Exploratory Data Analysis (EDA) in `pandas` Part 1[1]

Overview

In this handout, we will explore the basics of `Exploratory Data Analysis (EDA)`, a critical step in understanding the underlying patterns, trends, and structure within a dataset before diving into deeper analysis or modeling. EDA provides an opportunity to uncover insights, detect anomalies, and identify relationships within data. Through these methods, we can approach data more strategically, guiding the process of data cleaning, transformation, and feature engineering.

1 Introduction

In this handout, we will use a fictional gaming dataset to perform EDA, covering basic steps such as data inspection, handling missing values[2], and exploring summary statistics. This dataset captures various details about user activity and preferences in a gaming context. Here is the dataset:

Creating the Dataset

```
import pandas as pd
pd.options.display.precision = 1

game_data = {
    "User_ID": [101, 102, 103, 104, 105, 106, 107],
    "Age": [23, 35, 45, 30, 25, 28, 33],
    "Gender": ["F", "M", "M", "F", "F", "M", "F"],
    "Country": ["US", "CA", "US", "CA", "US", "US", "US"],
    "Game_Score": [88, 92, 78, 85, None, 95, 82],
    "Sessions": [5, 8, 6, 9, 3, 7, 4],
    "Device": ["Mobile", "Desktop", "Tablet", "Mobile", "Mobile", "Desktop", "Mobile"],
    "App_Rating": [5.0, None, 3.8, 4.2, 4.0, 4.7, 4.1],
    "Game_Type": [
        ["Candy Crush", "Warzone"],
        ["Fortnite", "Elden Ring"],
```

[1] Visualization is an important part of EDA and we will cover that in the follow-up handouts
[2] There will be another handout covering this topic in more details

```
        ["Super Mario", "Elden Ring"],
        ["Minecraft", "Among Us"],
        ["Roblox", "Call of Duty"],
        ["Candy Crush", "PUBG"],
        ["Among Us", "Candy Crush"]
    ]
}

game_df = pd.DataFrame(game_data)
display(game_df)
```

This will yield the following `DataFrame`:

	User_ID	Age	Gender	Country	Game_Score	Sessions	Device	App_Rating	Game_Type
0	101	23	F	US	88.0	5	Mobile	5.0	[Candy Crush, Warzone]
1	102	35	M	CA	92.0	8	Desktop	NaN	[Fortnite, Elden Ring]
2	103	45	M	US	78.0	6	Tablet	3.8	[Super Mario, Elden Ring]
3	104	30	F	CA	85.0	9	Mobile	4.2	[Minecraft, Among Us]
4	105	25	F	US	NaN	3	Mobile	4.0	[Roblox, Call of Duty]
5	106	28	M	US	95.0	7	Desktop	4.7	[Candy Crush, PUBG]
6	107	33	F	US	82.0	4	Mobile	4.1	[Among Us, Candy Crush]

In this `DataFrame`:

- `User_ID`: Unique identifier for each user.
- `Age`: Age of each user.
- `Gender`: Gender of the user, represented – in this very limited dataset – by two categories: `M` and `F`.
- `Country`: Country where each user is located.
- `Game_Score`: Overall score achieved by each user in the game.
- `Sessions`: Number of gaming sessions each user completed.
- `Device`: Type of device used by the user (e.g., `Mobile`, `Desktop`, `Tablet`).
- `App_Rating`: Rating given by each user to the gaming app.
- `Game_Type`: A `list` containing the genres of games each user has played. [3]

This dataset provides a mixture of numerical, categorical, and list-like data types, offering an ideal setting to practice and learn different data handling and analysis techniques in `pandas`.

2 Basic Data Overview

Before getting into detailed data analysis, it's important to understand the structure and completeness of the dataset. This section will explore some basic `pandas` functions to inspect and summarize the data.

[3] This column is unique because each entry contains a `list` of values instead of a single value. In this handout, we will explore methods to work effectively with this type of column.

2.1 Basic Inspection of Data

To begin, it is a good idea to look at a few data points. We can start by the first few rows using `head()` method. This helps us get a quick understanding of dataset structure and contents.

Displaying First Few Rows

```
game_df.head()
```

yields

	User_ID	Age	Gender	Country	Game_Score	Sessions	Device	App_Rating	Game_Type
0	101	23	F	US	88.0	5	Mobile	5.0	[Candy Crush, Warzone]
1	102	35	M	CA	92.0	8	Desktop	NaN	[Fortnite, Elden Ring]
2	103	45	M	US	78.0	6	Tablet	3.8	[Super Mario, Elden Ring]
3	104	30	F	CA	85.0	9	Mobile	4.2	[Minecraft, Among Us]
4	105	25	F	US	NaN	3	Mobile	4.0	[Roblox, Call of Duty]

Using `.head()` displays the first few rows of the `DataFrame` providing a glimpse of the initial entries in our dataset.

Common initial inspection methods are:

- `.head()`: By default, `.head()` shows the first 5 rows, but it also accepts an optional integer argument to specify the number of rows to display, such as `game_df.head(2)` to view the first 2 rows.
- `.tail()`: `.tail()` displays the last few rows of the `DataFrame`, allowing you to inspect the final entries in the dataset. By default `.tail()`, shows the 5 rows and also accepts an optional argument to control the number of rows shown.
- `.sample()`: `.sample()` returns a random selection of rows from the `DataFrame`, providing a snapshot of the data. By default, `.sample()` displays a single random row, but it also accepts an optional integer argument to specify the number of rows to retrieve. For example, `.sample(3)` will display 3 randomly selected rows.

Randomly Sampling Rows

```
game_df.sample(3)
```

This may yield an output like the following:

	User_ID	Age	Gender	Country	Game_Score	Sessions	Device	App_Rating	Game_Type
1	102	35	M	CA	92.0	8	Desktop	NaN	[Fortnite, Elden Ring]
5	106	28	M	US	95.0	7	Desktop	4.7	[Candy Crush, PUBG]
4	105	25	F	US	NaN	3	Mobile	4.0	[Roblox, Call of Duty]

`head()`, `tail()`, `sample()` Methods

The `head()` method is used to display the first few rows of a DataFrame.
The `tail()` method is used to display the last few rows of a DataFrame.
The `sample()` method randomly selects rows from a DataFrame.

2.2 Data Overview

The `.info()` function offers a concise summary, showing column names, non-null counts, and data types. This summary helps us quickly identify potential data types and missing values.

Data Overview

```
game_df.info()
```

gives

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 7 entries, 0 to 6
Data columns (total 9 columns):
 #   Column       Non-Null Count  Dtype
 0   User_ID      7 non-null      int64
 1   Age          7 non-null      int64
 2   Gender       7 non-null      object
 3   Country      7 non-null      object
 4   Game_Score   6 non-null      float64
 5   Sessions     7 non-null      int64
 6   Device       7 non-null      object
 7   App_Rating   6 non-null      float64
 8   Game_Type    7 non-null      object
dtypes: float64(2), int64(3), object(4)
memory usage: 632.0 bytes
```

The output of `game_df.info()` provides an overview of the structure and contents of the `DataFrame`. Here's what each section represents:

- **DataFrame Type**: The line `<class 'pandas.core.frame.DataFrame'>` confirms that this is a `DataFrame` object.
- **RangeIndex**: `RangeIndex: 7 entries, 0 to 6` shows that there are 7 rows in this dataset, with indices ranging from 0 to 6.
- **Data Columns**: `Data columns (total 9 columns)` indicates that there are 9 columns in this `DataFrame`.
- **Column Details**: The following section lists each column, with four key pieces of information:
  - **#**: The index number of each column.
  - **Column**: The column name.
  - **Non-Null Count**: This shows the count of non-missing values in each column. For example, `Game_Score` and `App_Rating` have 6 non-null values, indicating one missing value each.
  - **Dtype**: The data type for each column:
    - `int64`: Indicates integer values, used for columns like `User_ID`, `Age`, and `Sessions`.
    - `float64`: Indicates floating-point (decimal) numbers, used for columns like `Game_Score` and `App_Rating`.
    - `object`: Usually indicates text (string) data, but it can also include mixed data types or even lists. Here, `Gender`, `Country`, `Device`, and `Game_Type` are categorized as `object`.
- **Data Type Summary**: `dtypes: float64(2), int64(3), object(4)` provides a summary of data types

across the `DataFrame`, showing there are 2 columns with floating-point numbers, 3 with integers, and 4 with objects.

- **Memory Usage**: Finally, `memory usage: 632.0 bytes` indicates the memory footprint of the `DataFrame`, which can be helpful in optimizing performance with larger datasets.

This overview helps us understand the dataset structure, spot missing values, and identify the types of data present, which are all critical steps in preparing for analysis.

`info()` Method

The `info()` method provides a summary of the DataFrame, including column names, data types, non-null counts, and memory usage. It's useful for quickly checking the dataset's structure.

2.3 Summary Statistics

We usually look at summary statistics as a next step in overviewing the data. The `.describe()` function provides essential summary statistics for numerical columns, such as `mean`, `standard deviation`, `minimum`, and `quartile` values.

Summary Statistics using `describe()`

```
game_df.describe()
```

yields

	User_ID	Age	Game_Score	Sessions	App_Rating
count	7.0	7.0	6.0	7.0	6.0
mean	104.0	31.3	86.7	6.0	4.3
std	2.16	7.6	6.5	2.2	0.41
min	101.0	23.0	78.0	3.0	3.8
25%	102.5	25.0	82.0	4.0	4.0
50%	104.0	30.0	86.5	6.0	4.15
75%	105.5	35.0	92.0	7.0	4.7
max	107.0	45.0	95.0	9.0	5.0

Note: We can also try `game_df.describe(include='all')` to extend the summary to categorical columns as well.

Summary Statistics Including All Columns

```
game_df.describe(include='all')
```

	User_ID	Age	Gender	Country	Game_Score	Sessions	Device	App_Rating	Game_Type
count	7.0	7.0	7	7	6.0	7.0	7	6.0	7
unique	NaN	NaN	2	2	NaN	NaN	3	NaN	7
top	NaN	NaN	F	US	NaN	NaN	Mobile	NaN	[Candy Crush, Warzone]
freq	NaN	NaN	4	5	NaN	NaN	4	NaN	1
mean	104.0	31.3	NaN	NaN	86.7	6.0	NaN	4.3	NaN
std	2.16	7.6	NaN	NaN	6.5	2.2	NaN	0.41	NaN
min	101.0	23.0	NaN	NaN	78.0	3.0	NaN	3.8	NaN
25%	102.5	25.0	NaN	NaN	82.0	4.0	NaN	4.0	NaN
50%	104.0	30.0	NaN	NaN	86.5	6.0	NaN	4.15	NaN
75%	105.5	35.0	NaN	NaN	92.0	7.0	NaN	4.7	NaN
max	107.0	45.0	NaN	NaN	95.0	9.0	NaN	5.0	NaN

As you can see, for categorical columns such as `Gender`, `Country`, `Device`, and `Game_Type`, `describe()` provides:

- **`count`**: The total number of `non-null` entries in each column.
- **`unique`**: The number of unique values in the column. For example, `Gender` has 2 unique values in this limited small dataset (`M` and `F`), while `Device` has 3 unique values (e.g., `Mobile, Desktop, Tablet`).
- **`top`**: The most frequent value (or mode) in the column. For instance, in the `Country` column, `US` is the most common value.
- **`freq`**: The frequency of the most common value in the column. For `Country`, `US` appears 5 times, making it the most frequent entry.

For numerical columns, the provided statistics is self-explanatory.

This statistical summary is useful for understanding both the central tendency and variability of numerical data, as well as the frequency and distribution of categorical data.

`describe()` Method

The `describe()` method generates summary statistics for numerical columns, including count, mean, min, max, and quartiles. It helps in understanding the dataset's distribution.

2.4 Checking Data Completeness

Real-world datasets often contain missing values, which `pandas` represents as `NaN` (`Not a Number`). Detecting missing values early is essential for ensuring data quality. The `.isnull()` function allows us to identify missing values by returning a `DataFrame` of the same shape, where each cell is `True` if the value is missing and `False` otherwise.

Identifying Missing Values with `.isnull()`

```
game_df.isnull()
```

This produces the following output:

	User_ID	Age	Gender	Country	Game_Score	Sessions	Device	App_Rating	Game_Type
0	False	False	False	False	False	False	False	False	False
1	False	False	False	False	False	False	False	False	False
2	False	False	False	False	True	False	False	True	False
3	False	False	False	False	False	False	False	False	False
4	False	False	False	False	False	False	False	False	False
5	False	False	False	False	False	False	False	False	False
6	False	False	False	False	False	False	False	False	False

A quick visual inspection shows two `True` values in this `DataFrame`, indicating missing entries: one in the `Game_Score` column and another in the `App_Rating` column.

To obtain a summary of missing values across each column, we can use `.isnull().sum()`, which counts the total number of missing values in every column.

Summing Missing Values by Column

```
game_df.isnull().sum()
```

The output is as follows:

Column	Missing Values
User_ID	0
Age	0
Gender	0
Country	0
Game_Score	1
Sessions	0
Device	0
App_Rating	1
Game_Type	0

Finally, to find the total number of missing values across the entire `DataFrame`, we can use:

Total Missing Values in DataFrame

```
game_df.isnull().sum().sum()
```

This returns:

```
2
```

In this example, there are a total of 2 missing values in the `game_df` DataFrame.

3 Handling Missing Values

Missing values are a common challenge in real-world datasets and can significantly impact the results of data analysis. Addressing missing values is an essential step in exploratory data analysis (EDA). Strategies for handling missing values, including techniques like imputation, removal, and analysis of patterns, will be discussed comprehensively in another handout.

To guide our exploration and provide a practical context for the rest of this handout, I'd like to pose a series of questions. These questions will help us focus on the functions and techniques we'll cover and demonstrate their applications in meaningful ways

Exploratory Data Analysis Questions

1. **Which columns contain categorical data, and which contain numerical data?**
   - ***Purpose***: To help us understand the data types and filtering by data types.

2. **What are the unique categories in each categorical column (e.g., `Country`, `Device`, `Gender`)?**
   - ***Purpose***: This helps us identify categorical columns and understand the diversity in data values.

3. **What is the most common `Device` used by players, and how is `Gender` distributed across the dataset?**
   - ***Purpose***: To help us look into frequency distributions within categorical columns.

4. **How many players from each `Country` use a `Mobile` device?**
   - ***Purpose***: This will help us look into frequency distribution of more than one variable.

5. **What is the average `Game_Score` for each `Device` type, and how does this vary by `Country`?**
   - ***Purpose***: This will help us learn value aggregation by category.

6. **What are the average `Game_Score` and `App_Rating` for female players?**
   - ***Purpose***: To demonstrate how to retrieve specific groups within a grouped `DataFrame`.

7. **How many players have each combination of `Device` and `Gender`? What is the relative frequency of each combination?**
   - ***Purpose***: To explore relationships between multiple categorical variables.

8. **What is the average `Game_Score` by `Country` and `Device` type?**
   - ***Purpose***: To summarize continuous data by multiple categorical variables.

9. **What are the median `App_Rating` values for each `Game_Type`, separated by `Gender`?**
   - ***Purpose***: To analyze data across categorical and numerical dimensions.

10. **How many players of each `Gender` from each `Country` have played games with an `App_Rating` above 4.0?**
    - ***Purpose***: To explore categorical and numerical relationships.

11. **What is the correlation between `Game_Score` and `Sessions`? How about `Game_Score` and `App_Rating`?**
    - ***Purpose***: To introduce `corr()` for calculating the strength and direction of relationships between numerical columns.

4 Data Transformation

In data analysis, transforming data types and structures is often necessary to ensure consistency and prepare for various analyses. This section introduces several important transformation techniques in `pandas`:

- `astype()` for type conversion
- `explode()` for working with list-like data in columns
- `map()` and `apply()` for applying functions to data.

Besides the above functions, there are other methods and accessors such as `.dt` and `.str` that we will discuss later.

4.1 Type Conversion with `astype`

Real-world datasets often have mixed or unexpected types, which can interfere with data processing. The `astype` method in `pandas` is commonly used for converting columns to desired data types. This is especially useful in cases where:

- **Numeric calculations** require integer or float data types.
- **Categorical variables** should be in the `category` data type for optimal memory usage and efficient encoding.

Type conversion ensures data consistency and facilitates analysis. Although the current data types in `game_df` align with expectations, the following examples demonstrate how `astype` can be used effectively:

Applying Type Conversion with `astype`

```
## converting Age and Sessions to integers
game_df["Age"] = game_df["Age"].astype(int)
game_df["Sessions"] = game_df["Sessions"].astype(int)

## converting Gender to a categorical type
game_df["Gender"] = game_df["Gender"].astype("category")

## ensuring Game_Score is float
game_df["Game_Score"] = game_df["Game_Score"].astype(float)
```

`astype()` Method

The `astype()` method is used to convert a column's data type, ensuring consistency and enabling appropriate operations like calculations or memory optimization.

4.2 Selecting Columns by Data Type with `select_dtypes`

The `select_dtypes` function in `pandas` is a powerful tool for selecting columns based on their data types. This is particularly useful for exploring or performing operations on subsets of data with specific types. For instance:

- Select only numerical columns for statistical analysis.
- Select categorical columns for encoding or transformation.

Here is an example of using `select_dtypes` with the `game_df` dataset:

Using `select_dtypes` to Identify Numerical and Categorical Columns

```
## Selecting numerical columns
print('numerical columns')
numerical_columns = game_df.select_dtypes(include=["int64", "float64"])
display(numerical_columns.head())

## Selecting categorical columns
print('categorical columns')
categorical_columns = game_df.select_dtypes(include=["category", "object"])
display(categorical_columns.head())
```

This yields

numerical columns

User_ID	Age	Game_Score	Sessions	App_Rating
101	23	88.0	5	5.0
102	35	92.0	8	NaN
103	45	78.0	6	3.8
104	30	85.0	9	4.2
105	25	NaN	3	4.0

categorical columns

Gender	Country	Device
F	US	Mobile
M	CA	Desktop
M	US	Tablet
F	CA	Mobile
F	US	Mobile

`select_dtypes()` Method

The `select_dtypes()` method is used to filter DataFrame columns based on their data types, such as selecting only numerical or categorical columns.

4.3 Expanding List-Like Data with `explode`

The `Game_Type` column in `game_df` contains lists of game titles, meaning each row can contain multiple values. This list-like data structure can be challenging to analyze as is. The `explode` function in `pandas` is useful for expanding these lists so that each game title appears in its own row, making it easier to analyze individual game types. Here is an example:

Expanding List Values and Analyzing Popular Games

```
game_df_expanded = game_df.explode("Game_Type")
display(game_df_expanded)
```

This will yield the following long `DataFrame`:

User_ID	Age	Gender	Country	Game_Score	Sessions	Device	App_Rating	Game_Type
101	23	F	US	88.0	5	Mobile	5.0	Candy Crush
101	23	F	US	88.0	5	Mobile	5.0	Warzone
102	35	M	CA	92.0	8	Desktop	NaN	Fortnite
102	35	M	CA	92.0	8	Desktop	NaN	Elden Ring
103	45	M	US	78.0	6	Tablet	3.8	Super Mario
103	45	M	US	78.0	6	Tablet	3.8	Elden Ring
104	30	F	CA	85.0	9	Mobile	4.2	Minecraft
104	30	F	CA	85.0	9	Mobile	4.2	Among Us
105	25	F	US	NaN	3	Mobile	4.0	Roblox
105	25	F	US	NaN	3	Mobile	4.0	Call of Duty
106	28	M	US	95.0	7	Desktop	4.7	Candy Crush
106	28	M	US	95.0	7	Desktop	4.7	PUBG
107	33	F	US	82.0	4	Mobile	4.1	Among Us
107	33	F	US	82.0	4	Mobile	4.1	Candy Crush

Interpretation: After using `explode`, each game in `Game_Type` appears as a separate row, with other column values repeated as necessary. By using `value_counts` on the `Game_Type` column in this expanded format, we can easily see which games are played most frequently by users, per below code:

Analyzing Popular Games

```
popular_games = game_df_expanded["Game_Type"].value_counts()
display(popular_games)
```

yielding

Game_Type	Count
Candy Crush	3
Among Us	2
Elden Ring	2
Warzone	1
Fortnite	1
Super Mario	1
Minecraft	1
Roblox	1
Call of Duty	1
PUBG	1

By using `value_counts` on the `Game_Type` column in this expanded format, we can easily see which games are played most frequently by users.

`explode()` Method

The `explode()` method transforms list-like elements in a column into separate rows, making it easier to analyze or process individual elements.

4.4 `map` Function

The `map` and `apply` methods allow for element-wise transformations and more complex row or column operations. This section focuses on `map`; which is best suited for element-wise transformations in a single column.

Assume we want to categorize `App_Rating` into two groups: `"Low"` and `"High"`. To achieve this, we first define a function and then apply it using the `map` method, as shown in the code below.

Using map to Categorize Ratings

```
def categorize_rating(rating):
    """
    a function to categorize the app rating
    with a threshold of 4
    """
    if rating >= 4:
        return "High"
    else:
        return "Low"

# Apply the function using the 'map' method
game_df["Rating_Category"] = game_df["App_Rating"].map(categorize_rating)

# Display the updated DataFrame with App Rating and Rating Category
display(game_df[["App_Rating", "Rating_Category"]])
```

The following is a partial view of the updated `game_df`, showing the `App_Rating` and `Rating_Category` columns. Due to space limitations, only a subset of columns is displayed.

Index	App_Rating	Rating_Category
0	5.0	High
1	NaN	NaN
2	3.8	Low
3	4.2	High
4	4.0	High
5	4.7	High
6	4.1	High

Note that, as an alternative to defining a separate function (e.g., `categorize_rating()`), you can achieve the same result by using a concise `lambda` function.

You might wonder why one would convert a perfectly valid numerical column into categories. Here's why:

`map()` Method

The `map()` method applies a function, dictionary, or Series to transform each element in a pandas Series, allowing flexible and efficient element-wise operations.

4.5 Why Convert Numerical Values to Categories?

Converting numerical values into categories is a common practice in data analysis and modeling for several reasons:

- **Simplifying Analysis:** Categorizing numerical data can simplify complex datasets, making trends and patterns more apparent, especially in exploratory data analysis. For instance, dividing income levels into `"Low"`, `"Medium"`, and `"High"` categories helps to quickly compare groups.

- **Enabling Group-Based Comparisons:** Categories allow for easy aggregation and comparison across groups. For example, creating `"Age Groups"` (e.g., `"Under 18"`, `"18-35"`, `"35+"`) can facilitate analyzing behavioral patterns or trends within specific demographic segments.

- **Handling Non-Linear Relationships:** In some cases, the relationship between a numerical variable and the target outcome may not be linear or proportional across the variable's range. For instance, in a gaming context, a `Game_Score` of 95 and 85 may have a similar impact on predicting user retention, but a score of 50 might represent a drastically different user behavior. By categorizing such variables into meaningful groups (e.g., `"High"`, `"Medium"`, and `"Low"`), we can simplify the modeling process while capturing potential non-linear effects more effectively. Categorizing temperature to `"Cold"`, `"Moderate"`, and `"Hot"`) is another example.

- **Enhancing Model Interpretability:** In predictive models, such as most models discussed in your Machine

Learning course, categorical variables can make results more interpretable. For example, a `"High"` or `"Low"` rating is easier to explain to stakeholders than a continuous score like `4.3` or `3.8`.

- **Reducing Noise:** Numerical data often contains minor variations or fluctuations that may not hold significant meaning for the analysis. For example, slight differences in `App_Rating` scores such as 4.1 and 4.2 might not reflect a meaningful distinction in user satisfaction. By categorizing the ratings into broader groups like `"High"` and `"Low"`, we can reduce the impact of these small, potentially noisy variations, thereby focusing on the broader trends and patterns that are more relevant to the analysis.

- **Matching Specific Modeling Techniques:** Certain algorithms or methods, such as decision trees, can work more effectively with categorical variables for splitting the data into meaningful groups.

we will finish this subsection by showing an application of `apply` function for working

4.6 `apply` Function

The `apply` function is highly versatile and allows for operations involving multiple columns in a dataset. In the below example, we calculate an **Engagement Score** for each user in the `game_df` dataset. The Engagement Score is defined as:

$$\text{Engagement Score} = \frac{\text{Game Score} \times \text{App Rating}}{\text{Sessions}}$$

If `Sessions` is 0, the score is set to `0` to avoid division by zero.

This score evaluates a user's overall engagement with the app by combining their `Game_Score`, `App_Rating`, and `Sessions`; here is how you can do that using `apply()` function:

Calculating Engagement Score Using `apply`

```
def calculate_engagement_score(row):
    """
    Calculate engagement score based on Game_Score, App_Rating, and Sessions.

    The engagement score is calculated as:
        (Game_Score * App_Rating) / Sessions

    If Sessions is 0, the score is set to 0 to avoid division by zero.

    Parameters:
    row (pandas.Series): A single row of the DataFrame. Each row contains
    the values of all columns for that specific observation, allowing us
    to access the "Game_Score", "App_Rating", and "Sessions" values directly.

    Returns:
    float: The engagement score for the given row.
    """
```

```
    if row["Sessions"] > 0:
        return (row["Game_Score"] * row["App_Rating"]) / row["Sessions"]
    else:
        return 0

# Apply the function row-wise to calculate the Engagement Score
# We have axis = 1 indicating that operation is done for every row
game_df["Engagement_Score"] = game_df.apply(calculate_engagement_score, axis=1)

# Display the updated DataFrame with relevant columns
display(game_df[["Game_Score", "App_Rating", "Sessions", "Engagement_Score"]])
```

The following table shows a partial view of the updated `game_df`:

	Game_Score	App_Rating	Sessions	Engagement_Score
0	88.0	5.0	5	88.0
1	92.0	NaN	8	NaN
2	78.0	3.8	6	49.4
3	85.0	4.2	9	39.67
4	NaN	4.0	3	NaN
5	95.0	4.7	7	63.79
6	82.0	4.1	4	83.95

Why Use `apply`?

- **Row-Wise Operations:** The `apply` function allows operations involving multiple columns in a single row, such as combining `Game_Score`, `App_Rating`, and `Sessions`.
- **Custom Logic:** You can easily incorporate conditional statements into your transformations.
- **Flexibility:** Unlike `map`, which is limited to single-column transformations, `apply` can handle row-wise or column-wise custom computations across multiple columns.

`apply()` Method

The `apply()` method enables row-wise or column-wise application of functions in a DataFrame, making it ideal for custom transformations and calculations.

4.7 Other Useful Transformation Functions

In addition to the above methods, pandas provides other transformation functions that are valuable for preparing data:

- `to_datetime()`: Converts strings to datetime objects.
- `fillna()`: Fills missing values with specified values or methods.
- `cut()`: Bins numerical data into discrete intervals.

- `str`: Enables string operations for text data.
- `dt`: Facilitates operations on datetime objects.

Note: We will cover some of these capabilities in detail in subsequent sections.

5 Categorical Data Analysis

In this section, we'll analyze categorical data to better understand the distribution and unique values in columns like `Gender`, `Country`, and `Device`. We will use functions such as

- `.nunique()`
- `.unique()`
- `.value_counts()`

to explore these features. Understanding categorical data is essential in EDA as it reveals the diversity of values and common patterns that might be important for further analysis.

5.1 Counting Unique Values in Categorical Columns

The `.nunique()` function provides a quick count of unique values in each column. For categorical data, this helps us gauge the diversity of entries.

Counting Unique Values in Each Column

```
display(game_df.drop(columns=["Game_Type"]).nunique())
```

```
User_ID       7
Age           7
Gender        2
Country       2
Game_Score    6
Sessions      5
Device        3
App_Rating    5
```

This summary reveals that `Gender` and `Country` each contain only 2 unique values, confirming they are categorical variables. The `Device` column has 3 unique values, showing moderate diversity, while columns such as `User_ID` and `Age` have higher counts, reflecting their numerical nature; where unique values are expected to be higher due to their continuous or nearly continuous distributions. Additionally, note that we excluded the `Game_Type` column from this analysis because it contains lists, which are incompatible with `.nunique()` and would raise an error.

`nunique()` Method

The `nunique()` method returns the number of unique values in each column or row, helping identify diversity within data.

5.2 Displaying Unique Values in Specific Categorical Columns

We can use the `.unique()` function to list all unique values in a column. This helps in verifying the categories and ensuring no unexpected values are present.

Displaying Unique Values in Categorical Columns

```
print(game_df["Gender"].unique())
print(game_df["Country"].unique())
print(game_df["Device"].unique())
```

```
['F' 'M']
['US' 'CA']
['Mobile' 'Desktop' 'Tablet']
```

The output shows that `Gender` consists of two categories: `F` and `M`, `Country` contains two countries: `US` and `CA`, and `Device` includes three categories: `Mobile`, `Desktop`, and `Tablet`.

The `.unique()` function is very important for exploratory data analysis (EDA), especially for categorical variables. Some practical tips and considerations include:

- **Check for Typos or Anomalies:** If unexpected values appear (e.g., `'Male'` in a `Gender` column instead of `'M'`), it may indicate a data quality issue.
- **Understand Data Variability:** Knowing the range of categories helps in deciding the type of analysis to perform. For instance, `Gender` has only two categories in this limited dataset, so one-hot encoding may be appropriate.
- **Large Cardinality:** For columns with many unique values (e.g., `User_ID`), the output of `.unique()` may not be practical for display. In such cases, use `.nunique()` to count the number of unique values instead.
- **Integration with Filtering:** Combined with conditional filtering, `.unique()` helps in narrowing down subsets of data. For example:

  Combining using `unique()` with boolean indexing

  ```
  game_df[game_df["Device"] == "Mobile"]["Country"].unique()
  ```

  This finds the countries of users who use `Mobile` devices.

After identifying unique values, you can proceed to analyze their frequency using `.value_counts()` to gain a more detailed understanding of categorical distributions. This comes next!

`unique()` Method

The `unique()` method lists all distinct values in a column, making it useful for understanding categorical data or identifying unexpected values.

5.3 Frequency Distribution of Categorical Values

To examine the distribution of values within a categorical column, we use the `.value_counts()` function. The `.value_counts()` method returns a `Series` summarizing a column or subset of columns. It is widely used for

analyzing categorical variables as its output provides valuable insights for many data exploration questions.

Commonly used arguments of `value_counts()` include:

- `normalize`: Set to `True` to display proportions rather than raw frequencies.
- `subset`: Specifies multiple columns to use when counting unique values, useful for generating counts across categorical dimensions.

Let's see a few examples

The following code displays the counts of unique values in the `Device` column.

Counting Unique Devices

```
display(game_df["Device"].value_counts())
```

Device	Count
Mobile	4
Desktop	2
Tablet	1

This table shows that `Device` includes three distinct values, with `Mobile` being the most frequent.

To view the proportion of each `Device` category rather than raw counts, we can set `normalize=True` per below example:

Proportion of Each Device Type

```
display(game_df["Device"].value_counts(normalize=True))
```

Device	Proportion
Mobile	0.57
Desktop	0.29
Tablet	0.14

This output shows that 57% of users access the app via `Mobile`, etc.

The following example demonstrates how to use `value_counts()` across multiple columns, providing a breakdown by both `Device` and `Gender`.

Counts by Device and Gender

```
device_gender_counts = game_df.value_counts(subset=["Device", "Gender"])
display(device_gender_counts)
```

Device	Gender	Count
Mobile	F	4
Desktop	M	2
Tablet	M	1

The first two columns are index columns. This `MultiIndex` `Series` lets us see the breakdown of `Device` use by `Gender`, with counts for each combination.

The previous handout has introduced the concept of `MultiIndex`. Here is a brief reminder: a `MultiIndex` is a hierarchical index structure in `pandas` that allows multiple levels of indexing, enabling you to represent higher-dimensional data within a `DataFrame` or `Series`. In this case, the output of `value_counts()` when used with multiple columns creates a `Series` with a `MultiIndex`—an array of tuples where each tuple represents a unique combination of values in the specified columns.

To illustrate, let's examine the `device_gender_counts` output from the previous example. Running the following code displays the structure of the `MultiIndex` in this `Series`:

Exploring MultiIndex Structure

```
display(device_gender_counts.index)
```

This yields:

```
MultiIndex([('Mobile', 'F'),
('Desktop', 'M'),
('Tablet', 'M'),
names=['Device', 'Gender'])
```

Each entry in the `MultiIndex` corresponds to a unique combination of values from the `Device` and `Gender` columns. This structure enables intuitive access to data associated with specific combinations using the `.loc[]` accessor.

You can retrieve individual values or groups of values in a `MultiIndex` `Series` using `.loc[]` by specifying the desired combination of index values. For instance, to access the count of `Mobile` users who identify as `F`, use the following code:

Accessing MultiIndex Element

```
display(device_gender_counts.loc[("Mobile", "F")])
```

This yields:

```
4
```

This output indicates that there are `4` female users accessing the app via `Mobile`.

You can also retrieve data for multiple combinations by passing a list of tuples, as shown below:

Accessing Multiple MultiIndex Elements

```
display(device_gender_counts.loc[[("Mobile", "F"), ("Tablet", "M")]])
```

which produces:

Device	Gender	
Mobile	F	4
Tablet	M	1

`value_counts()` Method

The `value_counts()` method provides the frequency of each unique value in a column, offering insights into the distribution of categorical data.

5.4 Encoding Categorical Variables

Most practical datasets contain categorical variables, often stored as text values. In the `game_df` dataset, columns like `Device`, `Country`, and `Gender` are categorical. Many data analysis techniques, such as linear regression, cannot handle these text values directly, so categorical variables need to be converted into numerical formats. This process is called **encoding**, and it involves transforming text-based categories into numbers. Below are the three most commonly used encoding methods, along with guidelines for when to use each one.

Overview of Encoding Methods

- **Label Encoding**: A quick method to assign an integer code to each category level. Useful for ordinal data but can introduce unintended ordinal relationships if used on nominal data.
- **Find & Replace Encoding**: A custom encoding approach where you define mappings for each category manually. Provides flexibility for specific code assignments.
- **One-Hot Encoding**: Transforms each category level into a separate binary column, making it ideal for nominal data to avoid implying any order.

This section will focus on these three methods using examples from the `game_df` dataset.

5.4.1 Label Encoding (Illustration)

Label encoding assigns an integer code to each category level in a variable. This method is most suitable for ordinal data, where the order of categories carries meaning. Label encoding can be implemented in two steps:

- **Step 1:** Use `astype("category")` to convert the data type of the categorical variable to `category`.
- **Step 2:** Access the codes for each category level using `.cat.codes`.

Below, we apply label encoding to the `Device` column.

Caution: Note that `Device` does not have an inherent order, as we cannot say `Mobile` < `Tablet` < `Desktop`. This example is solely for illustration purposes. Label encoding is ideally suited for data with a meaningful order, such as `S`, `M`, and `L` in clothing sizes.

Label Encoding Device Column

```
# Step 1
game_df["Device"] = game_df["Device"].astype("category")

# Step 2
game_df["encoded_Device"] = game_df["Device"].cat.codes
```

To confirm the mapping of each code, use:

Viewing Category Mapping

```
cat_dict = dict(enumerate(game_df["Device"].cat.categories))
cat_dict
```

yields:

```
{0: 'Desktop', 1: 'Mobile', 2: 'Tablet'}
```

The partial `DataFrame`, shown below due to space limits, looks like this:

User_ID	Age	Device	encoded_Device	Game_Score	Sessions
101	23	Mobile	1	88.0	5
102	35	Desktop	0	92.0	8
103	45	Tablet	2	78.0	6
104	30	Mobile	1	85.0	9
105	25	Mobile	1	NaN	3
106	28	Desktop	0	95.0	7
107	33	Mobile	1	82.0	4

Interpretation: In this encoding, `Desktop` is represented by `0`, `Mobile` by `1`, and `Tablet` by `2`. Since `Device` has no natural order, using label encoding here can be misleading in models that interpret these codes as ordered data. For nominal data like `Device`, consider using one-hot encoding to avoid this issue.

Encoding Techniques

Label Encoding: Converts categories to integer values, ideal for ordinal data.

5.4.2 Method 2: Find and Replace Encoding

The find & replace method offers flexible, manual control over the numerical values assigned to each category. This method is especially helpful when specific codes need to be assigned for each category. This method can be implemented in two steps:

- **Step 1:** Create a dictionary mapping categories to their respective codes.
- **Step 2:** Use the `replace()` function to apply this mapping.

Here's how to encode the `Device` column:

Find and Replace Encoding Device Column

```
encoding_dict = {
    "Device": {"Mobile": 1, "Desktop": 2, "Tablet": 3}
```

```
}
game_df.replace(encoding_dict, inplace=True)
display(game_df[["Device"]])
```

The partial `DataFrame`, shown below due to space limits, looks like this:

User_ID	Device	Age	Country
101	1	23	US
102	2	35	CA
103	3	45	US
104	1	30	CA
105	1	25	US
106	2	28	US
107	1	33	US

Interpretation: Each device type in the `Device` column is now represented by a unique code, with `Mobile` encoded as `1`, `Desktop` as `2`, and `Tablet` as `3`. This approach allows for custom encoding, but remember that without a natural order in `Device`, one-hot encoding may be preferable to avoid false assumptions about hierarchy.

Encoding Techniques

Find and Replace: Replaces category values using a mapping dictionary, offering flexibility.

5.4.3 Method 3: One-Hot Encoding

One-hot encoding is particularly useful for nominal variables where no inherent order exists. This method creates a separate binary column for each category, marking its presence with `1` and absence with `0`. Use `pd.get_dummies()` to generate the one-hot encoded columns for the specified categorical variable.

Applying one-hot encoding to the `Device` column results in binary columns for each unique category, indicating whether each row's `Device` value matches a specific category.

One-Hot Encoding Device Column

```
game_df_one_hot = pd.get_dummies(game_df, columns=["Device"], dtype = int)
```

The partial `DataFrame`, showing selected columns for all rows, will look like this:

User_ID	Device	Device_Mobile	Device_Desktop	Device_Tablet
101	Mobile	1	0	0
102	Desktop	0	1	0
103	Tablet	0	0	1
104	Mobile	1	0	0
105	Mobile	1	0	0
106	Desktop	0	1	0
107	Mobile	1	0	0

Interpretation: Each unique value in `Device` now has its own column. A `1` in a column (e.g., `Device_Mobile`) indicates that the row corresponds to that device type, while a `0` indicates its absence. One-hot encoding prevents misinterpretation of categories as ordered values, making it well-suited for nominal variables.

Additional Encoding Methods: Beyond label, one-hot, and find-and-replace encoding, other techniques like **Binary Encoding**, **Frequency Encoding**, **Target Encoding**, and **Hash Encoding** are available for specialized use cases. Each method has unique trade-offs that may suit different data types or analysis goals.

Encoding Techniques

One-Hot Encoding: Creates binary columns for each category, useful for nominal variables.

6 Selecting and Filtering Data

When working with datasets, selecting and filtering specific rows and columns is a fundamental operation in data analysis. Previously, we have explored

- `.loc[]`
- `.iloc[]`

Now, let's learn the `query()` function. The `query()` function is a powerful tool for filtering rows based on conditions. It allows you to write conditions in a syntax similar to SQL, making the code more intuitive.

Basic Syntax

Basic syntax of `query()`

```
DataFrame.query("condition")
```

In this syntax:

- Column names are used directly without quotes (unless they contain special characters or spaces).
- Logical operators are written as `&` (AND), `|` (OR), and `~` (NOT).
- String comparisons are enclosed in quotes.

Let's see a few examples that show the use of `query()` for filtering data in the `game_df` dataset.
Let's select and display users with country equal to `US`

Querying for Users in the US

```
# Select rows where Country is US
us_users = game_df.query("Country == 'US'")
display(us_users)
```

yields

User_ID	Age	Gender	Country	Game_Score	Sessions	Device	App_Rating	Game_Type
101	23	F	US	88.0	5	Mobile	5.0	[Candy Crush, Warzone]
103	45	M	US	78.0	6	Tablet	3.8	[Super Mario, Elden Ring]
105	25	F	US	NaN	3	Mobile	4.0	[Roblox, Call of Duty]
106	28	M	US	95.0	7	Desktop	4.7	[Candy Crush, PUBG]
107	33	F	US	82.0	4	Mobile	4.1	[Among Us, Candy Crush]

As a second example, let's select users with `Game_Scores` Above 85 in `Mobile` Devices

Querying for High Scoring Mobile Users

```
# Select rows where Game_Score > 85 and Device is Mobile
high_scoring_mobile_users = game_df.query("Game_Score > 85 & Device == 'Mobile'")
display(high_scoring_mobile_users)
```

yielding

User_ID	Age	Gender	Country	Game_Score	Sessions	Device	App_Rating	Game_Type
101	23	F	US	88.0	5	Mobile	5.0	[Candy Crush, Warzone]

This filters users who achieved a `Game_Score` greater than 85 and used `Mobile` devices.
and finally, let's filter users by `Gender` and `Sessions`

Querying for Female Users with More Than 5 Sessions

```
# Select rows where Gender is F and Sessions > 5
active_female_users = game_df.query("Gender == 'F' & Sessions > 5")
display(active_female_users)
```

User_ID	Age	Gender	Country	Game_Score	Sessions	Device	App_Rating	Game_Type
104	30	F	CA	85.0	9	Mobile	4.2	[Minecraft, Among Us]

This query finds female users (`Gender == 'F'`) who had more than 5 gaming sessions.

6.0.1 Why Use `query()` compared to boolean indexing?

`query()` provides several benefits:

- **Readability:** Conditions are expressed in a concise and SQL-like syntax, making the code easier to read and write.

- **Chaining:** Integrates well into method chaining, enabling fluid and expressive data transformations.
- **Dynamic Queries:** Allows the use of variables in queries using the `@` symbol. For example:

Dynamic Queries

```
threshold = 85
game_df.query("Game_Score > @threshold")
```

Having explored `query()` in detail, you can now combine it with other `pandas` operations like `.groupby()` or `.agg()` for advanced filtering and summarization.

`query()` Function

The `query()` function filters rows of a `DataFrame` using a query string. It allows concise filtering with logical expressions, improving code readability.

7 Data Aggregation and Grouping

In data analysis, aggregating and grouping data is crucial for summarizing information, finding patterns, and extracting insights from specific subsets. This section introduces three powerful `pandas` methods for aggregation and grouping:

- `.groupby()`
- `.agg()`
- `.pivot_table()`.

Each has unique advantages and can handle various data manipulation tasks.

7.1 `.groupby()`: Grouping Data by Columns

The `.groupby()` function in `pandas` allows us to split data into groups based on one or more columns, then apply aggregate functions to these groups. This method is highly flexible and is commonly used to analyze patterns within subgroups.

7.1.1 Basic Usage

The basic syntax for `.groupby()` is:

Basic syntax of `groupby()`

```
df.groupby("column_name").aggregate_function()
```

In the context of the `game_df` dataset, we could use `.groupby()` to answer questions such as:

- What is the average `Game_Score` for each `Country`?
- How many users of each `Device` type are in the dataset?

Let's calculate average `Game_Score` for each `Country`. In order to do that, we need to group the dataset by the `Country` and then calculate the average score. The snippet is as below:

Grouping by Country and Calculating Average Game_Score

```
avg_game_score_by_country = game_df.groupby("Country")["Game_Score"].mean()
display(avg_game_score_by_country)
```

This code groups `game_df` by `Country` and calculates the average `Game_Score` for each country; yielding

```
Country
CA    88.5
US    85.8
Name: Game_Score, dtype: float64
```

7.1.2 Grouping by Multiple Columns

We can also group data by multiple columns to examine more detailed interactions. For instance, we could explore the average `Game_Score` for each `Device` type within each `Country` using

Grouping by Country and Device

```
avg_game_score_country_device = game_df.groupby(["Country", "Device"])["Game_Score"].mean()
display(avg_game_score_country_device)
```

yielding

```
Country  Device
CA       Desktop    92.0
         Mobile     85.0
         Tablet      NaN
US       Desktop    95.0
         Mobile     85.0
         Tablet     78.0
Name: Game_Score, dtype: float64
```

This output shows the average `Game_Score` for each `Device` type within each `Country`, helping us understand how `Device` usage and scores vary across regions.

7.1.3 Counting Observations in Each Group

`.groupby()` can also be used to count the number of entries in each group by applying the `count()` function as below:

Counting Users by Device

```
user_counts_by_device = game_df.groupby("Device")["User_ID"].count()
display(user_counts_by_device)
```

yielding

Device	
Desktop	2
Mobile	4
Tablet	1

This shows how many users are in each `Device` group, allowing us to assess device popularity.

`groupby()` Function

The `groupby()` function groups rows in a `DataFrame` based on column values, enabling aggregation or transformation operations such as `sum()`, `mean()`, or custom functions.

7.2 `.agg()`: Applying Multiple Aggregations

The `.agg()` function allows us to apply multiple aggregation functions to each group, which is useful when we want to calculate several statistics for a single column or across multiple columns. This flexibility makes `.agg()` particularly powerful for more complex data summarization.

7.2.1 Basic Syntax and Usage

The syntax for `.agg()` is:

```
df.groupby("column_name").agg({"col1": "func1", "col2": "func2"})
```

Here is an example:

Aggregating Multiple Statistics

```
summary_stats = game_df.groupby("Country").agg({
                                                "Game_Score": ["mean", "max"],
                                                "Sessions": "mean"})
display(summary_stats)
```

yielding

	Game_Score (mean)	Game_Score (max)	Sessions (mean)
Country			
CA	88.5	92.0	8.5
US	85.8	95.0	5.0

In this example:

- The mean and maximum `Game_Score` are calculated for each `Country`.
- The average number of `Sessions` is also calculated for each `Country`.

7.2.2 Custom Aggregations with Named Functions

We can also define custom functions and use them in `.agg()` to create tailored summaries; here is an example:

Custom Aggregation with Custom Functions

```
def score_range(input_s):
    """
    returning range of a Series

    """
    return input_s.max() - input_s.min()

# Using custom aggregation function in agg
score_summary = game_df.groupby("Device").agg({"Game_Score": ["mean", score_range]})
display(score_summary)
```

yielding

Device	Game_Score (mean)	score_range
Desktop	93.5	3.0
Mobile	85.0	6.0
Tablet	78.0	0.0

Here, the custom function `score_range` calculates the range (difference between max and min) for `Game_Score` for each `Device` type, showing variability in scores across devices.

`agg()` Function

The `agg()` function allows applying multiple aggregation functions (e.g., `mean`, `sum`) simultaneously to grouped data or specific columns in a flexible way.

7.3 `.pivot_table()`: Creating Summarized Tables

`.pivot_table()` is another powerful method for summarizing data. It's similar to Excel pivot tables and is ideal for creating multidimensional summaries. Unlike `.groupby()`, it can also handle missing values by filling them automatically with a specified value (e.g., 0).

7.3.1 Basic Syntax and Usage

The syntax for `.pivot_table()` is:

```
df.pivot_table(index="index_column",
               columns="column_name",
               values="value_column",
               aggfunc="func")
```

Let's see a few examples. Assume we are interested in finding the average `game_score` by `Country` and `Device`.

Pivot Table for Game_Score by Country and Device

```
# Creating a pivot table to find average Game_Score by Country and Device
pivot_table = game_df.pivot_table(index = "Country",
                                  columns = "Device",
                                  values = "Game_Score",
                                  aggfunc = "mean",
                                  fill_value = 0)
display(pivot_table)
```

yielding

Country	Desktop	Mobile	Tablet
CA	92.0	85.0	0.0
US	95.0	85.0	78.0

This pivot table shows the average `Game_Score` for each `Device` in each `Country`. Missing values are filled with 0, ensuring a complete table.

7.3.2 Using Multiple Aggregations in a Pivot Table

We can also use multiple aggregation functions within a pivot table to get a richer summary; here is an example:

Multiple Aggregations in Pivot Table

```
# Creating a pivot table with multiple aggregations
pivot_table_multi = game_df.pivot_table(index="Country",
                                        columns="Device",
                                        values=["Game_Score", "Sessions"],
                                        aggfunc={"Game_Score": "mean", "Sessions": "sum"},
                                        fill_value=0,
                                        margins = True)
display(pivot_table_multi)
```

The output is not displayed due to space limitations.
Here, the pivot table calculates:

- The mean `Game_Score` by `Country` and `Device`.
- The total `Sessions` for each `Country` and `Device` combination.
- It also provides the sum of each row and column due to `margins = True`.

`pivot_table()` Function

The `pivot_table()` function creates a spreadsheet-style pivot table for analyzing data. It supports multiple aggregation functions and allows handling missing values with `fill_value`.

8 Correlation

Understanding the relationships between variables is a crucial aspect of data analysis. Correlation measures the strength and direction of a linear relationship between two numerical variables. The `corr()` function in pandas calculates the *Pearson* correlation coefficient by default.

- A correlation value close to `1` indicates a strong positive linear relationship.
- A correlation value close to `-1` indicates a strong negative linear relationship.
- A correlation value close to `0` indicates no linear relationship.

Let's see an example:

Correlation Matrix for Game Data

```
# Select only numerical columns
# Dropped User ID as its numerical value holds no meaningful significance
numerical_game_df = game_df.select_dtypes(include=['number']).drop(columns = ['User_ID'])

# Compute correlation
correlation_matrix = numerical_game_df.corr()
display(correlation_matrix)
```

yielding

	Age	Game_Score	Sessions	App_Rating
Age	1.0	-0.6	0.3	-0.7
Game_Score	-0.6	1.0	0.4	0.8
Sessions	0.3	0.4	1.0	0.1
App_Rating	-0.7	0.8	0.1	1.0

Interpretation: The correlation matrix reveals the relationships between numerical columns. For instance:

- `Game_Score` and `App_Rating` have a strong positive correlation of `0.80`.
- `Age` and `App_Rating` have a strong negative correlation of `-0.7`.

`corr()` Function

The `corr()` function computes pairwise correlation coefficients between numerical columns. It supports methods like `'pearson'`, `'kendall'`, and `'spearman'` to evaluate linear or ranked relationships.

8.1 Visualizing Correlations with a Heatmap

To better understand the correlation matrix, we can visualize it using a heatmap. We will show using an external library like `seaborn` to generate this visualization.

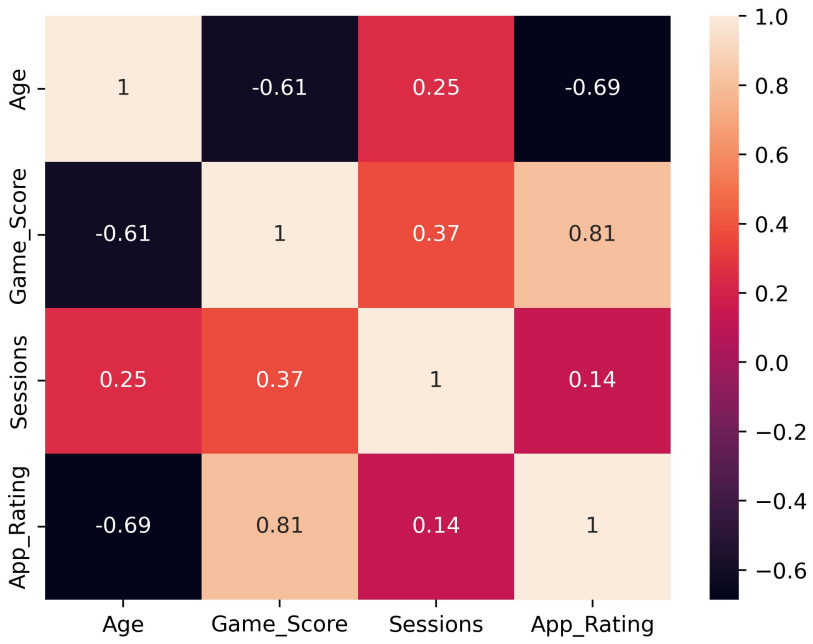

Figure 1: Heatmap

Heatmap for Correlation Matrix

```
import seaborn as sns

sns.heatmap(correlation_matrix, annot=True)
```

The heatmap assigns colors to correlation values, making it easier to interpret relationships.

9 Exercises

1. **Data Access:** Please download `Purchase_Records_Dataset.csv` from the course Github and read this `csv` file to a `DataFrame`.
2. **Dataset Overview:** What are the data types and the count of non-null values in each column?
3. **Summary Statistics:** What are the summary statistics for the numerical columns in the dataset?
4. **Missing Data:** Which columns have missing values, and how many missing values does each column have?
5. What is the total number of missing values in the dataset?
6. **Selecting Specific Data Types:** Select and display only the numerical columns in the dataset.
7. **Data Type Transformation:** Convert the `Age` column to an integer data type. Verify if the conversion was successful.
8. **Unique Values:** What are the unique regions and product categories in the dataset?
9. **Count of Unique Values:** How many unique customers are present in the dataset?
10. **Value Counts:** What are the most frequent product categories?
11. **Querying Data:** Which customers made purchases in the "North" region and paid using a "Credit Card"?
12. **Filtering with Queries:** Find all online orders for electronics with a discount greater than 10%.
13. **Grouping Data:** What is the average price of products purchased in each region?
14. **Aggregating Data:** Calculate the total quantity sold for each product category by region.
15. **Pivot Table:** Create a pivot table showing the average price of products for each payment method by region.
16. **Correlation Analysis:** What is the correlation between `Age`, `Quantity`, and `Price`? Visualize it using `seaborn`.
17. **Mapping Values:** Create a column categorizing `Age` into "Young" (`<40`), "Middle-aged" (`40-59`), and "Old" (`>=60`).
18. **Using Apply:** Create a column calculating the total value of a transaction (`Price * Quantity`) and apply a discount.
19. **Customer Analysis:** Which customers are returning customers (more than one transaction)? What percentage of all customers are returning customers?
20. **Lead Time Analysis:** What is the average lead time for online orders across different product categories?
21. **Region-Specific Trends:** Are discounts more frequent in one region compared to others? What is the average discount by region?
22. **Payment Method Preferences:** Which payment method is most commonly used for online purchases? Is there a significant difference between regions?
23. **Top Customers:** Identify the top 10 customers based on the total monetary value of their transactions.
24. **In-Store vs. Online Trends:** What is the average discount provided for in-store purchases compared to online purchases? Is one platform more likely to offer higher discounts?

10 References

References and Resources

The following references and resources were used in the preparation of these materials:

1. Official Python website at https://www.python.org/.
2. *Introduction to Computation and Programming Using Python*, John Guttag, The MIT Press, 2nd edition, 2016.
3. *Python for Data Science Handbook: Essential Tools for Working with Data*, Jake VanderPlas, O'Reilly Media, 1st edition, 2016.
4. *Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython*, Wes McKinney, O'Reilly Media, 2nd edition, 2017.
5. *Introduction to Python for Computer Science and Data Science*, Paul J. Deitel, Harvey Deitel, Pearson, 1st edition, 2019.
6. *Data Visualization in Python with Pandas and Matplotlib*, David Landup, Independently published, 2021.
7. *Python for Programmers with Introductory AI Case Studies*, Paul Deitel, Harvey Deitel, Pearson, 1st edition, 2019.
8. *Effective Pandas: Patterns for Data Manipulation (Treading on Python)*, Matt Harrison, Independently published, 2021.
9. *Introduction to Programming in Python; An Interdisciplinary Approach*, Robert Sedgewick, Kevin Wayne, Robert Dondero, Pearson, 1st edition, 2015.
10. Python tutorials at https://betterprogramming.pub/.
11. Python learning platform at https://www.learnpython.org/.
12. Python resources at https://realpython.com/.
13. Python courses and tutorials at https://www.datacamp.com/.