Module 4 | Concepts and Code Components

1. Data Loading and Initial Exploration
Importing Libraries:
Importing the necessary libraries is the first step in any data analysis task. Pandas is a powerful data manipulation library, while Matplotlib is used for creating visualizations. For this assignment, we need to import libraries like Pandas and Matplotlib because they provide essential tools for data manipulation and visualization, which are crucial for our assignment tasks.

```
import pandas as pd
import matplotlib.pyplot as plt
```

Loading a Dataset:
Loading data from an external source into a DataFrame is a crucial step. This involves reading the data from a CSV file, which is a common format for datasets. For this assignment, loading the dataset into a DataFrame allows us to work with the data in a structured way. This is the first step in data analysis, enabling us to inspect and manipulate the data easily.

```
url =
'https://raw.githubusercontent.com/CunyLaguardiaDataAnalytics/datasets/master/311_Service_Requests_from_2019May.csv'
df = pd.read_csv(url)
```

Exploring the Data:
Understanding the structure of the dataset is important. By printing the first few rows, checking the shape, and looking at the data types of each column, you get an overview of what your data looks like and how you can manipulate it. For this assignment, by printing the first few rows, checking the shape (number of rows and columns), and examining the data types, we get a basic understanding of the dataset's structure. This helps in planning further data manipulation and analysis steps.

```
print(df.head())  # First 5 rows
print(df.shape)   # Number of rows and columns
print(df.dtypes)  # Data types of each column
```

2. Basic Python Data Types within Dataframes
Variables and Data Types:
Variables are used to store data. Here, x, y, and z are variables storing different types of data. Knowing the type of data stored in a variable is important because different data types can be manipulated in different ways. In our assignment, we create variables to store specific values like the number of rows and columns. Understanding these data types (int, float, str) is essential because different operations apply to different data types.

```
x = df.shape[0]  # Number of rows
y = df.shape[1] / 2  # Number of columns divided by 2
z = 'NYC 311 Data'

print(type(x))  # int
print(type(y))  # float
print(type(z))  # str
```

Tuples and Dictionaries:
Tuples are immutable sequences, which means their values cannot be changed. Dictionaries are used to store data in key-value pairs, which is useful for organizing and accessing data quickly. Tuples and dictionaries are used to organize and store related data. In the assignment, we use them to store summary information about the dataset, which helps in quickly accessing and using these values in our analysis.

```
data_tuple = (x, y, z)
data_dict = {'rows': x, 'half_columns': y, 'title': z}

print(data_tuple)
print(data_dict)
```

3. Aggregating Data and Chaining Operations
Grouping Data:
The group by method in Pandas is used to split the data into groups based on some criteria. For example, grouping by 'Complaint Type' allows you to aggregate data and perform operations like counting the number of occurrences of each complaint type.

```
complaint_counts = df.groupby('Complaint Type').size()
print(complaint_counts)
```

Chaining Operations:
Chaining allows you to perform multiple operations in a single line of code, making your code more readable and concise. For example, you can filter rows and select specific columns in one go. In this assignment, Chaining allows us to perform multiple data manipulation steps in a single line of code. For instance, filtering rows for a specific borough and selecting certain columns helps streamline our data processing and makes the code more efficient and readable.

```
brooklyn_complaints = df[df['Borough'] == 'BROOKLYN'][['Complaint
Type', 'Resolution Description']]
print(brooklyn_complaints.head())
```

4. Filtering Columns and Rows
Filtering Columns:
Selecting specific columns from a DataFrame helps you focus on the data you need for analysis, making your dataset easier to work with. In this assignment, selecting specific columns (e.g., 'Complaint Type', 'Borough', 'Status') allows us to focus on relevant data for our analysis. This makes the dataset more manageable and helps in targeting our analysis on important aspects.

```
df_filtered_cols = df[['Complaint Type', 'Borough', 'Status']]
```

Filtering Rows:
Filtering rows based on conditions allows you to subset your data. For example, selecting rows where the status is 'Closed' or where a particular string is present in a column helps narrow down your dataset to the most relevant Information. In this assignment, Filtering rows based on conditions (e.g., 'Status' is 'Closed') helps narrow down the dataset to specific cases we are interested in analyzing. This step is crucial for focusing on relevant data points.

```
df_filtered_rows = df[df['Status'] == 'Closed']
df2 = df_filtered_rows.copy()

# Exact Match
noise_street = df[df['Complaint Type'] == 'Noise - Street/Sidewalk']

# Fuzzy Match
noise_resolution = df[df['Resolution Description'].str.contains('Noise',
na=False)]
```

5. Adding and Removing Columns
Adding a Column:
Creating new columns, such as calculating 'Request Length' by finding the difference between 'Created Date' and 'Closed Date', helps in deriving new insights from existing data. This is a common task in data analysis to enhance the dataset with additional useful information.

```
df['Created Date'] = pd.to_datetime(df['Created Date'])
df['Closed Date'] = pd.to_datetime(df['Closed Date'])
df['Request Length'] = (df['Closed Date'] - df['Created Date']).dt.days
print(df.head())
```

Removing a Column:
Removing columns that are no longer needed keeps the DataFrame clean and focused. This helps in reducing clutter and improving the efficiency of data processing.

```
df.drop('Request Length', axis=1, inplace=True)
```

Renaming Columns:
Renaming columns to more meaningful names improves the readability of the DataFrame. This is important for understanding and interpreting the data correctly, especially when presenting results.

```
df.rename(columns={'Complaint Type': 'Type', 'Created Date': 'Date'},
inplace=True)
print(df.head())
```

6. Creating Subsets and Dropping Rows
Creating a Subset:
Creating a subset of the DataFrame with only relevant columns ('Type', 'Borough', 'Date') helps in focusing on the essential data for our analysis. Dropping rows with unspecified values ensures data quality and relevance.

```
subset_df = df[['Type', 'Borough', 'Date']]
subset_df = subset_df[subset_df['Borough'] != 'Unspecified']
print(subset_df.head())
```

7. Conditional/Logic Tests on Dataframes
Filtering with Conditions:
In this assignment, Filtering data based on specific conditions (e.g., 'Type' is 'Illegal Parking' and 'Borough' is 'MANHATTAN') allows us to drill down into the data and analyze specific subsets that meet certain criteria. This is essential for targeted analysis and deriving specific insights.

```
filtered_subset_df = subset_df[(subset_df['Type'] == 'Illegal Parking') &
(subset_df['Borough'] == 'MANHATTAN')]
print(filtered_subset_df.head())
```

8. Visualizing Data using Matplotlib
Bar Chart:
Creating a bar chart to show the number of complaints per borough helps visualize the distribution of complaints across different areas. This makes it easier to identify patterns and trends.

```
borough_complaints = df['Borough'].value_counts()
borough_complaints.plot(kind='bar', figsize=(10, 6))
plt.xlabel('Borough')
plt.ylabel('Number of Complaints')
plt.title('Number of Complaints per Borough')
plt.show()
```

Line Plot:
A line plot showing the number of 'Illegal Parking' complaints over time helps in understanding temporal trends and patterns. This visualization is useful for identifying peaks and changes over time.

```
illegal_parking_df = df[df['Type'] == 'Illegal Parking']
illegal_parking_df.set_index('Date', inplace=True)
illegal_parking_df.resample('M').size().plot(kind='line', figsize=(10, 6))
plt.xlabel('Date')
plt.ylabel('Number of Illegal Parking Complaints')
plt.title('Illegal Parking Complaints Over Time')
plt.show()
```

9. Core Concepts

User Input:

Taking user input allows for interactive programs. The input function captures user input, which can be used to dynamically influence the program's behavior.

user_input = input("Enter a value: ")

* This function displays the prompt "Enter a value: " to the user and waits for the user to type something. Once the user presses Enter, the input function captures the entered value as a string.

user_input =:

* The captured input is then stored in the variable

user_input

* which can be used later in the program to dynamically influence the program's behavior.

Dictionaries:

Dictionaries are used to store data in key-value pairs. For the quiz, understanding how to retrieve keys (dict.keys()) and iterate over both keys and values (for key, value in dict.items()) is essential for manipulating dictionary data.

Retrieve keys: dict.keys()
Iterate over keys and values: for key, value in dict.items()

Looping:

Loops are used to repeat a block of code multiple times. In the quiz, knowing how a while loop operates (repeats as long as a condition is true) and initializing it correctly (while x <= 10:) is crucial for controlling the flow of your program.

while loop: Repeats code as long as a condition is true
Initialize while loop: while x <= 10:

Pandas Functions:

Grouping:

The groupby function is used to group data based on column values and apply aggregate functions, which is critical for summarizing and analyzing grouped data.

df.groupby()

Chaining operations:
Chaining operations using dot notation (df.operation1().operation2()) makes code concise and readable, allowing multiple transformations in one statement.

```
Use dot notation df.operation1().operation2()
```

Selecting columns:
Accessing specific columns (df['ColumnName']) helps focus on relevant data, which is necessary for targeted analysis.

```
df['ColumnName']
```

Filtering rows:
Applying conditions to filter rows (df[df['ColumnName'] > value]) helps in narrowing down data to relevant subsets.

```
df[df['ColumnName'] > value]
```

Creating subsets:
Creating a new DataFrame with selected columns (df[['Column1', 'Column2']]) helps in focusing the analysis on specific data points.

```
df[['Column1', 'Column2']]
```

Removing columns:
Dropping unnecessary columns (df.drop('ColumnName', axis=1)) keeps the DataFrame clean and manageable.

```
df.drop('ColumnName', axis=1)
```

Renaming columns:
Renaming columns (df.rename(columns={'OldName': 'NewName'})) improves readability and understanding of the data.

```
df.rename(columns={'OldName': 'NewName'})
```

Checking data types:
Knowing the data type of each column (df.dtypes) helps in selecting appropriate operations for data manipulation.

```
df.dtypes
```

Descriptive statistics:
‘df.describe()’ provides a summary of numerical columns, offering insights into data distribution and central tendency.

```
df.describe()
```

Resetting index:
Resetting the index (df.reset_index()) renumbers the rows, which can be useful after filtering or subsetting data.

```
df.reset_index()
```

Adding columns:
Combining multiple DataFrames (pd.concat([df1, df2])) is useful for merging datasets vertically.

```
df['NewColumn'] = value
```

Saving to CSV:
Exporting data to CSV (df.to_csv('filename.csv')) allows for easy sharing and further analysis.

```
df.to_csv('filename.csv')
```

Concatenating DataFrames:
Combining multiple DataFrames (pd.concat([df1, df2])) is useful for merging datasets vertically.

```
pd.concat([df1, df2])
```

Checking duplicates:
Identifying duplicate rows (df.duplicated()) ensures data integrity by highlighting repeated data entries.

```
df.duplicated()
```

Accessing elements with iloc:
Using iloc for integer-location based indexing (df.iloc[row, column]) accesses specific elements by their position.

```
df.iloc[row, column]
```

Sorting:
Sorting data (df.sort_values(by='ColumnName')) organizes the DataFrame, making it easier to analyze.

```
df.sort_values(by='ColumnName')
```

Calculating mean:
Finding the mean of a column (df['ColumnName'].mean()) provides a measure of central tendency, summarizing the data effectively.

```
df['ColumnName'].mean()
```