Mastering Data Analysis with Python: A Beginner's Guide to Data Collection and Visualization

This comprehensive guide will walk you through the fundamental concepts of data analysis using Python, focusing on data collection, manipulation, and visualization. We'll be using popular Python libraries like Pandas, NumPy, Matplotlib, and Seaborn to illustrate these concepts with practical examples using the provided Uber ride data.

Introduction

In the realm of data-driven decision making, the ability to extract meaningful insights from raw data is paramount. This process typically involves a series of steps known as the ETL (Extract, Transform, Load) pipeline, followed by data analysis and visualization.

The ETL Pipeline: A Foundation for Data Analysis

1. **Data Extraction (Collection):** The first step involves gathering raw data from various sources. These sources can range from simple files like CSVs, TSVs, and JSONs to complex databases and big data frameworks like Hive.
2. **Data Transformation (Cleaning and Preparation):** Raw data is often messy and requires cleaning and transformation before analysis. This step involves handling missing values, converting data types, and structuring the data into a usable format.
3. **Data Loading (Featurization):** The final step involves loading the transformed data into a suitable format for analysis. This often involves creating features (variables) that will be used for modeling and visualization.

Essential Python Libraries for Data Analysis

1. **Pandas:** This powerful library excels in data manipulation and analysis. It provides data structures like Series (one-dimensional) and DataFrames (two-dimensional), which are analogous to lists and tables, respectively. Pandas simplifies data cleaning, transformation, and exploration.

2. **NumPy:** This library forms the foundation for numerical computing in Python. It provides tools for working with arrays, performing mathematical operations, and generating random numbers. NumPy is essential for performing calculations on your data.
3. **Matplotlib:** This library serves as the base for data visualization in Python. It allows you to create static, interactive, and animated plots, providing a wide range of customization options. Matplotlib is crucial for visualizing patterns and trends in your data.
4. **Seaborn:** Built on top of Matplotlib, Seaborn offers a higher-level interface for creating statistically informative and visually appealing plots. It simplifies the process of generating common statistical visualizations, making it easier to gain insights from your data.
5. **OS:** This library provides functions for interacting with the operating system. It allows you to perform tasks like creating, deleting, and modifying files and directories. The OS library is helpful for managing your data files and accessing data from different locations.

Practical Example: Analyzing Uber Ride Data

Let's apply these concepts to your provided dataset containing information about Uber rides. We'll use Python code snippets to illustrate each step:

1. Importing Libraries:

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

2. Loading Data with Pandas:

```
# Define the file path
file_path = r"C:\Users\USER\Desktop\stats class\EDA Project 1\Datasets\uber-raw-data-janjune-15_sample.csv"

# Load the CSV file into a Pandas DataFrame
uber_data = pd.read_csv(file_path)

# Check the dimensions of the DataFrame
print(uber_data.shape)
```

This code snippet imports necessary libraries, defines the file path you provided, loads the CSV file into a DataFrame, and displays its dimensions (rows, columns).

3. Exploring Data with Pandas:

```
# View the first few rows of the DataFrame
print(uber_data.head())

# Get summary statistics of numerical columns
print(uber_data.describe())
```

These commands allow you to preview the data and obtain descriptive statistics, providing an initial understanding of its structure and content.

4. Data Visualization with Matplotlib and Seaborn:

```
# Create a histogram of ride fares
plt.hist(uber_data['Fare'], bins=20)
plt.xlabel('Fare')
plt.ylabel('Frequency')
plt.title('Distribution of Uber Ride Fares')
plt.show()

# Create a scatter plot of ride distance vs. fare (assuming 'Distance' column exists)
plt.scatter(uber_data['Distance'], uber_data['Fare'])
plt.xlabel('Distance')
plt.ylabel('Fare')
plt.title('Ride Distance vs. Fare')
plt.show()
```

These code snippets demonstrate how to create a histogram to visualize the distribution of ride fares and a scatter plot to explore the relationship between ride distance and fare. You might need to adjust column names based on your specific dataset.

Data Pre-processing and Cleaning with Pandas in Python

This section delves into the crucial step of data pre-processing and cleaning using the Pandas library in Python. Building upon the previous session on data extraction, we'll focus on transforming raw data into a usable format for analysis.

Data Cleaning: A Critical Step

Data cleaning involves identifying and rectifying errors, inconsistencies, and missing values within a dataset. This process ensures data quality and reliability for subsequent analysis. Common data cleaning tasks include:

- **Removing duplicate rows:** Eliminating redundant data points.
- **Handling missing values:** Addressing gaps in the data.

- **Checking and converting data types:** Ensuring data consistency.
- **Dealing with outliers:** Identifying and managing extreme values.

Identifying and Removing Duplicate Rows

```
# Check for duplicate rows in the 'uber_15' DataFrame
duplicates = uber_15.duplicated()

# Count the number of duplicate rows
number_of_duplicates = duplicates.sum()
print("Number of duplicate rows:", number_of_duplicates)

# Remove duplicate rows and update the DataFrame in place
uber_15.drop_duplicates(inplace=True)

# Verify the removal of duplicates
print("Number of duplicate rows after cleaning:", uber_15.duplicated().sum())
print("Shape of DataFrame after removing duplicates:", uber_15.shape)
```

This code snippet first identifies duplicate rows using the `duplicated()` function, counts them using `sum()`, and then removes them using `drop_duplicates(inplace=True)`. The `inplace=True` argument modifies the DataFrame directly. Finally, it verifies the removal and displays the updated DataFrame's shape.

Handling Missing Values and Checking Data Types

```
# Check data types of each column
print(uber_15.dtypes)

# Check for missing values in each column
print(uber_15.isnull().sum())

# Convert 'pickup_date' column to datetime objects
uber_15['pickup_date'] = pd.to_datetime(uber_15['pickup_date'])

# Verify the data type conversion
print(uber_15.dtypes)
```

This code first displays the data types of each column using `dtypes`. Then, it checks for missing values using `isnull().sum()`. The code then converts the 'pickup_date' column to datetime objects using `pd.to_datetime()`, addressing the issue of incorrect data type. Finally, it rechecks the data types to confirm the conversion.

Understanding Data Types and Categories

Data can be broadly classified into:

- **Categorical Data:** Represents characteristics or groups.
  - **Object data type:** Represents strings in Python.
  - **Boolean data type:** Represents True/False values.
- **Numerical Data:** Represents measurable quantities.
  - **Discrete data:** Integer values, countable (e.g., age).
  - **Continuous data:** Float values, measurable with fractions (e.g., weight).

Deep Dive into Data Types and Creating Grouped Bar Charts with Pandas

This section delves deeper into data types in Python, specifically focusing on integer variations, and demonstrates how to create insightful grouped bar charts using Pandas to visualize Uber pickup patterns.

Unsigned vs. Signed Integers: Understanding the Nuances

In Python, integers can be categorized as:

- **Signed Integers:** Represent both positive and negative whole numbers.
  - Variations: `int8`, `int16`, `int32`, `int64` (increasing range and memory usage)
- **Unsigned Integers:** Represent only positive whole numbers.
  - Variations: `uint8`, `uint16`, `uint32`, `uint64` (increasing range and memory usage)

The number in the data type (e.g., 16 in `int16`) indicates the number of bits used to store the integer, which directly affects its range:

- **Signed Integer Range:** -2^(n-1) to 2^(n-1) - 1 (where 'n' is the number of bits)
- **Unsigned Integer Range:** 0 to 2^n - 1

For instance, `int16` uses 16 bits (2 bytes) and can store numbers from -32,768 to 32,767.

Choosing the Right Integer Type:

- Use `int64` for most general cases, as it offers the widest range.
- Opt for smaller integer types (`int8`, `int16`) if memory usage is a concern and your data falls within their limited ranges.

Extracting Insights: Which Month Has the Most Uber Pickups?

Let's analyze the Uber dataset to determine the month with the highest number of pickups:

```
# Extract the month name from the 'pickup_date' column
uber_15['month'] = uber_15['pickup_date'].dt.month_name()

# Display the first few rows of the DataFrame with the new 'month' column
print(uber_15.head())

# Create a frequency table of Uber pickups for each month
monthly_pickups = uber_15['month'].value_counts()
print(monthly_pickups)

# Create a bar plot to visualize monthly pickups
monthly_pickups.plot(kind='bar', figsize=(8, 6))
plt.xlabel('Month')
plt.ylabel('Number of Uber Pickups')
plt.title('Monthly Uber Pickups in New York City')
plt.show()
```

This code first extracts the month name from the 'pickup_date' column using `dt.month_name()`. It then creates a frequency table using `value_counts()` to count pickups for each month and visualizes it using a bar plot.

Advanced Visualization: Grouped Bar Charts for Weekday Trends within Months

To gain deeper insights into daily pickup patterns within each month, let's create a grouped bar chart:

```
# Extract weekday, day, hour, and minute from the 'pickup_date' column
uber_15['weekday'] = uber_15['pickup_date'].dt.day_name()
uber_15['day'] = uber_15['pickup_date'].dt.day
uber_15['hour'] = uber_15['pickup_date'].dt.hour
uber_15['minute'] = uber_15['pickup_date'].dt.minute

# Display the first four rows with the new features
print(uber_15.head(4))

# Create a cross-tabulation (pivot table) of month vs. weekday
pivot_table = pd.crosstab(index=uber_15['month'], columns=uber_15['weekday'])

# Display the pivot table
print(pivot_table)

# Create a grouped bar chart from the pivot table
pivot_table.plot(kind='bar', figsize=(12, 6))  # Adjust figsize for better visualization
plt.xlabel('Month')
plt.ylabel('Number of Uber Pickups')
plt.title('Uber Pickups in New York City: Month vs. Weekday')
plt.show()
```

This code first extracts 'weekday', 'day', 'hour', and 'minute' from the 'pickup_date' column. Then, it creates a cross-tabulation (pivot table) using `pd.crosstab()`, summarizing pickups for each weekday within each month. Finally, it generates a grouped bar chart from the pivot table to visualize the trends.

Unveiling Hourly Rush Patterns in New York City Using Point Plots with Seaborn

This section focuses on analyzing hourly Uber rush patterns in New York City using point plots, a powerful visualization tool provided by the Seaborn library. We'll delve into data aggregation, trend analysis, and drawing meaningful insights from the visualizations.

Understanding the Problem: Visualizing Hourly Rush

Our goal is to identify peak Uber pickup hours for each day of the week in New York City. To achieve this, we'll create a point plot that displays:

- **X-axis:** Hours of the day (0-23).
- **Y-axis:** Count of Uber pickups for each hour.
- **Separate lines:** Representing each day of the week.

Data Preparation: Grouping and Aggregating

Before plotting, we need to prepare the data by grouping it by weekday and hour, then calculating the total pickups for each group:

```
# Group the 'uber_15' DataFrame by 'weekday' and 'hour'
grouped_data = uber_15.groupby(['weekday', 'hour'])

# Calculate the size of each group (count of pickups)
hourly_pickups = grouped_data.size()

# Reset the index to convert the result into a DataFrame
summary = hourly_pickups.reset_index(name='count')

# Display the 'summary' DataFrame
print(summary)
```

This code snippet first groups the DataFrame by 'weekday' and 'hour'. It then calculates the size of each group (representing the count of pickups) using `size()`. The result is converted into a DataFrame using `reset_index()`, and the count column is named 'count'.

Creating the Point Plot with Seaborn

Now, let's create the point plot using Seaborn:

```
# Import Seaborn (if not already imported)
import seaborn as sns

# Create the point plot
plt.figure(figsize=(10, 6))  # Adjust figure size for better visualization
sns.pointplot(x='hour', y='count', hue='weekday', data=summary)
plt.xlabel('Hour of the Day')
plt.ylabel('Number of Uber Pickups')
plt.title('Hourly Uber Rush Patterns in New York City')
plt.show()
```

This code first imports the Seaborn library (if not already imported). Then, it creates a point plot using `sns.pointplot()`. The `x`, `y`, and `hue` parameters specify the columns for the x-axis, y-axis, and separate lines (hue) respectively. The `data` parameter points to our 'summary' DataFrame.

Analyzing the Plot: Insights and Observations

Observing the generated point plot, we can draw several insights:

- **Weekends (Saturday & Sunday):** Show similar trends with increased pickups during late night and early morning hours, suggesting a preference for late-night outings and early morning returns.
- **Saturday Evenings:** Experience a continuous surge in pickups throughout the evening, indicating a consistent demand for rides.
- **Sunday Evenings:** Show a decline in pickups after the evening, potentially due to people returning home after the weekend.

- **Weekdays:** Peak during late evenings, particularly Thursday, Friday, and Saturday, suggesting that New Yorkers might be starting their weekend festivities as early as Thursday night.

Visualizing Active Vehicles by Base Number with Box Plots and Violin Plots in Plotly

This section focuses on analyzing the distribution of active Uber vehicles per base number using insightful visualizations: box plots and violin plots. We'll leverage the Plotly library in Python to create interactive and informative plots.

Understanding the Problem: Visualizing Distribution

Our goal is to understand how the number of active Uber vehicles varies across different base numbers. We aim to visualize:

- **Central Tendency:** The typical number of active vehicles for each base.
- **Spread and Variability:** The range and distribution of active vehicles per base.
- **Outliers:** Any unusual or extreme values in active vehicle counts.

Introducing the Box Plot: A Five-Point Summary

Box plots provide a concise visual summary of data distribution using the five-point summary:

- **Minimum:** The smallest value (or a lower percentile like 1st).
- **First Quartile (Q1):** The value below which 25% of the data falls.
- **Median (Q2):** The middle value when data is ordered.
- **Third Quartile (Q3):** The value below which 75% of the data falls.
- **Maximum:** The largest value (or an upper percentile like 99th).
- **Whiskers:** Lines extending from the box, often representing 1.5 times the Interquartile Range (IQR = Q3 - Q1) or specific percentiles.
- **Outliers:** Data points beyond the whiskers, potentially indicating unusual values.

Introducing the Violin Plot: Distribution and Density

Violin plots combine the features of box plots with kernel density estimation, providing a richer visualization of data distribution:

- **Box Plot Features:** Includes the five-point summary (minimum, Q1, median, Q3, maximum) and whiskers.
- **Density Curve:** A mirrored density plot on each side of the box, showing the probability density of data points at different values.

Setting up Plotly for Interactive Visualizations

Before creating the plots, let's install and import the necessary Plotly components:

```
# Install necessary packages (if not already installed)
!pip install chart-studio plotly

# Import required modules
import chart_studio.plotly as py
import plotly.graph_objs as go
import plotly.express as px
from plotly.offline import download_plotlyjs, init_notebook_mode, plot, iplot

# Initialize Plotly in offline mode for Jupyter Notebook
init_notebook_mode(connected=True)
```

Creating the Box Plot with Plotly

```
# Create the box plot
fig = px.box(data_frame=uber_file, x='dispatching_base_number', y='active_vehicles')

# Show the plot
fig.show()
```

This code snippet creates a box plot using `px.box()`. The `data_frame` parameter specifies the DataFrame ('uber_file'), while `x` and `y` define the columns for the x-axis (base number) and y-axis (active vehicles).

Creating the Violin Plot with Plotly

```
# Create the violin plot
fig = px.violin(data_frame=uber_file, x='dispatching_base_number', y='active_vehicles')

# Show the plot
fig.show()
```

This code creates a violin plot using `px.violin()`, using the same parameters as the box plot to visualize the distribution of active vehicles for each base number.

Interpreting the Plots: Insights and Observations

By examining the generated box plots and violin plots, we can gain insights into:

- **Central Tendency:** The median line within each box reveals the typical number of active vehicles for each base.
- **Spread and Variability:** The width of the boxes and the shape of the density curves in violin plots indicate the range and distribution of active vehicles per base. Wider boxes and flatter curves suggest higher variability.

- **Outliers:** Data points plotted beyond the whiskers might represent bases with unusually high or low numbers of active vehicles.

Consolidating Uber Data and Understanding Latitude and Longitude

This section focuses on merging multiple Uber datasets into a single DataFrame for comprehensive analysis and provides a clear explanation of latitude and longitude in the context of geographic data.

Combining DataFrames for a Holistic View

To gain a complete picture of Uber pickup patterns, we'll combine data from multiple CSV files representing different months:

```
# Define the file path where all datasets are located
file_path = r"C:\Users\USER\Desktop\stats class\EDA Project 1\Datasets"  # Replace with your actual path

# Get a list of all files in the directory
all_files = os.listdir(file_path)

# Filter the list to include only relevant CSV files (adjust file names as needed)
files = [file for file in all_files if file.startswith('uber-raw-data') and file.endswith('.csv')]

# Remove specific files (sample data and Uber Jan-Feb data) from the list
files.remove('uber-raw-data-janjune-15_sample.csv')
files.remove('uber-raw-data-janfeb-15.csv')

# Create an empty DataFrame to store the combined data
final = pd.DataFrame()

# Iterate through the list of files, read each file into a DataFrame, and concatenate
for file in files:
    current_df = pd.read_csv(os.path.join(file_path, file))  # Construct full file path
    final = pd.concat([final, current_df], ignore_index=True)

# Display the shape (dimensions) of the combined DataFrame
print(final.shape)
```

This code first defines the directory path and retrieves a list of all files. It then filters the list to include only the relevant Uber CSV files, excluding the sample data and January-February data. An empty DataFrame `final` is created, and a loop iterates through the remaining files, reading each into a DataFrame using `pd.read_csv()` and concatenating it to the `final` DataFrame.

Data Cleaning: Removing Duplicates

```
# Check for duplicate rows and display the count
print("Number of duplicate rows:", final.duplicated().sum())

# Remove duplicate rows and update the DataFrame in place
final.drop_duplicates(inplace=True)

# Display the shape of the DataFrame after removing duplicates
print("Shape after removing duplicates:", final.shape)
```

This code checks for and removes duplicate rows from the combined DataFrame using `duplicated()` and `drop_duplicates()`, ensuring data integrity.

Understanding Latitude and Longitude

Latitude and longitude are crucial for representing geographic locations:

- **Latitude:** Imaginary lines circling the Earth parallel to the equator.
  - Measured in degrees north or south of the equator (0° latitude).
  - Range: -90° (South Pole) to 90° (North Pole).
- **Longitude:** Imaginary lines running from the North Pole to the South Pole.
  - Measured in degrees east or west of the Prime Meridian (0° longitude), which passes through Greenwich, England.
  - Range: -180° (west) to 180° (east).

How Latitude and Longitude Pinpoint Locations:

Every point on Earth can be uniquely identified by its latitude and longitude coordinates, forming a grid system:

- **Latitude:** Specifies the north-south position.

- **Longitude:** Specifies the east-west position.

For instance, New York City is located at approximately 40.7° N latitude and 74° W longitude.

Visualizing Uber Rush Hotspots in New York City with Folium Heatmaps

This section demonstrates how to create insightful heatmaps using the Folium library in Python to visualize Uber rush hotspots in New York City. We'll leverage the previously prepared data and geographic coordinates to pinpoint areas with high Uber pickup density.

Introduction to Heatmaps: Visualizing Density

Heatmaps are a powerful visualization technique for representing data density on a geographical map. They use color variations to indicate areas of high and low concentration:

- **Hotspots:** Regions with a high density of data points are displayed in warmer colors (e.g., red, orange, yellow).
- **Coldspots:** Regions with a low density of data points are displayed in cooler colors (e.g., blue, green).

Preparing the Data: Grouping by Latitude and Longitude

To create the heatmap, we need to aggregate Uber pickup counts by latitude and longitude:

```
# Group the 'final' DataFrame by 'Lat' and 'Lon' columns
grouped_data = final.groupby(['Lat', 'Lon'])

# Calculate the size of each group (count of pickups) and reset the index
rush_uber = grouped_data.size().reset_index(name='count')

# Display the first few rows of the 'rush_uber' DataFrame
print(rush_uber.head(6))
```

This code snippet groups the `final` DataFrame by latitude (`Lat`) and longitude (`Lon`) columns. It then calculates the size of each group (representing the count of pickups) using `size()` and resets the index to create a new DataFrame called `rush_uber` with columns for latitude, longitude, and count.

Creating a Base Map with Folium

We'll use the Folium library to generate a base map of New York City:

```
# Install Folium (if not already installed)
!pip install folium

# Import the folium library
import folium

# Create a base map centered around New York City
base_map = folium.Map(location=[40.7128, -74.0060], zoom_start=12)  # Adjust zoom_start as needed

# Display the base map
base_map
```

This code first installs Folium if it's not already installed. It then imports the library and creates a base map centered around New York City's coordinates (latitude: 40.7128, longitude: -74.0060). The `zoom_start` parameter controls the initial zoom level.

Adding the Heatmap Layer

Now, let's overlay the Uber pickup data as a heatmap layer on the base map:

```
# Import the HeatMap plugin from folium
from folium.plugins import HeatMap

# Create a HeatMap layer using the 'rush_uber' DataFrame
heat_layer = HeatMap(data=rush_uber[['Lat', 'Lon', 'count']].values, radius=10)  # Adjust radius as needed

# Add the HeatMap layer to the base map
heat_layer.add_to(base_map)

# Display the map with the heatmap layer
base_map
```

This code imports the `HeatMap` plugin from Folium. It then creates a `HeatMap` layer using the latitude, longitude, and count data from the `rush_uber` DataFrame. The `radius` parameter controls the size of the heat spots. Finally, it adds the heatmap layer to the base map.

Interpreting the Heatmap: Identifying Rush Hotspots

The resulting heatmap visually highlights areas in New York City with high Uber pickup density:

- **Midtown Manhattan:** Emerges as a significant hotspot, indicating a consistently high demand for Uber rides in this area.
- **Lower Manhattan:** Also shows a dense concentration of pickups, particularly in areas south of Midtown.
- **Upper Manhattan and Brooklyn Heights:** Exhibit moderate pickup activity compared to Midtown and Lower Manhattan.

Analyzing Uber Rush Patterns Using Pivot Tables and Data Styling

This section dives into analyzing Uber rush patterns by creating insightful pivot tables and enhancing their visual appeal using data styling techniques in Python.

Recap: Heatmaps and Geographic Analysis

In the previous session, we explored creating heatmaps to visualize Uber rush hotspots based on geographic density. We learned how to:

- Group data by latitude and longitude.
- Calculate pickup counts for each location.
- Visualize density using color variations on a map.

Analyzing Rush Patterns Across Days and Hours

Now, we'll shift our focus to analyzing rush patterns by examining Uber pickups across different days of the week and hours of the day. Our goal is to identify peak demand periods.

Creating a Pivot Table: A Different Approach

While we've previously used the `crosstab()` function to create pivot tables, let's explore an alternative method using `groupby()` and `unstack()`:

```
# Convert the 'Date/Time' column to datetime objects
final['Date/Time'] = pd.to_datetime(final['Date/Time'], format='%m/%d/%Y %H:%M:%S')

# Extract 'day' and 'hour' features from the 'Date/Time' column
final['day'] = final['Date/Time'].dt.dayofweek  # Monday=0, Sunday=6
final['hour'] = final['Date/Time'].dt.hour

# Group by 'day' and 'hour', calculate the size (count), and reset the index
grouped_data = final.groupby(['day', 'hour']).size().reset_index(name='count')

# Create the pivot table using unstack()
pivot = grouped_data.set_index(['day', 'hour'])['count'].unstack()

# Display the pivot table
print(pivot)
```

This code first converts the 'Date/Time' column to datetime objects and extracts 'day' and 'hour' features. It then groups the data by 'day' and 'hour', calculates the pickup count for each group, and uses `unstack()` to create the pivot table with days as rows and hours as columns.

Enhancing Visual Clarity with Data Styling

To make the pivot table more visually appealing and easier to interpret, we can apply data styling:

```
# Apply background gradient styling to the pivot table
styled_pivot = pivot.style.background_gradient(cmap='coolwarm')  # Choose a suitable colormap

# Display the styled pivot table
styled_pivot
```

This code uses the `style.background_gradient()` method to apply a color gradient based on the values in the pivot table. The `cmap` parameter allows you to choose a colormap (e.g., 'coolwarm', 'viridis'). Darker colors now represent higher Uber pickup counts.

Interpreting the Results: Identifying Rush Hours

The styled pivot table provides a clear visual representation of Uber rush patterns:

- **Weekdays:** Show a consistent pattern with peak demand during morning (7-9 AM) and evening (5-8 PM) commuting hours.
- **Friday and Saturday Evenings:** Exhibit higher pickup counts compared to other weekdays, indicating increased activity during weekend nightlife hours.
- **Early Morning Hours (1-4 AM):** Show relatively lower demand across all days.

Automating Analysis with Python Functions: Streamlining Pivot Table Creation

This section focuses on automating data analysis tasks, specifically creating pivot tables, using Python functions. We'll learn how functions can streamline repetitive code and enhance analysis efficiency.

Recap: Pivot Tables and Data Styling

In the previous sessions, we explored creating and styling pivot tables to analyze Uber rush patterns:

- **Pivot Tables:** Summarize data based on two categorical variables (e.g., day and hour), showing aggregated values (e.g., pickup counts) for each combination.
- **Data Styling:** Enhances the visual clarity of pivot tables using color gradients and other formatting options.

The Need for Automation: Reducing Repetitive Code

Imagine needing to create multiple pivot tables with different combinations of variables (e.g., day vs. hour, weekday vs. weekend). Writing the same code repeatedly for each combination would be tedious and inefficient.

Introducing Functions: Automating Tasks

Functions in Python allow us to encapsulate a block of code that performs a specific task. We can then reuse this function multiple times with different inputs, automating the process.

Defining a Function for Pivot Table Creation

Let's define a function called `generate_pivot_table` that takes a DataFrame and two column names as input and returns a styled pivot table:

```python
import pandas as pd

def generate_pivot_table(df, column1, column2):
    """
    Generates a styled pivot table from a DataFrame.

    Args:
        df (pd.DataFrame): The input DataFrame.
        column1 (str): The name of the first column for grouping.
        column2 (str): The name of the second column for grouping.

    Returns:
        pd.DataFrame.style: The styled pivot table.
    """

    # Group by the specified columns, calculate the size (count), and reset the index
    grouped_data = df.groupby([column1, column2]).size().reset_index(name='count')

    # Create the pivot table using unstack()
    pivot = grouped_data.set_index([column1, column2])['count'].unstack()

    # Apply background gradient styling to the pivot table
    styled_pivot = pivot.style.background_gradient(cmap='coolwarm')

    return styled_pivot
```

This function:

1. Takes a DataFrame (`df`), the first column name (`column1`), and the second column name (`column2`) as input.
2. Groups the DataFrame by the specified columns, calculates the size (count), and resets the index.

3. Creates the pivot table using `unstack()`.
4. Applies background gradient styling using `style.background_gradient()`.
5. Returns the styled pivot table.

Using the Function: Simplifying Analysis

Now, we can easily generate pivot tables for different variable combinations:

```
# Generate a pivot table for 'day' vs. 'hour'
pivot_day_hour = generate_pivot_table(final, 'day', 'hour')
display(pivot_day_hour)  # Use 'display' in Jupyter Notebook for formatted output

# Generate a pivot table for 'weekday' vs. 'hour' (assuming 'weekday' column exists)
pivot_weekday_hour = generate_pivot_table(final, 'weekday', 'hour')
display(pivot_weekday_hour)
```

By simply calling the `generate_pivot_table` function with different column names, we automate the process of creating and styling pivot tables, significantly reducing code duplication and improving analysis efficiency.