Line by line Code explaination

Workshop Agenda

- **Section 0: Imports & Plotting Settings** - Setting up the environment.
- **Section 1: Load Dataset** - Getting the data into the notebook.
- **Section 2: Quick Summary & Missing Values** - Initial data inspection.
- **Section 3: Clean Types & Parse Dates** - Preparing the data for analysis.
- **Section 4: Inject Missing Values (Optional)** - A quick demo for handling missing data.
- **Section 5: Imputation** - Filling in missing values.
- **Section 6: Derived Features** - Creating new, informative columns.
- **Section 7: Univariate EDA — Numeric Features** - Understanding single numeric variables.
- **Section 8: Univariate EDA — Categorical Features** - Understanding single categorical variables.

- **Section 9: Target Variable** `purchase` - Analyzing the target for a machine learning task.
- **Section 10: Bivariate — Numeric vs.** `purchase` - Finding relationships between numeric features and the target.
- **Section 11: Bivariate — Categorical vs.** `purchase` - Finding relationships between categorical features and the target.
- **Section 12: Multivariate — Correlation Matrix** - Exploring relationships among multiple numeric features.
- **Section 13: Pairwise Scatter (Sample)** - Visualizing relationships between numeric features.
- **Section 14: Segmented Analysis** - Deep diving into specific user groups.
- **Section 15: Derived Metrics — Check Usefulness** - Evaluating the new features.
- **Section 16: Time-series Analysis** - Discovering daily and hourly patterns.
- **Section 17: Summary & Next Steps** - Concluding the analysis and planning future work.

Section 0: Imports & plotting settings

Code:

Python

```
# Basic imports
import pandas as pd            # for data manipulation (tables)
import numpy as np             # for numeric operations
import matplotlib.pyplot as plt  # for plotting charts
import os                      # for file path checks

# Make charts look clean and readable
plt.style.use('ggplot')        # a simple, presentation-friendly style
   plt.rcParams['figure.dpi'] = 110
plt.rcParams['font.size'] = 11
```

```
# Paths for the dataset and cleaned output
DATA_PATH = "/mnt/data/Realistic_E-Commerce_Dataset.csv"
CLEAN_PATH = "/mnt/data/ecomm_cleaned_for_modeling_simple.csv"

print("Ready. DATA_PATH:", DATA_PATH)
```

- **Purpose:** The purpose of this cell is to prepare the Python environment for data analysis. It loads necessary libraries and configures plotting settings to ensure all visualizations are consistent and easy to read.
- **Code Line by Line:**
  - `# Basic imports`: This is a comment that explains the purpose of the following lines.
  - `import pandas as pd`: This line imports the `pandas` library and assigns it the alias `pd`. Pandas is used for working with structured data in a tabular format, like a spreadsheet.
  - `import numpy as np`: This imports the `numpy` library as `np`, which is essential for numerical operations.
  - `import matplotlib.pyplot as plt`: This imports the `matplotlib.pyplot` module as `plt`, which is used for creating a variety of static charts and plots.
  - `import os`: This imports the `os` module, which provides a way to interact with the operating system, such as checking for file paths.
  - `# Make charts look clean and readable`: A comment explaining the purpose of the next few lines.
  - `plt.style.use('ggplot')`: This line sets a specific plotting style. The `'ggplot'` style is known for being presentation-friendly with a clean, grid-based aesthetic.
  - `plt.rcParams['figure.dpi'] = 110`: This line increases the dots-per-inch (DPI) of the figures, which makes the images sharper and higher quality.
  - `plt.rcParams['font.size'] = 11`: This sets the default font size for all text elements in the plots to 11, improving readability.

- `# Paths for the dataset and cleaned output`: A comment defining the purpose of the next two lines.
- `DATA_PATH = "/mnt/data/Realistic_E-Commerce_Dataset.csv"`: This line defines a variable to store the file path for the raw dataset.
- `CLEAN_PATH = "/mnt/data/ecomm_cleaned_for_modeling_simple.csv"`: This line defines a variable for the path where the cleaned dataset will be saved.
- `print("Ready. DATA_PATH:", DATA_PATH)`: This line prints a confirmation message, showing that the setup is complete and displaying the data path.

- **Workshop Use Case:** Explain the concept of a "virtual environment" or "notebook setup." Emphasize that these steps ensure reproducibility and a professional look for all analysis and reports.
- **Customizations for Graphs:**
  - **Change Plotting Style:** You can show students how to change the style by using different strings in `plt.style.use()`. Examples to suggest are `'seaborn-v0_8-whitegrid'` or `'dark_background'` for a different visual feel.
  - **Adjust Font Size and DPI:** Encourage students to play with `plt.rcParams['figure.dpi']` and `plt.rcParams['font.size']` to see how these parameters affect the visual appearance of their plots, making them clearer for presentations or reports.

Section 1: Load dataset

Code:

Python

```
# Check the file exists
if not os.path.exists(DATA_PATH):
    raise FileNotFoundError(f"Please upload the CSV to {DATA_PATH}")

# Read CSV into a pandas DataFrame
df = pd.read_csv(DATA_PATH)

# Print shape (#rows, #columns) and show first 5 rows
```

```
print('Shape:', df.shape)   # number of rows and columns
print('\nColumns:', list(df.columns))  # list of column names
display(df.head())          # show first 5 rows for a quick look
```

- **Purpose:** This cell's main goal is to load the dataset into a Pandas DataFrame, a fundamental data structure for data analysis in Python. It also performs a quick check to ensure the file is present.
- **Code Line by Line:**
  - `# Check the file exists`: A comment explaining the purpose of the following lines.
  - `if not os.path.exists(DATA_PATH):`: This is a conditional statement that checks if the file at the specified `DATA_PATH` exists.
  - `raise FileNotFoundError(...)`: If the file is not found, this line stops the code execution and displays a specific error message, prompting the user to upload the file.
  - `# Read CSV into a pandas DataFrame`: A comment describing the next action.
  - `df = pd.read_csv(DATA_PATH)`: This is the core line that reads the comma-separated values (CSV) file and stores the data in a DataFrame variable named `df`.
  - `# Print shape (#rows, #columns) and show first 5 rows`: A comment outlining the next steps for a quick data inspection.
  - `print('Shape:', df.shape)`: This prints the dimensions of the DataFrame as a tuple `(rows, columns)`.
  - `print('\nColumns:', list(df.columns))`: This prints a list of all the column names in the DataFrame, which is useful for understanding the available features.
  - `display(df.head())`: This displays the first 5 rows of the DataFrame, providing a quick visual overview of the data, its columns, and some sample values.
- **Workshop Use Case:** Stress that this is the very first step in any data analysis project. The `df.head()` command is the most important for gaining initial insights and forming hypotheses about the data before a single line of analysis code is written.

Section 2: Quick summary & missing values

Code:

Python

```
# Basic numeric summary
numeric = df.select_dtypes(include=[np.number])
print('Numeric columns summary:')
display(numeric.describe().T)   # mean, std, min, max for numeric features

# Missing values per column
print('\nMissing values per column:')
missing = df.isna().sum().sort_values(ascending=False)
display(missing.head(30))
```

- **Purpose:** This cell provides a high-level statistical summary of the dataset to guide the cleaning process. It gives an overview of the numeric data and a count of missing values in each column.
- **Code Line by Line:**
  - `# Basic numeric summary`: A comment to introduce the first part of the cell.
  - `numeric = df.select_dtypes(include=[np.number])`: This line creates a new DataFrame called `numeric` that contains only the columns with a numeric data type.
  - `print('Numeric columns summary:')`: Prints a heading for the summary table.
  - `display(numeric.describe().T)`:
    - `numeric.describe()`: This method generates a table of summary statistics for each numeric column, including count, mean, standard deviation, minimum, maximum, and quartiles.
    - `.T`: This transposes the table (swaps rows and columns) to make it more readable, with the statistics as rows and the columns as columns.
  - `# Missing values per column`: A comment to introduce the second part of the cell.
  - `print('\nMissing values per column:')`: Prints a new heading.

    - `missing = df.isna().sum().sort_values(ascending=False)`:
        - `df.isna()`: This method returns a DataFrame of the same shape as `df` but with `True` where values are missing (`NaN`) and `False` otherwise.
        - `.sum()`: This method sums the `True` values (which are treated as 1) for each column, giving a total count of missing values per column.
        - `.sort_values(ascending=False)`: This sorts the resulting Series in descending order, so the columns with the most missing values are at the top, making them easy to identify.
    - `display(missing.head(30))`: This displays the first 30 columns from the sorted list of missing values, showing which columns have the most missing data.
- **Workshop Use Case:** Explain to students that this is the diagnostic step of data cleaning. The output from this cell will determine which columns need imputation or transformation in later steps. The comparison of the mean and median in the `describe()` output can also hint at skewness and the presence of outliers.

Section 3: Clean types and parse dates

Code:

Python

```
# Parse 'date' column to datetime so we can extract hour/day later
if 'date' in df.columns:
    df['date'] = pd.to_datetime(df['date'], errors='coerce')  # convert strings to datetime; invalid → NaT

# Convert integer-like columns safely (preserve NA)
int_cols = ['user_id','session_id','age','time_spent_minutes','pages_viewed','scroll_depth','clicks','wishlist_items','cart_items']
for col in int_cols:
    if col in df.columns:
        df[col] = pd.to_numeric(df[col], errors='coerce')  # non-numeric → NaN

# Show dtypes after conversion
```

```
print('Dtypes after conversions:')
print(df.dtypes.head(15))
```

- **Purpose:** This cell focuses on a critical data cleaning step: ensuring that each column has the correct data type. It is particularly important to convert the date column to a proper `datetime` type for any time-based analysis.
- **Code Line by Line:**
  - `# Parse 'date' column...`: A comment explaining the goal of converting the date column.
  - `if 'date' in df.columns:`: This checks if the `date` column exists in the DataFrame, making the code robust.
  - `df['date'] = pd.to_datetime(df['date'], errors='coerce')`:
    - `pd.to_datetime()`: This function attempts to convert the specified column to a `datetime` object.
    - `errors='coerce'`: This is a crucial argument. If it encounters a value that cannot be converted to a date, it will replace that value with `NaT` (Not a Time), which is a specific type of missing value for datetimes. This prevents the code from crashing.
  - `# Convert integer-like columns safely...`: A comment for the next code block.
  - `int_cols = [...]`: This defines a list of columns that are expected to contain integer-like data.
  - `for col in int_cols:`: This starts a loop that iterates through each column name in the `int_cols` list.
  - `if col in df.columns:`: This ensures the code won't fail if a column from the list is not present in the dataset.
  - `df[col] = pd.to_numeric(df[col], errors='coerce')`: This line attempts to convert the column to a numeric data type. `errors='coerce'` handles any non-numeric values by replacing them with `NaN` (Not a Number), preserving the integrity of the rest of the column.
  - `# Show dtypes after conversion`: A comment to indicate the next step.

    - `print('Dtypes after conversions:')`: Prints a heading for the output.
    - `print(df.dtypes.head(15))`: This prints the data type (`dtype`) of the first 15 columns, allowing a quick verification of the successful conversions.
- **Workshop Use Case:** This is where you explain the importance of data types for computation. Explain that you can't perform math on a column that is a string, and you can't extract a day of the week from a column that isn't a proper `datetime` object. The use of `errors='coerce'` is a good practical tip to highlight for dealing with real-world, messy data.

Section 4. (Optional) Inject a few missing values for demo

Code:

Python

```
# Only inject if data has zero missing values (keeps original data otherwise)
if df.isna().sum().sum() == 0:
    rng = np.random.default_rng(1)
    idx = rng.choice(df.index, size=6, replace=False)
    # Make some ages missing
    if 'age' in df.columns:
        df.loc[idx[:2], 'age'] = np.nan
    # Make time_spent_minutes missing
    if 'time_spent_minutes' in df.columns:
        df.loc[idx[2:4], 'time_spent_minutes'] = np.nan
    # Make product_category missing
    if 'product_category' in df.columns:
        df.loc[idx[4:], 'product_category'] = np.nan
    print('Injected missing at indices:', list(idx))
else:
    print('Dataset already has missing values; no injection done.')

# Show missing summary
display(df.isna().sum().sort_values(ascending=False).head(10))
```

- **Purpose:** This cell is a teaching tool designed to simulate the messy nature of real-world data. It injects a controlled number of missing values into a clean dataset, which allows the instructor to demonstrate the imputation process in the next section.
- **Code Line by Line:**
  - `# Only inject if data has zero missing values...`: A comment to explain the conditional logic.
  - `if df.isna().sum().sum() == 0:`:
    - `df.isna()`: As seen before, this returns a boolean DataFrame with `True` for missing values.
    - `.sum()`: The first `.sum()` counts the missing values for each column.
    - `.sum()`: The second `.sum()` adds up all those counts to get a single total of missing values in the entire DataFrame. The condition is `True` only if the total is zero, meaning the dataset is perfectly clean.
  - `rng = np.random.default_rng(1)`: This line creates a new, modern random number generator instance using NumPy's `default_rng()` function. The number `1` is a seed, which ensures that the sequence of "random" numbers generated is the same every time the code is run, making the example reproducible.
  - `idx = rng.choice(df.index, size=6, replace=False)`:
    - `rng.choice()`: This method randomly selects items from a given array.
    - `df.index`: This provides the list of all row indices from which to choose.
    - `size=6`: This specifies that we want to choose exactly 6 random indices.
    - `replace=False`: This is a critical argument that ensures each selected index is unique and not picked more than once.
  - `if 'age' in df.columns: ...`: This is a check to ensure the `age` column exists before attempting to modify it.
  - `df.loc[idx[:2], 'age'] = np.nan`:
    - `df.loc[...]`: This is the Pandas way of selecting rows and columns by their labels.

        - `idx[:2]`: This selects the first two indices from our list of 6 random indices.
        - `'age'`: This specifies the column to modify.
        - `= np.nan`: This assigns the value `NaN` (a standard representation for a missing value) to the selected cells.
    - The next two blocks of code repeat this process for `time_spent_minutes` and `product_category` using the next set of indices.
    - `print('Injected missing at indices:', list(idx))`: This prints the indices that were modified so the user can verify the changes.
    - `else: print(...)`: This block is executed if the dataset already has missing values, preventing the injection of more.
    - `display(df.isna().sum()...)`: This final line is a quick check to show the updated count of missing values, confirming that the injection was successful.
- **Workshop Use Case:** Emphasize that this step is purely for teaching purposes. In a real project, you would skip this cell and proceed directly to imputation if you found missing values in Section 2. Explain the concept of "reproducibility" and how using a random seed with `default_rng()` is crucial for ensuring that your results are consistent.

Section 5. Imputation (simple and explainable)

Code:

Python

```
# Fill numeric missing values with median (explainable and robust to outliers)
for col in ['age','time_spent_minutes']:
    if col in df.columns:
        median = df[col].median(skipna=True)
        df[col] = df[col].fillna(median)
        print(f'Filled {col} missing with median =', median)

# Fill categorical missing with 'Unknown' so they become a category
if 'product_category' in df.columns:
```

```
    df['product_category'] = df['product_category'].fillna('Unknown')
    print('Filled product_category missing with Unknown')

# Quick check
display(df.isna().sum().sort_values(ascending=False).head(10))
```

- **Purpose:** The purpose of this cell is to fill in the missing values in the dataset using simple and robust methods. This step is necessary because most machine learning models cannot handle missing values.
- **Code Line by Line:**
  - `# Fill numeric missing values with median...`: A comment explaining the first imputation method.
  - `for col in ['age','time_spent_minutes']:`: This loop iterates through a list of numeric columns that are known to have or were just injected with missing values.
  - `if col in df.columns:`: This conditional statement checks if the current column exists in the DataFrame, making the code more resilient to different datasets.
  - `median = df[col].median(skipna=True)`:
    - `df[col].median()`: This method calculates the median of the specified column. The median is chosen because it is a measure of central tendency that is less affected by outliers than the mean.
    - `skipna=True`: This ensures that any existing `NaN` values are ignored when calculating the median.
  - `df[col] = df[col].fillna(median)`:
    - `df[col].fillna()`: This method fills all the `NaN` values in the column with the value provided as an argument.
    - `median`: The calculated median is used as the fill value.
  - `print(...)`: A confirmation message is printed, showing which column was filled and with what median value.
  - `# Fill categorical missing with 'Unknown'...`: A comment explaining the second imputation method for categorical data.

- `if 'product_category' in df.columns:`: Checks for the existence of the `product_category` column.
- `df['product_category'] = df['product_category'].fillna('Unknown')`: This fills missing values in the `product_category` column with the string `'Unknown'`. This is a good practice as it preserves the information that the value was missing and creates a new, distinct category for the model to learn from.
- `print(...)`: Prints a confirmation message.
- `# Quick check`: A comment for the final check.
- `display(df.isna().sum()...)`: This final line re-runs the missing value check to confirm that the imputation process was successful and there are no more missing values in the target columns.

- **Workshop Use Case:** Discuss the different imputation strategies (mean, median, mode, constant value, or even model-based imputation) and their trade-offs. Explain why the median is a good default for numeric data due to its robustness to outliers, and why `'Unknown'` is a safe choice for categorical data.

Section 6. Derived features (simple and useful)

Code:

Python

```
# Extract day of week and hour from the timestamp
if 'date' in df.columns:
    df['day_of_week'] = df['date'].dt.day_name()   # Monday, Tuesday, ...
    df['hour'] = df['date'].dt.hour                # 0-23 hour of the day
else:
    df['day_of_week'] = 'Unknown'
    df['hour'] = 0

# Age groups: simple bins
df['age_group'] = pd.cut(df['age'], bins=[17,25,40,65], labels=['18-25','26-40','41-65'])
```

```
# cart_to_wishlist_ratio: intent measure (cart items divided by wishlist size)
df['cart_to_wishlist_ratio'] = df.apply(lambda r: (r['cart_items'] / r['wishlist_ite
ms']) if (r.get('wishlist_items',0) and r['wishlist_items']>0) else r['cart_items'],
axis=1)

# engagement_score: small composite metric from time, pages, clicks (weight
s are chosen for simplicity)
df['engagement_score'] = (df['time_spent_minutes']/60)*0.4 + (df['pages_vie
wed']/20)*0.35 + (df['clicks']/50)*0.25

# value_per_cart_item: avg session value divided by cart items (fallback to avg
if cart is zero)
df['value_per_cart_item'] = df.apply(lambda r: (r['avg_session_value']/r['cart_it
ems']) if (r.get('cart_items',0) and r['cart_items']>0) else r['avg_session_valu
e'], axis=1)

# Show a few rows with derived columns
display(df[['date','age','age_group','hour','cart_items','wishlist_items','cart_to_w
ishlist_ratio','engagement_score','value_per_cart_item']].head())
```

- **Purpose:** This cell demonstrates the concept of "feature engineering". It creates new features from the existing ones that are often more insightful and predictive for a machine learning model.
- **Code Line by Line:**
  - `# Extract day of week and hour from the timestamp`: A comment explaining the purpose.
  - `if 'date' in df.columns: ... else: ...`: This block checks for the presence of the `date` column and either extracts the time-based features or creates placeholder columns if the `date` column is missing.
  - `df['day_of_week'] = df['date'].dt.day_name()`: This line uses the `.dt` accessor, available for `datetime` columns, to extract the day name (e.g., 'Monday') and store it in a new column.
  - `df['hour'] = df['date'].dt.hour`: Similarly, this extracts the hour (0-23) from the timestamp.

- `# Age groups: simple bins`: A comment for the next part.
- `df['age_group'] = pd.cut(df['age'], ...)`: `pd.cut()` is a function that bins continuous data into discrete intervals or categories. It takes the `age` column, a list of bin boundaries (`bins`), and a list of labels for the new categories (`labels`). This is useful for creating age cohorts.
- `# cart_to_wishlist_ratio...`: A comment explaining the new ratio feature.
- `df['cart_to_wishlist_ratio'] = df.apply(...)`:
  - `df.apply()`: This method applies a function along a DataFrame's rows (`axis=1`) or columns.
  - `lambda r: ...`: A `lambda` function is a small, anonymous function. Here, it takes a row (`r`) as input.
  - `if ... else ...`: This is a conditional expression to handle a potential division-by--zero error. If `wishlist_items` is greater than zero, it calculates the ratio; otherwise, it assigns the value of `cart_items` to avoid an error. This is a robust way to handle edge cases.
- `# engagement_score...`: A comment for the composite score.
- `df['engagement_score'] = (...)`: This line creates a new feature by combining three existing ones with simple weights. This is a heuristic approach to create a single metric that represents overall user engagement.
- `# value_per_cart_item...`: A comment for the final derived metric.
- `df['value_per_cart_item'] = df.apply(...)`: Similar to the `cart_to_wishlist_ratio`, this calculates the average value of an item in the cart, with a fallback to the `avg_session_value` to handle cases where `cart_items` is zero.
- `display(df[[...]].head())`: This shows the first few rows of the DataFrame, specifically displaying the new columns, to confirm the feature engineering was successful.

- **Workshop Use Case:** Emphasize that feature engineering is a creative process based on domain knowledge. Explain that these new features are often more predictive than the raw data because they directly capture a specific behavioral hypothesis.

Section 7. Univariate EDA — Numeric features (simple plots)

Code:

Python

```
num_cols = ['age','time_spent_minutes','pages_viewed','avg_session_value','engagement_score','value_per_cart_item']

for col in num_cols:
    if col in df.columns:
        data = df[col].dropna()
        plt.figure(figsize=(6,3))
        plt.hist(data, bins=20, edgecolor='black')
        plt.title(col)
        # show mean and median lines
        plt.axvline(data.mean(), color='red', linestyle='--', label=f'mean={data.mean():.1f}')
        plt.axvline(data.median(), color='black', linestyle='-', label=f'median={data.median():.1f}')
        plt.legend()
        plt.show()
        # print short outcome
        print(f"{col} → mean: {data.mean():.2f}, median: {data.median():.2f}, missing: {data.isna().sum()}\n")
```

- **Purpose:** The purpose of this cell is to perform a univariate analysis of the numeric features. Univariate analysis looks at the distribution of a single variable, and histograms are a powerful way to do this for numeric data.
- **Code Line by Line:**
    - `num_cols = [...]`: This line defines a list of numeric columns to be analyzed.
    - `for col in num_cols:`: This loop iterates through each column name in the `num_cols` list.
    - `if col in df.columns:`: A check to ensure the column exists before processing.

    - `data = df[col].dropna()`: This line selects the current column and removes any missing values, which is necessary for creating the histogram.
    - `plt.figure(figsize=(6,3))`: This creates a new figure for the plot with a specified width and height in inches.
    - `plt.hist(data, bins=20, edgecolor='black')`:
        - `plt.hist()`: This is the function that generates a histogram.
        - `data`: The data to plot.
        - `bins=20`: This specifies that the data should be divided into 20 equally sized bins (bars).
        - `edgecolor='black'`: This adds a black line around each bar, making the histogram clearer.
    - `plt.title(col)`: Sets the title of the plot to the name of the current column.
    - `plt.axvline(...)`: This draws a vertical line on the plot.
    - `data.mean()`: Calculates the mean of the data for the red dashed line.
    - `data.median()`: Calculates the median for the black solid line.
    - `plt.legend()`: Displays the labels for the mean and median lines.
    - `plt.show()`: Displays the generated plot.
    - `print(...)`: Prints a summary of the mean, median, and missing values for each column.
- **Workshop Use Case:** Teach students how to read a histogram. Explain that the shape of the histogram reveals important characteristics like skewness and the presence of outliers. A big difference between the mean and median often indicates a skewed distribution.
- **Customizations for Graphs:**
    - **Change Bin Count:** Show how changing `bins` (e.g., `bins=50`) can provide more or less detail in the distribution.
    - **Add Kernel Density Estimate (KDE):** For a smoother representation, you could introduce `seaborn` and replace `plt.hist()` with a KDE plot (e.g., `sns.kdeplot(data, fill=True)`), which shows a continuous probability density curve.

Section 8. Univariate EDA — Categorical features (counts)

Code:

Python

```
cat_cols = ['gender','product_category','membership_status','device_type','traffic_source','time_of_day']

for col in cat_cols:
    if col in df.columns:
        counts = df[col].value_counts()
        plt.figure(figsize=(6,3))
        plt.bar(counts.index.astype(str), counts.values)
        plt.title(col)
        plt.xticks(rotation=30)
        plt.show()
        print(f"{col} top 3: {list(counts.head(3).index)} → counts {list(counts.head(3).values)}\n")
```

- **Purpose:** This cell focuses on the univariate analysis of categorical features. Bar charts are used to visualize the frequency of each category, which helps to identify the most common values.
- **Code Line by Line:**
  - `cat_cols = [...]`: Defines a list of categorical columns to analyze.
  - `for col in cat_cols:`: Loops through each column in the list.
  - `if col in df.columns:`: Checks if the column exists.
  - `counts = df[col].value_counts()`: This is a key line that counts the number of occurrences of each unique value in the column. It returns a Pandas Series with the unique values as the index and their counts as the data.
  - `plt.figure(figsize=(6,3))`: Creates a new figure for the plot.
  - `plt.bar(counts.index.astype(str), counts.values)`:
    - `plt.bar()`: The function to create a bar chart.

        - `counts.index.astype(str)`: Uses the categories (the index of the `counts` Series) for the x-axis, converted to strings.
        - `counts.values`: Uses the counts for the y-axis.
    - `plt.title(col)`: Sets the plot title to the column name.
    - `plt.xticks(rotation=30)`: Rotates the x-axis labels by 30 degrees to prevent them from overlapping, especially with long category names.
    - `plt.show()`: Displays the plot.
    - `print(...)`: Prints a summary of the top 3 most frequent categories and their counts.
- **Workshop Use Case:** Explain how this analysis can reveal a class imbalance within categorical features. If one category has a very high count and others are very low, it might impact how a model handles that feature. This is also a good opportunity to discuss one-hot encoding for models.
- **Customizations for Graphs:**
    - **Horizontal Bars:** Show how to use `plt.barh()` for horizontal bar charts. This is often better when category labels are long.
    - **Proportions instead of Counts:** Instead of plotting raw counts, students could plot the percentage of each category using `counts.value_counts(normalize=True) * 100` for the y-axis.

Section 9. Target variable `purchase` (balance)

Code:

Python

```
if 'purchase' in df.columns:
    rate = df['purchase'].mean()       # fraction of rows where purchase==1
    plt.figure(figsize=(4,3))
    plt.pie([1-rate, rate], labels=[f'No ({(1-rate):.1%})', f'Yes ({rate:.1%})'], autop
ct='%1.1f%%')
    plt.title('Purchase Rate')
    plt.show()
```

```
    print('Purchase rate =', round(rate,3))
else:
    print('No purchase column found.')
```

- **Purpose:** This cell focuses on the target variable for the machine learning problem: `purchase`. The goal is to check for class imbalance, which is a common issue in classification tasks and can significantly affect model performance.
- **Code Line by Line:**
  - `if 'purchase' in df.columns:`: Checks if the target column exists.
  - `rate = df['purchase'].mean()`: Since the `purchase` column contains only `0`s and `1`s, the mean of the column is equivalent to the proportion or fraction of `1`s, which represents the purchase rate.
  - `plt.figure(figsize=(4,3))`: Creates a new figure for the pie chart.
  - `plt.pie([1-rate, rate], ...)`:
    - `plt.pie()`: The function to create a pie chart.
    - `[1-rate, rate]`: The sizes of the two slices of the pie. The first slice is the proportion of `0`s (no purchase), and the second is the proportion of `1`s (yes purchase).
    - `labels=[...]`: Provides text labels for each slice, including the percentage formatted nicely.
    - `autopct='%1.1f%%'`: This formats the percentage value to be displayed on the pie slices, showing one decimal place.
  - `plt.title('Purchase Rate')`: Sets the title of the chart.
  - `plt.show()`: Displays the plot.
  - `print('Purchase rate =', round(rate,3))`: Prints the exact purchase rate, rounded to three decimal places.
- **Workshop Use Case:** Use this to introduce the concept of "class imbalance" in machine learning. Explain that if the rate is very low (e.g., 5%), a simple model that always predicts "No purchase" would achieve 95% accuracy, but it

would be useless from a business perspective. This highlights the need for more sophisticated evaluation metrics beyond simple accuracy.

Section 10. Bivariate — Numeric vs purchase (boxplots)

Code:

Python

```
cols = ['time_spent_minutes','pages_viewed','engagement_score','value_per_cart_item']

for col in cols:
    if col in df.columns and 'purchase' in df.columns:
        no = df[df['purchase']==0][col].dropna()
        yes = df[df['purchase']==1][col].dropna()
        plt.figure(figsize=(6,3))
        plt.boxplot([no, yes], labels=['No','Yes'])
        plt.title(f'{col} by Purchase')
        plt.show()
        print(f'{col} medians → No: {no.median():.2f}, Yes: {yes.median():.2f}\n')
```

- **Purpose:** The purpose of this cell is to perform a bivariate analysis, exploring the relationship between numeric features and the binary target variable, `purchase`. Boxplots are a perfect tool for this, as they allow for a quick comparison of the distributions of a numeric feature for each of the two target classes.
- **Code Line by Line:**
  - `cols = [...]`: Defines a list of numeric columns to analyze.
  - `for col in cols:`: Loops through each numeric column.
  - `if col in df.columns and 'purchase' in df.columns:`: Checks for the existence of both the numeric column and the target column.

- `no = df[df['purchase']==0][col].dropna()`: This line filters the DataFrame to get all rows where `purchase` is `0` (no purchase) and then selects the current numeric column (`col`) from that filtered result. `dropna()` removes any remaining missing values.
- `yes = df[df['purchase']==1][col].dropna()`: This does the same for rows where `purchase` is `1` (yes purchase).
- `plt.figure(figsize=(6,3))`: Creates a new plot figure.
- `plt.boxplot([no, yes], labels=['No','Yes'])`: This function creates a side-by-side boxplot. The first list item is the data for the "No" boxplot, and the second is for the "Yes" boxplot. The `labels` argument provides the names for each boxplot on the x-axis.
- `plt.title(...)`: Sets the title of the plot.
- `plt.show()`: Displays the plot.
- `print(...)`: Prints the median values for both groups, providing a clear numeric comparison to complement the visual plot.

- **Workshop Use Case:** Teach students how to read and interpret boxplots. Explain that a significant difference in the median lines (the horizontal lines inside the boxes) between the "No" and "Yes" groups indicates a strong predictive relationship. For example, if the median time spent for purchasers is much higher than for non-purchasers, `time_spent_minutes` is likely a good feature for a model.

Section 11. Bivariate — Categorical vs purchase (conversion rates)

Code:

Python

```
cat_list = ['product_category','membership_status','gender','device_type','traffi
c_source','time_of_day']

for col in cat_list:
    if col in df.columns and 'purchase' in df.columns:
```

```
    conv = df.groupby(col)['purchase'].mean().sort_values(ascending=False)
    plt.figure(figsize=(6,3))
    plt.bar(conv.index.astype(str), conv.values)
    plt.title('Conversion by ' + col)
    plt.xticks(rotation=30)
    plt.show()
    print(f'Top converters in {col}:', conv.head(3).round(3).to_dict(), '\n')
```

- **Purpose:** This cell performs a bivariate analysis on the categorical features, examining their relationship with the `purchase` target variable. It calculates and visualizes the conversion rate (the average purchase) for each category.
- **Code Line by Line:**
  - `cat_list = [...]`: Defines a list of categorical columns to analyze.
  - `for col in cat_list:`: Loops through each categorical column.
  - `if col in df.columns and 'purchase' in df.columns:`: Checks for the existence of both columns.
  - `conv = df.groupby(col)['purchase'].mean().sort_values(ascending=False)`:
    - `df.groupby(col)`: This groups the DataFrame's rows by the unique values in the current categorical column.
    - `['purchase'].mean()`: For each group, it calculates the mean of the `purchase` column. Since `purchase` is 0 or 1, this gives the proportion of purchases, which is the conversion rate.
    - `.sort_values(ascending=False)`: This sorts the conversion rates from highest to lowest.
  - `plt.figure(figsize=(6,3))`: Creates a new plot figure.
  - `plt.bar(conv.index.astype(str), conv.values)`: Creates a bar chart showing the conversion rate for each category.
  - `plt.title(...)`: Sets the title.
  - `plt.xticks(rotation=30)`: Rotates the x-axis labels to prevent overlap.
  - `plt.show()`: Displays the chart.

- `print(...)`: Prints the top 3 categories with the highest conversion rates, providing a clear summary.

- **Workshop Use Case:** This is where you can connect the analysis directly to business strategy. A high conversion rate for a specific traffic source or a particular product category can inform marketing budgets or product development decisions.

Section 12. Multivariate — Correlation matrix

Code:

Python

```
# Numeric correlations
num = df.select_dtypes(include=[np.number]).copy()
corr = num.corr()
plt.figure(figsize=(7,6))
plt.imshow(corr, cmap='coolwarm', vmin=-1, vmax=1)
plt.colorbar()
plt.xticks(range(len(corr.columns)), corr.columns, rotation=90)
plt.yticks(range(len(corr.columns)), corr.columns)
plt.title('Correlation matrix')
plt.tight_layout()
plt.show()

# Print top correlated pairs (absolute value)
pairs = corr.abs().unstack().sort_values(ascending=False)
pairs = pairs[pairs < 0.9999]  # remove self-correlation
top = pairs.drop_duplicates().head(8)
display(top.reset_index().rename(columns={'level_0':'f1','level_1':'f2',0:'abs_co
rr'}))
```

- **Purpose:** This cell performs a multivariate analysis by examining the relationships between all pairs of numeric features at once. A correlation matrix and its visual representation (a heatmap) help to identify multicollinearity, where features are highly correlated with each other.

- **Code Line by Line:**
  - `num = df.select_dtypes(...)`: Creates a new DataFrame with only numeric columns. `.copy()` is used to avoid a `SettingWithCopyWarning` in Pandas.
  - `corr = num.corr()`: This method calculates the pairwise correlation coefficients between all numeric columns. The result is a square matrix where the value at row `i`, column `j` is the correlation between column `i` and column `j`.
  - `plt.figure(figsize=(7,6))`: Creates a new plot figure.
  - `plt.imshow(corr, ...)`: This function creates a heatmap from the correlation matrix.
    - `cmap='coolwarm'`: Sets the color map, which provides a clear gradient from negative (blue) to positive (red) correlations.
    - `vmin=-1, vmax=1`: Sets the minimum and maximum values for the color scale, ensuring a consistent range from perfect negative correlation to perfect positive correlation.
  - `plt.colorbar()`: Adds a color bar to the side of the plot to show what each color represents.
  - `plt.xticks(...)` and `plt.yticks(...)`: These lines set the labels for the x and y axes to be the column names, rotating the x-axis labels for readability.
  - `plt.title(...)` and `plt.tight_layout()`: Sets the plot title and adjusts plot parameters to give a tight layout.
  - `# Print top correlated pairs...`: A comment for the next code block.
  - `pairs = corr.abs().unstack().sort_values(...)`:
    - `corr.abs()`: Takes the absolute value of the correlation matrix, as we are interested in the strength of the relationship, not the direction.
    - `.unstack()`: Converts the matrix into a Series of `(feature1, feature2)` pairs with their correlation as the value.
    - `.sort_values(ascending=False)`: Sorts the Series from highest to lowest correlation.

- `pairs = pairs[pairs < 0.9999]`: This removes the perfect self-correlation of `1.0` (each feature with itself).
- `top = pairs.drop_duplicates().head(8)`: This removes duplicate pairs (e.g., `(A, B)` and `(B, A)`) and selects the top 8 most correlated pairs.
- `display(top.reset_index()...)`: Displays the final list in a nice, clean table.

- **Workshop Use Case:** Explain the concept of multicollinearity. Discuss how a high correlation between two features (e.g., `pages_viewed` and `clicks`) means they are providing redundant information to a model. In some cases, you might choose to drop one of the features to simplify the model.

Section 13. Pairwise scatter (sample)

Code:

Python

```
sample_cols = ['time_spent_minutes','pages_viewed','avg_session_value','enga
gement_score']
available = [c for c in sample_cols if c in df.columns]
if len(available) >= 2:
    sample = df[available].sample(n=min(200, len(df)), random_state=2)
    pd.plotting.scatter_matrix(sample, diagonal='hist', figsize=(8,8))
    plt.suptitle('Scatter matrix (sample)')
    plt.show()
else:
    print('Not enough columns for scatter matrix.')
```

- **Purpose:** This cell provides a visual complement to the correlation matrix. A scatter plot matrix shows the relationships between multiple numeric features, making it easy to spot clusters, linear relationships, or other patterns that are not obvious from a correlation value alone.
- **Code Line by Line:**
  - `sample_cols = [...]`: A list of numeric columns to be included in the scatter matrix.

- `available = [...]`: A list comprehension that filters the `sample_cols` list to only include columns that are actually present in the DataFrame.
- `if len(available) >= 2:`: The `scatter_matrix` function requires at least two columns, so this conditional check prevents an error.
- `sample = df[available].sample(...)`:
  - `df[available]`: Selects the columns that are available.
  - `.sample()`: Takes a random sample of rows from the DataFrame. This is important for large datasets to keep the plot from being overcrowded and unreadable.
  - `n=min(200, len(df))`: Takes a sample of at most 200 rows, or fewer if the DataFrame has fewer than 200 rows.
  - `random_state=2`: Sets a seed for the random sampling, ensuring the same rows are selected each time for reproducibility.
- `pd.plotting.scatter_matrix(...)`:
  - `diagonal='hist'`: This is a useful argument that places a histogram for each variable on the diagonal of the matrix, showing its distribution.
  - `figsize=(8,8)`: Sets the size of the overall plot.
- `plt.suptitle(...)`: Adds a title to the entire figure.
- `plt.show()`: Displays the plot.
- `else: print(...)`: A message is printed if there are not enough columns to create the plot.

- **Workshop Use Case:** Explain to students how to visually interpret the scatter plots. For example, a tight cluster could indicate a specific user segment, while a linear pattern confirms a strong correlation seen in the previous step.

Section 14. Segmented analysis — pivot & grouped view

Code:

Python

```
# Pivot: membership_status x product_category conversion
if 'membership_status' in df.columns and 'product_category' in df.columns:
    pivot = df.pivot_table(index='membership_status', columns='product_categ
ory', values='purchase', aggfunc='mean')
    display(pivot.round(3))
    # Plot membership conversion overall
    mem = df.groupby('membership_status')['purchase'].mean()
    plt.figure(figsize=(5,3))
    plt.bar(mem.index, mem.values)
    plt.title('Conversion by membership_status')
    plt.show()
```

- **Purpose:** This cell demonstrates a more granular level of analysis by creating a pivot table. This allows for a deeper dive into how different combinations of categorical variables (e.g., membership status and product category) affect the conversion rate.
- **Code Line by Line:**
  - `# Pivot: membership_status x product_category conversion`: A comment explaining the purpose.
  - `if 'membership_status' in df.columns...`: Checks for the existence of the necessary columns.
  - `pivot = df.pivot_table(...)`: This is the function for creating a pivot table.
    - `index='membership_status'`: The `membership_status` column will form the rows of the new table.
    - `columns='product_category'`: The `product_category` column will form the columns of the new table.
    - `values='purchase'`: The values in the table will be based on the `purchase` column.
    - `aggfunc='mean'`: The aggregation function is the mean, so each cell in the table will show the average purchase (i.e., the conversion rate) for that specific combination of membership status and product category.

- `display(pivot.round(3))`: Displays the pivot table, rounded to three decimal places for clarity.
- `# Plot membership conversion overall`: A comment for the second part of the cell.
- `mem = df.groupby('membership_status')['purchase'].mean()`: This is similar to the code in Section 11, but it groups by only `membership_status` to get the overall conversion rate for each membership type.
- `plt.figure(...)`, `plt.bar(...)`, `plt.title(...)`, `plt.show()`: These lines create and display a simple bar chart of the overall conversion rate by membership status.

- **Workshop Use Case:** This is where you can generate direct, actionable business insights. For example, you might discover that while "Gold" members have a high overall conversion rate, they convert very poorly on "Electronics" products. This could lead to a hypothesis that the product category is not appealing to that segment.

Section 15. Derived metrics — check usefulness

Code:

Python

```
if 'engagement_score' in df.columns and 'purchase' in df.columns:
    plt.figure(figsize=(6,3))
    plt.boxplot([df[df['purchase']==0]['engagement_score'], df[df['purchase']==1]['engagement_score']], labels=['No','Yes'])
    plt.title('Engagement score by purchase')
    plt.show()
    print('engagement_score medians →', df.groupby('purchase')['engagement_score'].median().to_dict())

if 'cart_to_wishlist_ratio' in df.columns and 'purchase' in df.columns:
    plt.figure(figsize=(6,3))
    plt.boxplot([df[df['purchase']==0]['cart_to_wishlist_ratio'], df[df['purchase']==1]['cart_to_wishlist_ratio']], labels=['No','Yes'])
```

```
plt.title('Cart-to-wishlist ratio by purchase')
plt.show()
```

- **Purpose:** This cell specifically evaluates the effectiveness of the new features created in Section 6. The goal is to see if the `engagement_score` and `cart_to_wishlist_ratio` are useful for separating purchasers from non-purchasers.
- **Code Line by Line:**
  - `if 'engagement_score' in df.columns...`: Checks if both the derived feature and the target variable exist.
  - `plt.boxplot(...)`: As in Section 10, this creates a side-by-side boxplot comparing the distribution of the `engagement_score` for the "No" and "Yes" purchase groups.
  - `plt.title(...)`: Sets the title of the boxplot.
  - `plt.show()`: Displays the plot.
  - `print(...)`: Prints the median engagement scores for both groups, providing a clear numerical comparison.
  - The second `if` block repeats this exact process for the `cart_to_wishlist_ratio`.
- **Workshop Use Case:** Explain to students that this step is crucial for validating feature engineering efforts. If the boxplots show a clear separation between the two groups, it confirms that the newly created features are valuable and should be included in a machine learning model.

Section 16. Time-series — daily & hourly patterns

Code:

Python

```
if 'date' in df.columns and 'purchase' in df.columns:
    df['date_only'] = df['date'].dt.date
    daily = df.groupby('date_only').agg(total=('purchase','count'), purchases=
('purchase','sum'))
    daily['rate'] = daily['purchases'] / daily['total']
```

```
    plt.figure(figsize=(9,3))
    plt.plot(daily.index, daily['rate'], marker='o')
    plt.title('Daily purchase rate')
    plt.xticks(rotation=45)
    plt.show()

    # Hourly conversion
    hourly = df.groupby('hour')['purchase'].mean()
    plt.figure(figsize=(7,3))
    plt.plot(hourly.index, hourly.values, marker='o')
    plt.title('Hourly purchase rate')
    plt.show()
else:
    print('Missing date or purchase column.')
```

- **Purpose:** This cell analyzes the dataset's time-based patterns. It reveals how the purchase rate changes on a daily basis and across different hours of the day. These insights are useful for business operations and scheduling marketing campaigns.
- **Code Line by Line:**
  - `if 'date' in df.columns...`: Checks that both `date` and `purchase` columns exist.
  - `df['date_only'] = df['date'].dt.date`: Creates a new column that contains only the date part of the timestamp, removing the time component.
  - `daily = df.groupby('date_only').agg(...)`:
    - `df.groupby('date_only')`: Groups the data by each unique date.
    - `.agg(...)`: This method allows for multiple aggregations at once. It counts the total number of sessions (`total`) and sums the number of purchases (`purchases`) for each day.
  - `daily['rate'] = daily['purchases'] / daily['total']`: Calculates the daily purchase rate by dividing the number of purchases by the total number of sessions.

    - `plt.figure(...)`, `plt.plot(...)`, `plt.title(...)`, `plt.xticks(...)`, `plt.show()`: These lines create and display a line plot showing the daily purchase rate over time. The markers and title make the plot easy to read.
    - `# Hourly conversion`: A comment for the next plot.
    - `hourly = df.groupby('hour')['purchase'].mean()`: This groups the DataFrame by the `hour` column and calculates the average `purchase` rate for each hour.
    - `plt.figure(...)`, `plt.plot(...)`, `plt.title(...)`, `plt.show()`: These lines create and display a line plot showing the purchase rate throughout the day.
- **Workshop Use Case:** Explain the practical applications of this analysis. For example, the daily plot can show weekly trends or seasonal variations. The hourly plot can pinpoint peak conversion hours, which is perfect for timing promotional emails or running targeted advertisements.

Section 17. Summary & next steps

- **Purpose:** This final markdown cell serves as a conclusion to the exploratory data analysis (EDA). It summarizes the key insights gained and provides a clear roadmap for the next stages of a data science project.
- **Content:**
    - **Summary:** The summary highlights the most important findings from the analysis, such as the key signals for predicting a purchase (e.g., engagement, pages viewed, cart behavior, and membership status). This is a concise overview of what has been learned about the data.
    - **Next steps:** This section outlines the future work required to build a functional model. It includes building a preprocessing pipeline, training baseline models, evaluating them with business-driven metrics, and finally, deploying a prediction API.
- **Workshop Use Case:** Conclude the workshop by reinforcing the value of EDA. Explain that EDA is not an end in itself but a critical process that informs every subsequent step, from feature engineering to model selection and evaluation. The "next steps" provide a clear path forward, showing that this analysis is part of a larger, structured data science workflow.