OVERVIEW:

1. **Choice of PySpark:**

Choosing PySpark over traditional Python aligns with the organization's future scalability needs. PySpark's distributed computing capabilities ensure it can handle large volumes of data efficiently as the company grows. By starting with PySpark early, the organization avoids the need for costly code refactoring later and streamlines maintenance efforts. This proactive approach future-proofs data processing pipelines, fosters codebase consistency, and ultimately leads to long-term cost savings.

2. **Use of Libraries:**

The use of libraries such as requests, json, pandas, and os reflects practicality in leveraging existing tools for data fetching, manipulation, and file management. These libraries are widely adopted in the Python ecosystem and provide robust functionality, reducing the need for custom implementations.
Efficient Data Retrieval:

3. **Efficient Data Retrieval**:

The code employs functions like fetch_data_from_api to retrieve data from external APIs. This approach streamlines the data retrieval process and encapsulates error handling, ensuring robustness in handling network errors or API responses.

4. **Structured Data Processing:**

PySpark's DataFrame API is utilized for structured data processing. This choice simplifies data manipulation tasks by providing a familiar SQL-like interface for filtering, transforming, and aggregating data. The objective is for streamlining data processing workflows and promoting code maintainability.

5. **Parallel Data Processing:**

The main DataFrame creation process (main_dataframecreation) demonstrates parallel data processing capabilities inherent in PySpark. By distributing data processing tasks across multiple workers, the team can efficiently handle large datasets in parallel, improving overall throughput and performance.

6. **Data Quality Assurance:**

Functions like write_dataframe_to_csv ensure data quality and consistency by writing clean and organized data to CSV files. This approach facilitates data validation and auditing, enabling stakeholders to trust the integrity of the data exported for further analysis or consumption.

7. **Optimized Performance:**

Techniques such as DataFrame caching (df = df.cache()) and drop duplicates (drop_duplicates) are employed to optimize performance and resource utilization. By caching intermediate results and removing redundant data, which allows computational efficiency and reduces overhead in subsequent data processing steps.

8. **Modular and Scalable Design:**

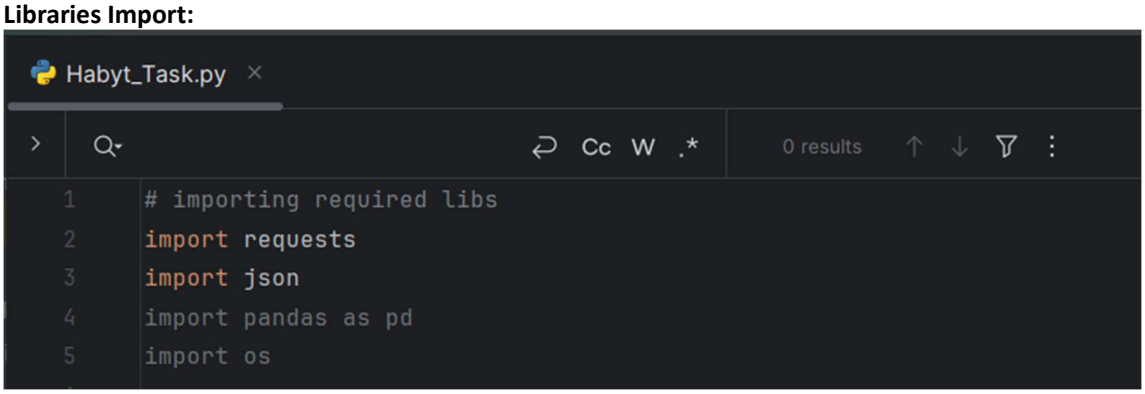
The code is modularized into functions, each responsible for a specific data processing task. This design promotes code reusability, maintainability, and scalability, allowing the team to easily extend functionality or incorporate additional data sources as needed.
Robust Error Handling:

9. **Robust Error Handling:**

The code incorporates error handling mechanisms to gracefully manage exceptions, such as network errors during data retrieval. By logging errors and providing informative messages, the team ensures visibility into potential issues and facilitates troubleshooting and debugging.
The Objectives are for scalability, performance optimization, data quality assurance, and maintainability.

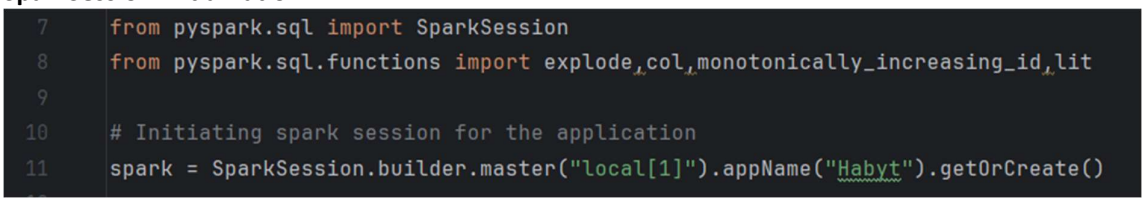
CODE LEVEL RATIONALE AND THE CHOICES:

1. **Libraries Import:**



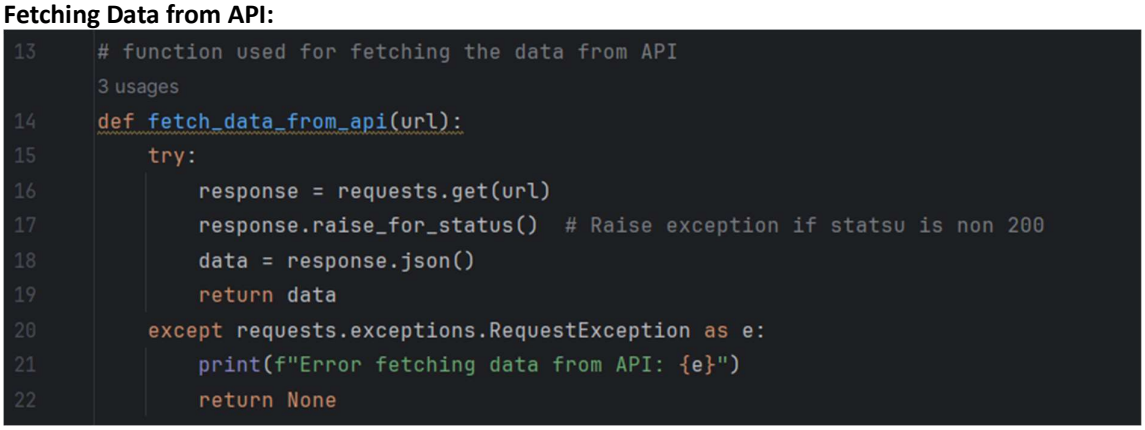
Line 2. requests: Used for making HTTP requests to fetch data from APIs.
Line 3. json: Utilized for handling JSON data.
Line4. pandas: it's a common library for data manipulation in Python.
Line5. os: Library for interacting with the operating system

2. **SparkSession Initialization:**



Line7. PySpark requires a SparkSession to be initialized for any Spark functionality. PySpark's SQL module, used for working with structured data.
Line 8. pyspark.sql.functions: Contains functions used for various DataFrame operations like column manipulation, aggregation, etc.
Line11. The SparkSession is created using SparkSession.builder with configurations for master URL and application name.

3. **Fetching Data from API:**



Line14 – line 22. The function fetch_data_from_api is defined to fetch JSON data from an API using the requests library. If successful, it returns the JSON data; otherwise, it prints an error message and returns None.

4. **Creating DataFrame from JSON:**

```
24  # Function to create the Dataframe from json which is coming from the api
    3 usages
25  def create_dataframe_from_json(spark, json_data):
26      if json_data:
27          rdd = spark.sparkContext.parallelize([json.dumps(json_data)])
28          df = spark.read.json(rdd)
29          return df
30      else:
31          print("JSON data is None. Unable to create DataFrame.")
32          return None
```

Line25. The function create_dataframe_from_json converts JSON data into a PySpark DataFrame. It first checks if JSON data is not empty, then converts it to an RDD (Resilient Distributed Dataset) and finally reads it as a DataFrame using spark.read.json.

5. **Main DataFrame Creation:**

```
34  # creating the main dataframe which will have all the listing data for all the cities from the cities API.
    1 usage
35  def main_dataframecreation(cities_df_c):
36      for row in cities_df_c.collect():
37          ct_name = row["urlSlug"]
38          API_URL_Listings = f"https://www.common.com/cmn-api/listings/common?city={ct_name}"
39          print(API_URL_Listings)
40          response = requests.get(API_URL_Listings)
41          data = response.json()
42          rdd = spark.sparkContext.parallelize([json.dumps(data)])
43          df = spark.read.json(rdd)
44          df = df.cache()
45      print(df.count())
46      return df
```

Line35-Line46. The function main_dataframecreation iterates over the cities DataFrame, constructs the API URL for listings using the city URL slug, fetches listing data from the API, and creates a DataFrame for each city's listings. It caches the DataFrame for performance optimization and returns the final DataFrame.

6. **DataFrame Transformation Functions:**

Line49 – Line300. Several functions are defined to transform the main DataFrame into various sub-dataframes based on specific requirements such as property data, listings, fees, pricing, addresses, cities, neighborhoods, images, applicants, deposits, and contracts. Each function takes the main DataFrame as input and returns the transformed DataFrame based on the specified logic.

7. **Writing DataFrames to CSV:**

```
    1 usage
302 def write_dataframe_to_csv(df, file_name):
303     file_path = f"/app/{file_name}.csv"
304     print("===Start writing files")
305     df = df.toPandas()
306     #Write the DataFrame to CSV with headers
307     df.to_csv(file_path,index=False,header=True)
308     print(f"DataFrame written to: {file_path}")
```

Line302 – Line308. The function write_dataframe_to_csv is defined to write each DataFrame to a CSV file. It converts the PySpark DataFrame to a Pandas DataFrame and then writes it to a CSV file with headers and without an index.

8. Main Execution:

Line310 – Line380. The script's main execution section fetches data from various APIs for cities, neighborhoods, and properties. It then transforms the data into multiple DataFrames using the defined functions and writes each DataFrame to a CSV file.