographic Insights:

- Most popular pickup locations?
- Most common routes?
- Distance patterns by location?

Part 2: Revenue Analysis (20 minutes)

Question 1: Top Revenue-Generating Locations

python

```python
# Group by location and calculate total revenue
revenue_by_location = df_clean.groupby('Pickup Location')['Booking Value'].agg([
    ('Total_Revenue', 'sum'),
    ('Avg_Booking', 'mean'),
    ('Num_Rides', 'count')
]).sort_values('Total_Revenue', ascending=False)

# Show top 10
print("Top 10 Revenue-Generating Locations:")
print(revenue_by_location.head(10))

# Visualize
plt.figure(figsize=(12, 6))
top_10_locations = revenue_by_location.head(10)
plt.barh(range(len(top_10_locations)), top_10_locations['Total_Revenue'], color='green')
plt.yticks(range(len(top_10_locations)), top_10_locations.index)
plt.xlabel('Total Revenue (₹)', fontsize=12)
plt.title('Top 10 Locations by Revenue', fontsize=16, fontweight='bold')
plt.gca().invert_yaxis()  # Highest at top
plt.grid(True, alpha=0.3, axis='x')
plt.tight_layout()
plt.show()
```

Understanding `.agg()` with Custom Names:

python

```python
.agg([
    ('Custom_Name', 'function'),
    ('Another_Name', 'mean')
])
```

- Creates columns with specified names
- More readable than default names
- Can apply multiple functions to same column

Understanding `.gca().invert_yaxis()`:

- **`.gca()`:** "Get Current Axes" - gets the current plot
- **`.invert_yaxis()`:** Flips y-axis (highest value at top)
- Useful for rankings - #1 should be at top!
